Package 'eplusr'

February 25, 2021

Title A Toolkit for Using Whole Building Simulation Program 'EnergyPlus'

Version 0.14.1

Description A rich toolkit of using the whole building simulation program 'EnergyPlus'(<https://energyplus.net>), which enables programmatic navigation, modification of 'EnergyPlus' models and makes it less painful to do parametric simulations and analysis.

License MIT + file LICENSE

URL https://hongyuanjia.github.io/eplusr/, https://github.com/hongyuanjia/eplusr

BugReports https://github.com/hongyuanjia/eplusr/issues

Depends R (>= 3.2.0)

Imports callr (>= 2.0.4), checkmate, cli (>= 1.1.0), crayon, data.table (>= 1.12.4), hms, lubridate, methods, processx (>= 3.2.0), progress (>= 1.2.0), R6, RSQLite, stringi, units

Suggests covr, decido, knitr, pkgdown, rgl (>= 0.100.26), rmarkdown, testthat (>= 2.1.0)

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

SystemRequirements EnergyPlus (>= 8.3, optional) (<https://energyplus.net>); udunits2

Collate 'constants.R' 'assert.R' 'diagram.R' 'eplusr.R' 'utils.R' 'impl.R' 'parse.R' 'impl-epw.R' 'impl-idd.R' 'impl-idf.R' 'idf.R' 'idd.R' 'epw.R' 'err.R' 'format.R' 'geom.R' 'group.R' 'iddobj.R' 'idfobj-sch.R' 'impl-idfobj.R' 'idfobj.R' 'impl-geom.R' 'impl-iddobj.R' 'impl-idfobj-sch.R' 'impl-sql.R' 'impl-viewer.R' 'install.R' 'job.R' 'options.R' 'param.R' 'rdd.R' 'reload.R' 'run.R' 'sql.R' 'transition.R' 'units.R' 'validate.R' 'viewer.R' 'zzz.R'

NeedsCompilation no

Author Hongyuan Jia [aut, cre] (<https://orcid.org/0000-0002-0075-8183>)

Maintainer Hongyuan Jia <hongyuan.jia@bears-berkeley.sg>

Repository CRAN

Date/Publication 2021-02-25 15:40:03 UTC

R topics documented:

`eplusr-package` *eplusr: A Toolkit for Using EnergyPlus in R*

Description

A rich toolkit of using the whole building simulation program 'EnergyPlus'(<https://energyplus.net>), which enables programmatic navigation, modification of 'EnergyPlus' models and makes it less painful to do parametric simulations and analysis.

Details

eplusr provides a rich toolkit of using EnergyPlus directly in R, which enables programmatic navigation, modification of EnergyPlus models and makes it less painful to do parametric simulations and analysis.

Features

- Download and install EnergyPlus in R
- Read, parse and modify EnergyPlus:
  - Input Data File (IDF)
  - Weather File (EPW)
  - Report Data Dictionary (RDD) & Meter Data Dictionary (MDD)
  - Error File (ERR)
- Modify multiple versions of IDFs and run corresponding EnergyPlus both in the background and in the front
- Rich-featured interfaces to query and modify IDFs
- Automatically handle referenced fields and validate input during modification
- Take fully advantage of most common used data structure for data science in R – data.frame
  - Extract model, weather data into data.frames
  - Modify multiple objects via data.frames input
  - Query output via SQL in Tidy format which is much better for data analysis and visualization
- Provide a simple yet extensible prototype of conducting parametric simulations and collect all results in one go
- A pure R-based version updater `transition()` which is much faster than VersionUpdater distributed with EnergyPlus

Author(s)

Hongyuan Jia

See Also

Useful links:

- https://hongyuanjia.github.io/eplusr/
- https://github.com/hongyuanjia/eplusr
- Report bugs at https://github.com/hongyuanjia/eplusr/issues

---

`as.character.IddObject` *Coerce an IddObject into a Character Vector*

Description

Coerce an IddObject into an empty object of current class in a character vector format. It is formatted exactly the same as in IDF Editor.

Usage

```
## S3 method for class 'IddObject'
as.character(x, comment = NULL, leading = 4L, sep_at = 29L, all = FALSE, ...)
```

Arguments

`x`	An IddObject object.
`comment`	A character vector to be used as comments of returned string format object. If `NULL`, no comments are inserted. Default: `NULL`.
`leading`	Leading spaces added to each field. Default: `4`.
`sep_at`	The character width to separate value string and field string. Default: `29` which is the same as IDF Editor.'
`all`	If `TRUE`, all fields in current class are returned, otherwise only minimum fields are returned.
`...`	Further arguments passed to or from other methods.

Value

A character vector.

Examples

```
## Not run:
as.character(use_idd(8.8, download = "auto")$Materal, leading = 0)

## End(Not run)
```

`as.character.Idf` *Coerce an Idf object into a Character Vector*

Description

Coerce an Idf object into a character vector.

Usage

```
## S3 method for class 'Idf'
as.character(
  x,
  comment = TRUE,
  header = TRUE,
  format = eplusr_option("save_format"),
  leading = 4L,
  sep_at = 29L,
  ...
)
```

Arguments

`x`	An Idf object.
`comment`	If `FALSE`, all comments will not be included. Default: `TRUE`.
`header`	If `FALSE`, the header will not be included. Default: `TRUE`.
`format`	Specific format used when formatting. For details, please see $save(). Default: `eplusr_option("save_format")`
`leading`	Leading spaces added to each field. Default: `4L`.
`sep_at`	The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.
`...`	Further arguments passed to or from other methods.

Value

A character vector.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
idf_path <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")
as.character(read_idf(idf_path, use_idd(8.8, "auto")), leading = 0)

## End(Not run)
```

`as.character.IdfObject` *Coerce an IdfObject into a Character Vector*

Description

Coerce an IdfObject into a character vector in the same way as in IDF Editor.

Usage

```
## S3 method for class 'IdfObject'
as.character(x, comment = TRUE, leading = 4L, sep_at = 29L, all = FALSE, ...)
```

Arguments

`x`	An IdfObject object.
`comment`	If `FALSE`, all comments will not be included. Default: `TRUE`.
`leading`	Leading spaces added to each field. Default: `4L`.
`sep_at`	The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.
`all`	If `TRUE`, values of all possible fields in current class the IdfObject belongs to are returned. Default: `FALSE`
`...`	Further arguments passed to or from other methods.

Value

A character vector.

Examples

```
## Not run:
idf <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"),
    idd = use_idd(8.8, download = "auto"))

# get the IdfObject of material named "C5 - 4 IN HW CONCRETE"
mat <- idf$Material[["C5 - 4 IN HW CONCRETE"]]

as.character(mat, leading = 0, sep_at = 10)

## End(Not run)
```

clean_wd *Clean working directory of a previous EnergyPlus simulation*

Description

Clean working directory of an EnergyPlus simulation by deleting all input and output files of previous simulation.

Usage

```
clean_wd(path)
```

Arguments

path	An `.idf` or `.imf` file path.

Details

`clean_wd()` imitates the same process that EnergyPlus does whenever a new simulation is getting to start. It deletes all related output files that have the same name prefix as the input path. The input model itself and any weather file are not deleted. `clean_wd()` is called internally when running EnergyPlus models using `run_idf()` and `run_multi()`.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
# run a test simulation
idf_path <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")
epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData",
     "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
)
dir <- file.path(tempdir(), "test")
run_idf(idf_path, epw_path, output_dir = dir, echo = FALSE)

list.files(dir)

# remove all output files
clean_wd(file.path(dir, basename(idf_path)))

list.files(dir)

## End(Not run)
```

custom_validate *Customize validation components*

Description

`custom_validate()` makes it easy to customize what validation components should be included during IDF object modifications using $dup(), $add(), $set() and other methods in Idf class.

Usage

```
custom_validate(
  required_object = FALSE,
  unique_object = FALSE,
  unique_name = FALSE,
  extensible = FALSE,
  required_field = FALSE,
  auto_field = FALSE,
  type = FALSE,
  choice = FALSE,
  range = FALSE,
  reference = FALSE
)
```

Arguments

required_object	Check if required objects are missing in current model. Default: FALSE.
unique_object	Check if there are multiple objects in one unique-object class. Default: FALSE.
unique_name	Check if all objects in every class have unique names. Default: FALSE.
extensible	Check if all fields in an extensible group have values. Default: FALSE.
required_field	Check if all required fields have values. Default: FALSE.
auto_field	Check if all fields with value "Autosize" and "Autocalculate" are valid or not. Default: FALSE.
type	Check if all fields have values with valid types, i.e. character, numeric and integer fields should be filled with corresponding type of values. Default: FALSE.
choice	Check if all choice fields have valid choice values. Default: FALSE.
range	Check if all numeric fields have values within defined ranges. Default: FALSE.
reference	Check if all fields whose values refer to other fields are valid. Default: FALSE.

Details

There are 10 different validation check components in total. Three predefined validation level are included, i.e. "none", "draft" and "final". To get what validation components those levels contain, see level_checks().

Value

A named list with 10 elements.

Examples

```
custom_validate(unique_object = TRUE)

# only check unique name during validation
eplusr_option(validate_level = custom_validate(unique_name = TRUE))
```

`download_weather` *Download EnergyPlus Weather File (EPW) and Design Day File (DDY)*

Description

`download_weather()` makes it easy to download EnergyPlus weather files (EPW) and design day files (DDY).

Usage

```
download_weather(
  pattern,
  filename = NULL,
  dir = ".",
  type = c("all", "epw", "ddy"),
  ask = TRUE,
  max_match = 3
)
```

Arguments

`pattern`	A regular expression used to search locations, e.g. `"los angeles.*tmy3"`. The search is case-insensitive.
`filename`	File names (without extension) used to save downloaded files. Internally, `make.unique()` is called to ensure unique names.
`dir`	Directory to save downloaded files
`type`	File type to download. Should be one of `"all"`, `"epw"` and `"ddy"`. If `"all"`, both weather files and design day files will be downloaded.
`ask`	If `TRUE`, a command line menu will be shown to let you select which one to download. If `FALSE` and the number of returned results is less than `max_match`, files are downloaded automatically without asking.
`max_match`	The max results allowed to download when `ask` is `FALSE`.

Value

A character vector containing paths of downloaded files.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
download_weather("los angeles.*tmy3", "LosAngeles", tempdir(), ask = FALSE)

## End(Not run)
```

`dt_to_load` *Format Long Table to Standard Input for* `Idf$load()` *Method*

Description

`dt_to_load()` takes a data.table, usually created from `Idf$to_table()` or `IdfObject$to_table()` with `wide` being `TRUE`, and format it into a data.table in acceptable format for $load() method in Idf class.

Usage

```
dt_to_load(dt, string_value = TRUE)
```

Arguments

`dt`	A data.table created using `Idf$to_table()` and `IdfObject$to_table()`. `dt` should at least contain column `id` (indicator used to distinguish object definitions), `class` (class names). If a `name` column exists, it will be preserved.
`string_value`	If `TRUE`, all value will be coerced into character and the `value` column of returned datat.table will be character type. If `FALSE`, the original value will be preserved and the `value` column of returned data.table will be list type.

Value

A data.table with 5 or 6 columns:

- `id`: Integer type. Used to distinguish each object definition.
- `name`: Character type. Only exists when input `dt` has a `name` column.
- `class`: Character type.
- `index`: Integer type. Field indices.
- `field`: Character type. Field names.
- `value`: Character or list type. The value of each field to be added.

Examples

```
## Not run:
# read an example distributed with eplusr
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")
idf <- read_idf(path_idf)

# extract all material object data and return it as a wide table
dt <- idf$to_table(class = "Material", wide = TRUE)

dt_to_load(dt)

## End(Not run)
```

`empty_idf` *Create an Empty Idf*

Description

`empty_idf()` takes a valid IDD version and creates an empty Idf object that only contains a Version object.

Usage

```
empty_idf(ver = "latest")
```

Arguments

`ver`	Any acceptable input of `use_idd()`. If `latest`, which is the default, the latest IDD released version is used.

Value

An Idf object

Examples

```
## Not run:
if (is_avail_idd(8.8)) empty_idf(8.8)

## End(Not run)
```

EplusGroupJob *Create and Run Parametric Analysis, and Collect Results*

Description

EplusGroupJob class is a wrapper of run_multi() and provides an interface to group multiple EnergyPlus simulations together for running and collecting outputs.

group_job() takes IDFs and EPWs as input and returns a EplusGroupJob.

Usage

```
group_job(idfs, epws)
```

Arguments

idfs	Paths to EnergyPlus IDF files or a list of IDF files and Idf objects.
epws	Paths to EnergyPlus EPW files or a list of EPW files and Epw objects. Each element in the list can be NULL, which will force design-day-only simulation when $run() method is called. Note this needs at least one Sizing:DesignDay object exists in that Idf. If epws is NULL, design-day-only simulation will be conducted for all input models.

Value

A EplusGroupJob object.

Methods

Public methods:

- EplusGroupJob$new()
- EplusGroupJob$run()
- EplusGroupJob$kill()
- EplusGroupJob$status()
- EplusGroupJob$errors()
- EplusGroupJob$output_dir()
- EplusGroupJob$locate_output()
- EplusGroupJob$list_table()
- EplusGroupJob$read_table()
- EplusGroupJob$read_rdd()
- EplusGroupJob$read_mdd()
- EplusGroupJob$report_data_dict()
- EplusGroupJob$report_data()
- EplusGroupJob$tabular_data()
- EplusGroupJob$print()

Method `new()`**:** Create an `EplusGroupJob` object

Usage:

```
EplusGroupJob$new(idfs, epws)
```

Arguments:

`idfs` Paths to EnergyPlus IDF files or a list of IDF files and Idf objects. If only one IDF supplied, it will be used for simulations with all EPWs.

`epws` Paths to EnergyPlus EPW files or a list of EPW files and Epw objects. Each element in the list can be `NULL`, which will force design-day-only simulation. Note this needs at least one `Sizing:DesignDay` object exists in that Idf. If `epws` is `NULL`, design-day-only simulation will be conducted for all input models. If only one EPW supplied, it will be used for simulations with all IDFs.

Returns: An `EplusGroupJob` object.

Examples:

```
\dontrun{
if (is_avail_eplus(8.8)) {
    dir <- eplus_config(8.8)$dir
    path_idfs <- list.files(file.path(dir, "ExampleFiles"), "\\.idf",
        full.names = TRUE)[1:5]
    path_epws <- list.files(file.path(dir, "WeatherData"), "\\.epw",
        full.names = TRUE)[1:5]

    # create from local files
    group <- group_job(path_idfs, path_epws)

    # create from Idfs and Epws object
    group_job(lapply(path_idfs, read_idf), lapply(path_epws, read_epw))
}
}
```

Method `run()`**:** Run grouped simulations

Usage:

```
EplusGroupJob$run(
  dir = NULL,
  wait = TRUE,
  force = FALSE,
  copy_external = FALSE,
  echo = wait,
  separate = TRUE
)
```

Arguments:

`dir` The parent output directory for specified simulations. Outputs of each simulation are placed in a separate folder under the parent directory.

`wait` If `TRUE`, R will hang on and wait all EnergyPlus simulations finish. If `FALSE`, all EnergyPlus simulations are run in the background. Default: `TRUE`.

`force` Only applicable when the last simulation runs with `wait` equals to `FALSE` and is still running. If `TRUE`, current running job is forced to stop and a new one will start. Default: `FALSE`.

`copy_external` If `TRUE`, the external files that current `Idf` object depends on will also be copied into the simulation output directory. The values of file paths in the Idf will be changed automatically. This ensures that the output directory will have all files needed for the model to run. Default is `FALSE`.

`echo` Only applicable when `wait` is `TRUE`. Whether to simulation status. Default: same as `wait`.

`separate` If `TRUE`, all models are saved in a separate folder with each model's name under `dir` when simulation. If `FALSE`, all models are saved in `dir` when simulation. Default: `TRUE`.

Details: $run() runs all grouped simulations in parallel. The number of parallel EnergyPlus process can be controlled by `eplusr_option("num_parallel")`. If `wait` is FALSE, then the job will be run in the background. You can get updated job status by just printing the `EplusGroupJob` object.

Returns: The `EplusGroupJob` object itself, invisibly.

Examples:

```
\dontrun{
# only run design day
group$run(NULL)

# do not show anything in the console
group$run(echo = FALSE)

# specify output directory
group$run(tempdir(), echo = FALSE)

# run in the background
group$run(wait = TRUE, echo = FALSE)
# see group job status
group$status()

# force to kill background group job before running the new one
group$run(force = TRUE, echo = FALSE)

# copy external files used in the model to simulation output directory
group$run(copy_external = TRUE, echo = FALSE)
}
```

Method `kill()`**:** Kill current running jobs

Usage:

```
EplusGroupJob$kill()
```

Details: $kill() kills all background EnergyPlus processes that are current running if possible. It only works when simulations run in non-waiting mode.

Returns: A single logical value of `TRUE` or `FALSE`, invisibly.

Examples:

```
\dontrun{
group$kill()
}
```

Method `status()`**:** Get the group job status

Usage:

```
EplusGroupJob$status()
```

Details: $status() returns a named list of values indicates the status of the job:

- `run_before`: `TRUE` if the job has been run before. `FALSE` otherwise.
- `alive`: `TRUE` if the job is still running in the background. `FALSE` otherwise.
- `terminated`: `TRUE` if the job was terminated during last simulation. `FALSE` otherwise. `NA` if the job has not been run yet.
- `successful`: `TRUE` if all simulations ended successfully. `FALSE` if there is any simulation failed. `NA` if the job has not been run yet.
- `changed_after`: `TRUE` if the models has been modified since last simulation. `FALSE` otherwise.
- `job_status`: A `data.table::data.table()` contains meta data for each simulation job. For details, please see `run_multi()`. If the job has not been run before, a `data.table::data.table()` with 4 columns is returned:
  - `index`: The index of simulation
  - `status`: The status of simulation. As the simulation has not been run, `status` will always be "idle".
  - `idf`: The path of input IDF file.
  - `epw`: The path of input EPW file. If not provided, `NA` will be assigned.

Returns: A named list of 6 elements.

Examples:

```
\dontrun{
group$status()
}
```

Method `errors()`**:** Read group simulation errors

Usage:

```
EplusGroupJob$errors(which = NULL, info = FALSE)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`info` If `FALSE`, only warnings and errors are printed. Default: `FALSE`.

Details: $errors() returns a list of ErrFile objects which contain all contents of the simulation error files (`.err`). If `info` is `FALSE`, only warnings and errors are printed.

Returns: A list of ErrFile objects.

Examples:

```
\dontrun{
group$errors()

# show all information
group$errors(info = TRUE)
}
```

Method `output_dir()`**:** Get simulation output directory

Usage:

```
EplusGroupJob$output_dir(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: $output_dir() returns the output directory of simulation results.

Returns: A character vector.

Examples:

```
\dontrun{
# get output directories of all simulations
group$output_dir()

# get output directories of specified simulations
group$output_dir(c(1, 4))
}
```

Method `locate_output()`**:** Get paths of output file

Usage:

```
EplusGroupJob$locate_output(which = NULL, suffix = ".err", strict = TRUE)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`suffix` A string that indicates the file extension of simulation output. Default: `".err"`.

`strict` If `TRUE`, it will check if the simulation was terminated, is still running or the file exists or not. Default: `TRUE`.

Details: $locate_output() returns the path of a single output file of specified simulations.

Returns: A character vector.

Examples:

```
\dontrun{
# get the file path of the error file
group$locate_output(c(1, 4), ".err", strict = FALSE)

# can detect if certain output file exists
group$locate_output(c(1, 4), ".expidf", strict = TRUE)
}
```

Method `list_table()`**:** List all table names in EnergyPlus SQL outputs

Usage:

```
EplusGroupJob$list_table(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: $list_table() returns a list of character vectors that contain all available table and view names in the EnergyPlus SQLite files for group simulations. The list is named using IDF names.

Returns: A named list of character vectors.

Examples:

```
\dontrun{
group$list_table(c(1, 4))
}
```

Method `read_table()`**:** Read the same table from EnergyPlus SQL outputs

Usage:

```
EplusGroupJob$read_table(which = NULL, name)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`name` A single string specifying the name of table to read.

Details: $read_table() takes a simulation index and a valid table `name` of those from `$list_table()` and returns that table data in a `data.table::data.table()` format. The two column will always be `index` and `case` which can be used to distinguish output from different simulations. `index` contains the indices of simulated models and `case` contains the model names without extensions.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
# read a specific table
group$read_table(c(1, 4), "Zones")
}
```

Method `read_rdd()`**:** Read Report Data Dictionary (RDD) files

Usage:

```
EplusGroupJob$read_rdd(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: $read_rdd() return the core data of Report Data Dictionary (RDD) files. For details, please see `read_rdd()`. The two column will always be `index` and `case` which can be used to distinguish output from different simulations. `index` contains the indices of simulated models and `case` contains the model names without extensions.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
group$read_rdd(c(1, 4))
}
```

Method `read_mdd()`**:** Read Meter Data Dictionary (MDD) files

Usage:

```
EplusGroupJob$read_mdd(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: $read_mdd() return the core data of Meter Data Dictionary (MDD) files. For details, please see `read_mdd()`. The two column will always be `index` and `case` which can be used to distinguish output from different simulations. `index` contains the indices of simulated models and `case` contains the model names without extensions.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
group$read_mdd(c(1, 4))
}
```

Method `report_data_dict()`**:** Read report data dictionary from EnergyPlus SQL outputs

Usage:

```
EplusGroupJob$report_data_dict(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: $report_data_dict() returns a `data.table::data.table()` which contains all information about report data.
For details on the meaning of each columns, please see "2.20.2.1 ReportDataDictionary Table" in EnergyPlus "Output Details and Examples" documentation.

Returns: A `data.table::data.table()` of 10 columns:

- `index`: The index of simulated model. This column can be used to distinguish output from different simulations
- `case`: The model name without extension. This column can be used to distinguish output from different simulations

- report_data_dictionary_index: The integer used to link the dictionary data to the variable data. Mainly useful when joining different tables
- is_meter: Whether report data is a meter data. Possible values: 0 and 1
- timestep_type: Type of data timestep. Possible values: Zone and HVAC System
- key_value: Key name of the data
- name: Actual report data name
- reporting_frequency:
- schedule_name: Name of the the schedule that controls reporting frequency.
- units: The data units

Examples:

```
\dontrun{
group$report_data_dict(c(1, 4))
}
```

Method `report_data()`**:** Read report data

Usage:

```
EplusGroupJob$report_data(
  which = NULL,
  key_value = NULL,
  name = NULL,
  year = NULL,
  tz = "UTC",
  all = FALSE,
  wide = FALSE,
  period = NULL,
  month = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  interval = NULL,
  simulation_days = NULL,
  day_type = NULL,
  environment_name = NULL
)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If NULL, results of all parametric simulations are returned. Default: NULL.

`key_value` A character vector to identify key values of the data. If NULL, all keys of that variable will be returned. key_value can also be data.frame that contains key_value and name columns. In this case, name argument in $report_data() is ignored. All available key_value for current simulation output can be obtained using $report_data_dict(). Default: NULL.

`name` A character vector to identify names of the data. If NULL, all names of that variable will be returned. If key_value is a data.frame, name is ignored. All available name for current simulation output can be obtained using $report_data_dict(). Default: NULL.

`year` Year of the date time in column `datetime`. If `NULL`, it will calculate a year value that meets the start day of week restriction for each environment. Default: `NULL`.

`tz` Time zone of date time in column `datetime`. Default: `"UTC"`.

`all` If `TRUE`, extra columns are also included in the returned `data.table::data.table()`.

`wide` If `TRUE`, the output is formatted in the same way as standard EnergyPlus csv output file.

`period` A Date or POSIXt vector used to specify which time period to return. The year value does not matter and only month, day, hour and minute value will be used when subsetting. If `NULL`, all time period of data is returned. Default: `NULL`.

`month, day, hour, minute` Each is an integer vector for month, day, hour, minute subsetting of `datetime` column when querying on the SQL database. If `NULL`, no subsetting is performed on those components. All possible `month`, `day`, `hour` and `minute` can be obtained using `$read_table(NULL, "Time")`. Default: `NULL`.

`interval` An integer vector used to specify which interval length of report to extract. If `NULL`, all interval will be used. Default: `NULL`.

`simulation_days` An integer vector to specify which simulation day data to extract. Note that this number resets after warmup and at the beginning of an environment period. All possible `simulation_days` can be obtained using `$read_table(NULL, "Time")`. If `NULL`, all simulation days will be used. Default: `NULL`.

`day_type` A character vector to specify which day type of data to extract. All possible day types are: `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Holiday`, `SummerDesignDay`, `WinterDesignDay`, `CustomDay1`, and `CustomDay2`. All possible values for current simulation output can be obtained using `$read_table(NULL, "Time")`. A few grouped options are also provided:

- `"Weekday"`: All working days, i.e. from Monday to Friday
- `"Weekend"`: Saturday and Sunday
- `"DesignDay"`: Equivalent to `"SummerDesignDay"` plus `"WinterDesignDay"`
- `"CustomDay"`: CustomDay1 and CustomDay2
- `"SpecialDay"`: Equivalent to `"DesignDay"` plus `"CustomDay"`
- `"NormalDay"`: Equivalent to `"Weekday"` and `"Weekend"` plus `"Holiday"`

`environment_name` A character vector to specify which environment data to extract. If `NULL`, all environment data are returned. Default: `NULL`. All possible `environment_name` for current simulation output can be obtained using:

```
$read_table(NULL, "EnvironmentPeriods")
```

`case` If not `NULL`, a character column will be added indicates the case of this simulation. If `"auto"`, the name of the IDF file without extension is used.

Details: $report_data() extracts the report data in a `data.table::data.table()` using key values, variable names and other specifications.
$report_data() can also directly take all or subset output from $report_data_dict() as input, and extract all data specified.
The returned column numbers varies depending on `all` argument.

- `all` is `FALSE`, the returned `data.table::data.table()` has 6 columns:
  - `index`: The index of simulated model. This column can be used to distinguish output from different simulations
  - `case`: The model name. This column can be used to distinguish output from different simulations

  - datetime: The date time of simulation result
  - key_value: Key name of the data
  - name: Actual report data name
  - units: The data units
  - value: The data value
- all is TRUE, besides columns described above, extra columns are also included:
  - month: The month of reported date time
  - day: The day of month of reported date time
  - hour: The hour of reported date time
  - minute: The minute of reported date time
  - dst: Daylight saving time indicator. Possible values: 0 and 1
  - interval: Length of reporting interval
  - simulation_days: Day of simulation
  - day_type: The type of day, e.g. Monday, Tuesday and etc.
  - environment_period_index: The indices of environment.
  - environment_name: A text string identifying the environment.
  - is_meter: Whether report data is a meter data. Possible values: 0 and 1
  - type: Nature of data type with respect to state. Possible values: Sum and Avg
  - index_group: The report group, e.g. Zone, System
  - timestep_type: Type of data timestep. Possible values: Zone and HVAC System
  - reporting_frequency: The reporting frequency of the variable, e.g. HVAC System Timestep, Zone Timestep.
  - schedule_name: Name of the the schedule that controls reporting frequency.

With the datetime column, it is quite straightforward to apply time-series analysis on the simulation output. However, another painful thing is that every simulation run period has its own Day of Week for Start Day. Randomly setting the year may result in a date time series that does not have the same start day of week as specified in the RunPeriod objects.
eplusr provides a simple solution for this. By setting year to NULL, which is the default behavior, eplusr will calculate a year value (from current year backwards) for each run period that compliances with the start day of week restriction.
It is worth noting that EnergyPlus uses 24-hour clock system where 24 is only used to denote midnight at the end of a calendar day. In EnergyPlus output, "00:24:00" with a time interval being 15 mins represents a time period from "00:23:45" to "00:24:00", and similarly "00:15:00" represents a time period from "00:24:00" to "00:15:00" of the next day. This means that if current day is Friday, day of week rule applied in schedule time period "00:23:45" to "00:24:00" (presented as "00:24:00" in the output) is also Friday, but not Saturday. However, if you try to get the day of week of time "00:24:00" in R, you will get Saturday, but not Friday. This introduces inconsistency and may cause problems when doing data analysis considering day of week value.
With wide equals TRUE, $report_data() will format the simulation output in the same way as standard EnergyPlus csv output file. Sometimes this can be useful as there may be existing tools/workflows that depend on this format. When both wide and all are TRUE, columns of runperiod environment names and date time components are also returned, including: environment_period_index", "environment_name, simulation_days, datetime, month, day, hour, minute, day_type.
For convenience, input character arguments matching in $report_data() are **case-insensitive**.

Returns: A data.table::data.table().

Examples:

```
\dontrun{
# read report data
group$report_data(c(1, 4))

# specify output variables using report data dictionary
dict <- group$report_data_dict(1)
group$report_data(c(1, 4), dict[units == "C"])

# specify output variables using 'key_value' and 'name'
group$report_data(c(1, 4), "environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
group$report_data(c(1, 4), dict[1], year = 2020, tz = "Etc/GMT+8")

# get all possible columns
group$report_data(c(1, 4), dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
group$report_data(c(1, 4), dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
group$report_data(c(1, 4), dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
group$report_data(c(1, 4), dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)
}
```

Method `tabular_data()`**:** Read tabular data

Usage:

```
EplusGroupJob$tabular_data(
  which = NULL,
  report_name = NULL,
  report_for = NULL,
  table_name = NULL,
  column_name = NULL,
  row_name = NULL,
  wide = FALSE,
  string_value = !wide
)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`report_name, report_for, table_name, column_name, row_name` Each is a character vector for subsetting when querying the SQL database. For the meaning of each argument, please see the description above.

`wide` If `TRUE`, each table will be converted into the similar format as it is shown in EnergyPlus HTML output file. Default: `FALSE`.

`string_value` Only applicable when `wide` is `TRUE`. If `string_value` is `FALSE`, instead of keeping all values as characters, values in possible numeric columns are converted into numbers. Default: the opposite of `wide`. Possible numeric columns indicate column that:

- columns that have associated units
- columns that contents numbers

Details: $tabular_data() extracts the tabular data in a `data.table::data.table()` using report, table, column and row name specifications. The returned `data.table::data.table()` has 9 columns:

- `index`: The index of simulated model. This column can be used to distinguish output from different simulations
- `case`: The model name. This column can be used to distinguish output from different simulations
- `index`: Tabular data index
- `report_name`: The name of the report that the record belongs to
- `report_for`: The `For` text that is associated with the record
- `table_name`: The name of the table that the record belongs to
- `column_name`: The name of the column that the record belongs to
- `row_name`: The name of the row that the record belongs to
- `units`: The units of the record
- `value`: The value of the record **in string format** by default

For convenience, input character arguments matching in $tabular_data() are **case-insensitive**.

Returns: A `data.table::data.table()` with 9 columns (when `wide` is `FALSE`) or a named list of `data.table::data.table()`s where the names are the combination of `report_name`, `report_for` and `table_name`.

Examples:

```
\dontrun{
# read all tabular data
group$tabular_data(c(1, 4))

# explicitly specify data you want
str(group$tabular_data(c(1, 4),
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))

# get tabular data in wide format and coerce numeric values
str(group$tabular_data(c(1, 4),
    report_name = "AnnualBuildingUtilityPerformanceSummary",
```

```
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy",
    wide = TRUE, string_value = FALSE
))
}
```

Method `print()`**:** Print `EplusGroupJob` object

Usage:

```
EplusGroupJob$print()
```

Details: $print() shows the core information of this `EplusGroupJob`, including the path of IDFs and EPWs and also the simulation job status.
$print() is quite useful to get the simulation status, especially when `wait` is `FALSE` in $run(). The job status will be updated and printed whenever $print() is called.

Returns: The `EplusGroupJob` object itself, invisibly.

Examples:

```
\dontrun{
group$print()
}
```

Author(s)

Hongyuan Jia

See Also

`eplus_job()` for creating an EnergyPlus single simulation job.

Examples

```
## ------------------------------------------------
## Method `EplusGroupJob$new`
## ------------------------------------------------

## Not run:
if (is_avail_eplus(8.8)) {
    dir <- eplus_config(8.8)$dir
    path_idfs <- list.files(file.path(dir, "ExampleFiles"), "\\.idf",
        full.names = TRUE)[1:5]
    path_epws <- list.files(file.path(dir, "WeatherData"), "\\.epw",
        full.names = TRUE)[1:5]

    # create from local files
    group <- group_job(path_idfs, path_epws)

    # create from Idfs and Epws object
```

```
    group_job(lapply(path_idfs, read_idf), lapply(path_epws, read_epw))
}

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$run`
## ------------------------------------------------

## Not run:
# only run design day
group$run(NULL)

# do not show anything in the console
group$run(echo = FALSE)

# specify output directory
group$run(tempdir(), echo = FALSE)

# run in the background
group$run(wait = TRUE, echo = FALSE)
# see group job status
group$status()

# force to kill background group job before running the new one
group$run(force = TRUE, echo = FALSE)

# copy external files used in the model to simulation output directory
group$run(copy_external = TRUE, echo = FALSE)

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$kill`
## ------------------------------------------------

## Not run:
group$kill()

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$status`
## ------------------------------------------------

## Not run:
group$status()

## End(Not run)
```

```
## ------------------------------------------------
## Method `EplusGroupJob$errors`
## ------------------------------------------------

## Not run:
group$errors()

# show all information
group$errors(info = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$output_dir`
## ------------------------------------------------

## Not run:
# get output directories of all simulations
group$output_dir()

# get output directories of specified simulations
group$output_dir(c(1, 4))

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$locate_output`
## ------------------------------------------------

## Not run:
# get the file path of the error file
group$locate_output(c(1, 4), ".err", strict = FALSE)

# can detect if certain output file exists
group$locate_output(c(1, 4), ".expidf", strict = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$list_table`
## ------------------------------------------------

## Not run:
group$list_table(c(1, 4))

## End(Not run)


## ------------------------------------------------
```

```
## Method `EplusGroupJob$read_table`
## ------------------------------------------------

## Not run:
# read a specific table
group$read_table(c(1, 4), "Zones")

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$read_rdd`
## ------------------------------------------------

## Not run:
group$read_rdd(c(1, 4))

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$read_mdd`
## ------------------------------------------------

## Not run:
group$read_mdd(c(1, 4))

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$report_data_dict`
## ------------------------------------------------

## Not run:
group$report_data_dict(c(1, 4))

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$report_data`
## ------------------------------------------------

## Not run:
# read report data
group$report_data(c(1, 4))

# specify output variables using report data dictionary
dict <- group$report_data_dict(1)
group$report_data(c(1, 4), dict[units == "C"])

# specify output variables using 'key_value' and 'name'
```

```
group$report_data(c(1, 4), "environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
group$report_data(c(1, 4), dict[1], year = 2020, tz = "Etc/GMT+8")

# get all possible columns
group$report_data(c(1, 4), dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
group$report_data(c(1, 4), dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
group$report_data(c(1, 4), dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
group$report_data(c(1, 4), dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$tabular_data`
## ------------------------------------------------

## Not run:
# read all tabular data
group$tabular_data(c(1, 4))

# explicitly specify data you want
str(group$tabular_data(c(1, 4),
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))

# get tabular data in wide format and coerce numeric values
str(group$tabular_data(c(1, 4),
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy",
    wide = TRUE, string_value = FALSE
))

## End(Not run)


## ------------------------------------------------
## Method `EplusGroupJob$print`
## ------------------------------------------------
```

```
## Not run:
group$print()

## End(Not run)
```

EplusJob *Run EnergyPlus Simulation and Collect Outputs*

Description

EplusJob class wraps the EnergyPlus command line interface and provides methods to extract simulation outputs.

eplus_job() takes an IDF and EPW as input, and returns an EplusJob object for running EnergyPlus simulation and collecting outputs.

Usage

```
eplus_job(idf, epw)
```

Arguments

idf	A path to an local EnergyPlus IDF file or an Idf object.
epw	A path to an local EnergyPlus EPW file or an Epw object. epw can also be NULL which will force design-day-only simulation when $run() method is called. Note this needs at least one Sizing:DesignDay object exists in the Idf.

Details

eplusr uses the EnergyPlus SQL output for extracting simulation outputs.

EplusJob has provide some wrappers that do SQL query to get report data results, i.e. results from Output:Variable and Output:Meter*. But for Output:Table results, you have to be familiar with the structure of the EnergyPlus SQL results, especially for table *"TabularDataWithStrings"*. For details, please see *"2.20 eplusout.sql"*, especially *"2.20.4.4 TabularData Table"* in EnergyPlus *"Output Details and Examples"* documentation. An object in Output:SQLite with Option Type value of SimpleAndTabular will be automatically created if it does not exists, to ensure that the output collection functionality works successfully.

In order to make sure .rdd (Report Data Dictionary) and .mdd (Meter Data Dictionary) files are created during simulation, an object in Output:VariableDictionary class with Key Field value being IDF will be automatically created if it does not exists.

Value

An EplusJob object.

Methods

Public methods:

- EplusJob$new()
- EplusJob$version()
- EplusJob$path()
- EplusJob$run()
- EplusJob$kill()
- EplusJob$status()
- EplusJob$errors()
- EplusJob$output_dir()
- EplusJob$locate_output()
- EplusJob$read_rdd()
- EplusJob$read_mdd()
- EplusJob$list_table()
- EplusJob$read_table()
- EplusJob$report_data_dict()
- EplusJob$report_data()
- EplusJob$tabular_data()
- EplusJob$print()

Method `new()`**:** Create an `EplusJob` object

Usage:

```
EplusJob$new(idf, epw)
```

Arguments:

`idf` Path to an local EnergyPlus IDF file or an Idf object.

`epw` Path to an local EnergyPlus EPW file or an Epw object.

Returns: An `EplusJob` object.

Examples:

```
\dontrun{
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    job <- eplus_job(idf_path, epw_path)

    # create from an Idf and an Epw object
    job <- eplus_job(read_idf(idf_path), read_epw(epw_path))
}
}
```

Method `version()`**:** Get the version of IDF in current job

Usage:

```
EplusJob$version()
```

Details: $version() returns the version of IDF that current `EplusJob` uses.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
job$version()
}
```

Method `path()`**:** Get the paths of file that current `EpwSql` uses

Usage:

```
EplusJob$path(type = c("all", "idf", "epw"))
```

Arguments:

`type` If `"all"`, both the Idf path and Epw path are returned. If `"idf"`, only IDF path is returned. If `"epw"`, only EPW path is returned. If `epw` is `NULL`, `NA` is returned for EPW path. Default: `"all"`.

Details: $path() returns the path of IDF or EPW of current job.

Returns: A character vector.

Examples:

```
\dontrun{
job$path()
job$path("idf")
job$path("epw")
}
```

Method `run()`**:** Run simulationA

Usage:

```
EplusJob$run(
  epw,
  dir = NULL,
  wait = TRUE,
  force = FALSE,
  echo = wait,
  copy_external = FALSE
)
```

Arguments:

`epw` A path to an `.epw` file or an Epw object. `epw` can also be `NULL` which will force design-day-only simulation. Note this needs at least one `Sizing:DesignDay` object exists in the `Idf`. If not given, the `epw` input used when creating this `EplusJob` object will be used.

`dir` The directory to save the simulation results. If `NULL`, the input `idf` folder will be used. Default: `NULL`.

`wait` If `TRUE`, R will hang on and wait for the simulation to complete. EnergyPlus standard output (stdout) and error (stderr) is printed to R console. If `FALSE`, simulation will be run in a background process. Default: `TRUE`.

`force` Only applicable when the last job runs with `wait` equals to `FALSE` and is still running. If `TRUE`, current running job is forced to stop and a new one will start. Default: `FALSE`.

`echo` Only applicable when `wait` is `TRUE`. Whether to show standard output and error from EnergyPlus. Default: same as `wait`.

`copy_external` If `TRUE`, the external files that current `Idf` object depends on will also be copied into the simulation output directory. The values of file paths in the Idf will be changed automatically. This ensures that the output directory will have all files needed for the model to run. Default is `FALSE`.

Details: $run() runs the simulation using input IDF and EPW file. If `wait` is `FALSE`, the job is run in the background. You can get updated job status by just printing the `EplusJob` object. Parameter `epw` can be used to reset the EPW file to use for simulation. If not given, the `epw` input used when creating this `EplusJob` object will be used.

Returns: The `EplusJob` object itself, invisibly.

Examples:

```
\dontrun{
# only run design day
job$run(NULL)

# specify output directory
job$run(dir = tempdir())

# run in the background
job$run(wait = TRUE)
# see job status
job$status()

# force to kill background job before running the new one
job$run(force = TRUE)

# do not show anything in the console
job$run(echo = FALSE)

# copy external files used in the model to simulation output directory
job$run(copy_external = TRUE)
}
```

Method `kill()`**:** Kill current running job

Usage:

```
EplusJob$kill()
```

Details: $kill() kills the background EnergyPlus process if possible. It only works when simulation runs in non-waiting mode.

Returns: A single logical value of `TRUE` or `FALSE`, invisibly.

Examples:

```
\dontrun{
job$kill()
}
```

Method `status()`**:** Get the job status

Usage:

```
EplusJob$status()
```

Details: $status() returns a named list of values that indicates the status of the job:

- `run_before`: `TRUE` if the job has been run before. `FALSE` otherwise.
- `alive`: `TRUE` if the simulation is still running in the background. `FALSE` otherwise.
- `terminated`: `TRUE` if the simulation was terminated during last simulation. `FALSE` otherwise. `NA` if the job has not been run yet.
- `successful`: `TRUE` if last simulation ended successfully. `FALSE` otherwise. `NA` if the job has not been run yet.
- `changed_after`: `TRUE` if the IDF file has been changed since last simulation. `FALSE` otherwise. `NA` if the job has not been run yet.

Returns: A named list of 5 elements.

Examples:

```
\dontrun{
job$status()
}
```

Method `errors()`**:** Read simulation errors

Usage:

```
EplusJob$errors(info = FALSE)
```

Arguments:

`info` If `FALSE`, only warnings and errors are printed. Default: `FALSE`.

Details: $errors() returns an ErrFile object which contains all contents of the simulation error file (`.err`). If `info` is `FALSE`, only warnings and errors are printed.

Returns: An ErrFile object.

Examples:

```
\dontrun{
job$errors()

# show all information
job$errors(info = TRUE)
}
```

Method `output_dir()`**:** Get simulation output directory

Usage:

```
EplusJob$output_dir(open = FALSE)
```

Arguments:

`open` If `TRUE`, the output directory will be opened.

Details: $output_dir() returns the output directory of simulation results.

Examples:

```
\dontrun{
job$output_dir()

# Below will open output directory
# job$output_dir(open = TRUE)
}
```

Method `locate_output()`**:** Get path of output file

Usage:

```
EplusJob$locate_output(suffix = ".err", strict = TRUE)
```

Arguments:

`suffix` A string that indicates the file extension of simulation output. Default: `".err"`.

`strict` If `TRUE`, it will check if the simulation was terminated, is still running or the file exists or not. Default: `TRUE`.

Details: $locate_output() returns the path of a single output file specified by file suffix.

Examples:

```
\dontrun{
# get the file path of the error file
job$locate_output(".err", strict = FALSE)

# can use to detect if certain output file exists
job$locate_output(".expidf", strict = TRUE)
}
```

Method `read_rdd()`**:** Read Report Data Dictionary (RDD) file

Usage:

```
EplusJob$read_rdd()
```

Details: $read_rdd() return the core data of Report Data Dictionary (RDD) file. For details, please see `read_rdd()`.

Returns: An RddFile object.

Examples:

```
\dontrun{
job$read_rdd()
}
```

Method `read_mdd()`**:** Read Report Data Dictionary (RDD) file

Usage:

```
EplusJob$read_mdd()
```

Details: $read_mdd() return the core data of Meter Data Dictionary (MDD) file. For details, please see `read_mdd()`.

Returns: An MddFile object.

Examples:

```
\dontrun{
job$read_mdd()
}
```

Method `list_table()`**:** List all table names in EnergyPlus SQL output

Usage:

```
EplusJob$list_table()
```

Details: $list_table() returns all available table and view names in the EnergyPlus SQLite file.

Returns: A character vector

Examples:

```
\dontrun{
job$list_table()
}
```

Method `read_table()`**:** Read a single table from EnergyPlus SQL output

Usage:

```
EplusJob$read_table(name)
```

Arguments:

`name` A single string specifying the name of table to read.

Details: $read_table() takes a valid table `name` of those from `$list_table()` and returns that table data in a `data.table::data.table()` format.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
# read a specific table
job$read_table("Zones")
}
```

Method `report_data_dict()`**:** Read report data dictionary from EnergyPlus SQL output

Usage:

```
EplusJob$report_data_dict()
```

Details: $report_data_dict() returns a `data.table::data.table()` which contains all information about report data.
For details on the meaning of each columns, please see "2.20.2.1 ReportDataDictionary Table" in EnergyPlus "Output Details and Examples" documentation.

Returns: A `data.table::data.table()` of 10 columns:

- `report_data_dictionary_index`: The integer used to link the dictionary data to the variable data. Mainly useful when joining different tables
- `is_meter`: Whether report data is a meter data. Possible values: `0` and `1`
- `timestep_type`: Type of data timestep. Possible values: `Zone` and HVAC System
- `key_value`: Key name of the data
- `name`: Actual report data name
- `reporting_frequency`: Data reporting frequency
- `schedule_name`: Name the the schedule that controls reporting frequency.
- `units`: The data units

Examples:

```
\dontrun{
job$report_data_dict()
}
```

Method `report_data()`**:** Read report data

Usage:

```
EplusJob$report_data(
  key_value = NULL,
  name = NULL,
  year = NULL,
  tz = "UTC",
  case = "auto",
  all = FALSE,
  wide = FALSE,
  period = NULL,
  month = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  interval = NULL,
  simulation_days = NULL,
  day_type = NULL,
  environment_name = NULL
)
```

Arguments:

`key_value` A character vector to identify key values of the data. If `NULL`, all keys of that variable will be returned. `key_value` can also be data.frame that contains `key_value` and `name` columns. In this case, `name` argument in $report_data() is ignored. All available `key_value` for current simulation output can be obtained using `$report_data_dict()`. Default: `NULL`.

`name` A character vector to identify names of the data. If `NULL`, all names of that variable will be returned. If `key_value` is a data.frame, `name` is ignored. All available `name` for current simulation output can be obtained using `$report_data_dict()`. Default: `NULL`.

`year` Year of the date time in column `datetime`. If `NULL`, it will calculate a year value that meets the start day of week restriction for each environment. Default: `NULL`.

`tz` Time zone of date time in column `datetime`. Default: `"UTC"`.

`case` If not `NULL`, a character column will be added indicates the case of this simulation. If `"auto"`, the name of the IDF file without extension is used.

`all` If `TRUE`, extra columns are also included in the returned `data.table::data.table()`.

`wide` If `TRUE`, the output is formatted in the same way as standard EnergyPlus csv output file.

`period` A Date or POSIXt vector used to specify which time period to return. The year value does not matter and only month, day, hour and minute value will be used when subsetting. If `NULL`, all time period of data is returned. Default: `NULL`.

`month, day, hour, minute` Each is an integer vector for month, day, hour, minute subsetting of `datetime` column when querying on the SQL database. If `NULL`, no subsetting is performed on those components. All possible `month`, `day`, `hour` and `minute` can be obtained using `$read_table("Time")`. Default: `NULL`.

`interval` An integer vector used to specify which interval length of report to extract. If `NULL`, all interval will be used. Default: `NULL`.

`simulation_days` An integer vector to specify which simulation day data to extract. Note that this number resets after warmup and at the beginning of an environment period. All possible `simulation_days` can be obtained using `$read_table("Time")`. If `NULL`, all simulation days will be used. Default: `NULL`.

`day_type` A character vector to specify which day type of data to extract. All possible day types are: `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Holiday`, `SummerDesignDay`, `WinterDesignDay`, `CustomDay1`, and `CustomDay2`. All possible values for current simulation output can be obtained using `$read_table("Time")`.

`environment_name` A character vector to specify which environment data to extract. If `NULL`, all environment data are returned. Default: `NULL`. All possible `environment_name` for current simulation output can be obtained using:

```
$read_table("EnvironmentPeriods")
```

Details: $report_data() extracts the report data in a `data.table::data.table()` using key values, variable names and other specifications.
$report_data() can also directly take all or subset output from $report_data_dict() as input, and extract all data specified.
The returned column numbers varies depending on `all` argument.

- `all` is `FALSE`, the returned `data.table::data.table()` has 6 columns:
  - `case`: Simulation case specified using `case` argument
  - `datetime`: The date time of simulation result
  - `key_value`: Key name of the data
  - `name`: Actual report data name
  - `units`: The data units
  - `value`: The data value
- `all` is `TRUE`, besides columns described above, extra columns are also included:
  - `month`: The month of reported date time

- day: The day of month of reported date time
- hour: The hour of reported date time
- minute: The minute of reported date time
- dst: Daylight saving time indicator. Possible values: 0 and 1
- interval: Length of reporting interval
- simulation_days: Day of simulation
- day_type: The type of day, e.g. Monday, Tuesday and etc.
- environment_period_index: The indices of environment.
- environment_name: A text string identifying the environment.
- is_meter: Whether report data is a meter data. Possible values: 0 and 1
- type: Nature of data type with respect to state. Possible values: Sum and Avg
- index_group: The report group, e.g. Zone, System
- timestep_type: Type of data timestep. Possible values: Zone and HVAC System
- reporting_frequency: The reporting frequency of the variable, e.g. HVAC System Timestep, Zone Timestep.
- schedule_name: Name of the the schedule that controls reporting frequency.

With the datetime column, it is quite straightforward to apply time-series analysis on the simulation output. However, another painful thing is that every simulation run period has its own Day of Week for Start Day. Randomly setting the year may result in a date time series that does not have the same start day of week as specified in the RunPeriod objects.
eplusr provides a simple solution for this. By setting year to NULL, which is the default behavior, eplusr will calculate a year value (from current year backwards) for each run period that compliances with the start day of week restriction.
It is worth noting that EnergyPlus uses 24-hour clock system where 24 is only used to denote midnight at the end of a calendar day. In EnergyPlus output, "00:24:00" with a time interval being 15 mins represents a time period from "00:23:45" to "00:24:00", and similarly "00:15:00" represents a time period from "00:24:00" to "00:15:00" of the next day. This means that if current day is Friday, day of week rule applied in schedule time period "00:23:45" to "00:24:00" (presented as "00:24:00" in the output) is also Friday, but not Saturday. However, if you try to get the day of week of time "00:24:00" in R, you will get Saturday, but not Friday. This introduces inconsistency and may cause problems when doing data analysis considering day of week value.
With wide equals TRUE, $report_data() will format the simulation output in the same way as standard EnergyPlus csv output file. Sometimes this can be useful as there may be existing tools/workflows that depend on this format. When both wide and all are TRUE, columns of runperiod environment names and date time components are also returned, including: environment_period_index", "environment_name, simulation_days, datetime, month, day, hour, minute, day_type.
For convenience, input character arguments matching in $report_data() are **case-insensitive**.

Returns: A data.table::data.table().

Examples:

```
\dontrun{
# read all report data
job$report_data()
```

```
# specify output variables using report data dictionary
dict <- job$report_data_dict()
job$report_data(dict[units == "C"])

# specify output variables using 'key_value' and 'name'
job$report_data("environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
job$report_data(dict[1], year = 2020, tz = "Etc/GMT+8")

# explicitly specify case name
job$report_data(dict[1], case = "example")

# get all possible columns
job$report_data(dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
job$report_data(dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
job$report_data(dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
job$report_data(dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)

# only get specified run period data
job$read_table("EnvironmentPeriods") # possible environment name
job$report_data(dict[1], environment_name = "San Francisco Intl Ap CA USA TMY3 WMO#=724940")
# can also be done using 'environment_period_index' column
job$report_data(dict[1], all = TRUE)[environment_period_index == 3L]
}
```

Method `tabular_data()`**:** Read tabular data

Usage:

```
EplusJob$tabular_data(
  report_name = NULL,
  report_for = NULL,
  table_name = NULL,
  column_name = NULL,
  row_name = NULL,
  wide = FALSE,
  string_value = !wide
)
```

Arguments:

`report_name`, `report_for`, `table_name`, `column_name`, `row_name` Each is a character vector for subsetting when querying the SQL database. For the meaning of each argument,

please see the description above.

`wide` If `TRUE`, each table will be converted into the similar format as it is shown in EnergyPlus HTML output file. Default: `FALSE`.

`string_value` Only applicable when `wide` is `TRUE`. If `string_value` is `FALSE`, instead of keeping all values as characters, values in possible numeric columns are converted into numbers. Default: the opposite of `wide`. Possible numeric columns indicate column that:

- columns that have associated units
- columns that contents numbers

Details: $tabular_data() extracts the tabular data in a `data.table::data.table()` using report, table, column and row name specifications. The returned `data.table::data.table()` has 9 columns:

- `case`: Simulation case specified using `case` argument
- `index`: Tabular data index
- `report_name`: The name of the report that the record belongs to
- `report_for`: The `For` text that is associated with the record
- `table_name`: The name of the table that the record belongs to
- `column_name`: The name of the column that the record belongs to
- `row_name`: The name of the row that the record belongs to
- `units`: The units of the record
- `value`: The value of the record **in string format** by default

For convenience, input character arguments matching in $tabular_data() are **case-insensitive**.

Returns: A `data.table::data.table()` with 8 columns (when `wide` is `FALSE`) or a named list of `data.table::data.table()`s where the names are the combination of `report_name`, `report_for` and `table_name`.

Examples:

```
\dontrun{
# read all tabular data
job$tabular_data()

# explicitly specify data you want
str(job$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))

# get tabular data in wide format and coerce numeric values
str(job$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy",
    wide = TRUE, string_value = FALSE
))
```

```
}
```

Method `print()`**:** Print `EplusSql` object

Usage:

```
EplusJob$print()
```

Details: $print() shows the core information of this `EplusJob` object, including the path of model and weather, the version and path of EnergyPlus used to run simulations, and the simulation job status.
$print() is quite useful to get the simulation status, especially when `wait` is `FALSE` in $run(). The job status will be updated and printed whenever $print() is called.

Returns: The `EplusSql` object itself, invisibly.

Examples:

```
\dontrun{
job$print()
}
```

Author(s)

Hongyuan Jia

See Also

ParametricJob class for EnergyPlus parametric simulations.

`param_job()` for creating an EnergyPlus parametric job.

Examples

```
## ------------------------------------------------
## Method `EplusJob$new`
## ------------------------------------------------

## Not run:
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    job <- eplus_job(idf_path, epw_path)

    # create from an Idf and an Epw object
    job <- eplus_job(read_idf(idf_path), read_epw(epw_path))
}
```

```
## End(Not run)

## ------------------------------------------------
## Method `EplusJob$version`
## ------------------------------------------------

## Not run:
job$version()

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$path`
## ------------------------------------------------

## Not run:
job$path()
job$path("idf")
job$path("epw")

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$run`
## ------------------------------------------------

## Not run:
# only run design day
job$run(NULL)

# specify output directory
job$run(dir = tempdir())

# run in the background
job$run(wait = TRUE)
# see job status
job$status()

# force to kill background job before running the new one
job$run(force = TRUE)

# do not show anything in the console
job$run(echo = FALSE)

# copy external files used in the model to simulation output directory
job$run(copy_external = TRUE)

## End(Not run)


## ------------------------------------------------
```

```
## Method `EplusJob$kill`
## ------------------------------------------------

## Not run:
job$kill()

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$status`
## ------------------------------------------------

## Not run:
job$status()

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$errors`
## ------------------------------------------------

## Not run:
job$errors()

# show all information
job$errors(info = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$output_dir`
## ------------------------------------------------

## Not run:
job$output_dir()

# Below will open output directory
# job$output_dir(open = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$locate_output`
## ------------------------------------------------

## Not run:
# get the file path of the error file
job$locate_output(".err", strict = FALSE)
```

```
# can use to detect if certain output file exists
job$locate_output(".expidf", strict = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$read_rdd`
## ------------------------------------------------

## Not run:
job$read_rdd()

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$read_mdd`
## ------------------------------------------------

## Not run:
job$read_mdd()

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$list_table`
## ------------------------------------------------

## Not run:
job$list_table()

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$read_table`
## ------------------------------------------------

## Not run:
# read a specific table
job$read_table("Zones")

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$report_data_dict`
## ------------------------------------------------

## Not run:
job$report_data_dict()
```

```
## End(Not run)


## ------------------------------------------------
## Method `EplusJob$report_data`
## ------------------------------------------------

## Not run:
# read all report data
job$report_data()

# specify output variables using report data dictionary
dict <- job$report_data_dict()
job$report_data(dict[units == "C"])

# specify output variables using 'key_value' and 'name'
job$report_data("environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
job$report_data(dict[1], year = 2020, tz = "Etc/GMT+8")

# explicitly specify case name
job$report_data(dict[1], case = "example")

# get all possible columns
job$report_data(dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
job$report_data(dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
job$report_data(dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
job$report_data(dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)

# only get specified run period data
job$read_table("EnvironmentPeriods") # possible environment name
job$report_data(dict[1], environment_name = "San Francisco Intl Ap CA USA TMY3 WMO#=724940")
# can also be done using 'environment_period_index' column
job$report_data(dict[1], all = TRUE)[environment_period_index == 3L]

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$tabular_data`
## ------------------------------------------------

## Not run:
# read all tabular data
```

```
job$tabular_data()

# explicitly specify data you want
str(job$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))

# get tabular data in wide format and coerce numeric values
str(job$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy",
    wide = TRUE, string_value = FALSE
))

## End(Not run)


## ------------------------------------------------
## Method `EplusJob$print`
## ------------------------------------------------

## Not run:
job$print()

## End(Not run)
```

`eplusr_option` *Get and Set eplusr options*

Description

Get and set eplusr options which affect the way in which eplusr computes and displays its results.

Usage

```
eplusr_option(...)
```

Arguments

`...`	Any available options to define, using `name = value`. All available options are shown below. If no options are given, all values of current options are returned. If a single option name, its value is returned.

Details

- `validate_level`: The strictness level of validation during field value modification and model error checking. Possible value: `"none"`, `"draft"` and `"final"` or a custom validation level using `custom_validate()`. Default: `"final"`. For what validation components each level contains, see `level_checks()`.
- `view_in_ip`: Whether models should be presented in IP units. Default: `FALSE`. It is not recommended to set this option to `TRUE` as currently IP-units support in eplusr is not fully tested.
- `save_format`: The default format to use when saving Idf objects to `.idf` files. Possible values: `"asis"`, `"sorted"`, `"new_top"` and `"new_bot"`. The later three have the same effect as Save Options settings `"Sorted"`, `"Original with New at Top"` and `"Original with New at Bottom"` in IDF Editor, respectively. For `"asis"`, the saving format will be set according to the header of IDF file. If no header found, `"sorted"` is used. Default: `"asis"`.
- `num_parallel`: Maximum number of parallel simulations to run. Default: `parallel::detectCores()`.
- `verbose_info`: Whether to show information messages. Default: `TRUE`.
- `autocomplete`: Whether to turn on autocompletion on class and field names. Underneath, `makeActiveBinding()` is used to add or move active bindings in Idf and IdfObjects to directly return objects in class or field values. This will make it possible to dynamically show current class and field names in both RStudio and in the terminal. However, this process does have a penalty on the performance. It can make adding or modifying large mounts of Idf and IdfObjects extremely slower. Default: `interactive()`.

Value

If called directly, a named list of input option values. If input is a single option name, a length-one vector whose type is determined by that option. If input is new option values, a named list of newly set option values.

Author(s)

Hongyuan Jia

Examples

```
# list all current options
eplusr_option() # a named list

# get a specific option value
eplusr_option("verbose_info")

# set options
eplusr_option(verbose_info = TRUE, view_in_ip = FALSE)
```

EplusSql *Retrieve Simulation Outputs Using EnergyPlus SQLite Output File*

Description

`EplusSql` class wraps SQL queries that can retrieve simulation outputs using EnergyPlus SQLite output file.

Details

SQLite output is an optional output format for EnergyPlus. It will be created if there is an object in class `Output:SQLite`. If the value of field `Option` in class `Output:SQLite` is set to `"SimpleAndTabular"`, then database tables related to the tabular reports will be also included.

There are more than 30 tables in the SQLite output file which contains all of the data found in EnergyPlus's tabular output files, standard variable and meter output files, plus a number of reports that are found in the eplusout.eio output file. The full description for SQLite outputs can be found in the EnergyPlus *"Output Details and Examples"* documentation. Note that all column names of tables returned have been tidied, i.e. `"KeyValue"` becomes `"key_value"`, `"IsMeter"` becomes `"is_meter"` and etc.

`EplusSql` class makes it possible to directly retrieve simulation results without creating an EplusJob object. EplusJob can only get simulation outputs after the job was successfully run before.

However, it should be noted that, unlike EplusJob, there is no checking on whether the simulation is terminated or completed unsuccessfully or, the parent Idf has been changed since last simulation. This means that you may encounter some problems when retrieve data from an unsuccessful simulation. It is suggested to carefully go through the `.err` file using `read_err()` to make sure the output data in the SQLite is correct and reliable.

Methods

Public methods:

- `EplusSql$new()`
- `EplusSql$path()`
- `EplusSql$path_idf()`
- `EplusSql$list_table()`
- `EplusSql$read_table()`
- `EplusSql$report_data_dict()`
- `EplusSql$report_data()`
- `EplusSql$tabular_data()`
- `EplusSql$print()`

Method `new()`**:** Create an `EplusSql` object

Usage:

```
EplusSql$new(sql)
```

Arguments:

`sql` A path to an local EnergyPlus SQLite output file.

Returns: An `EplusSql` object.

Examples:

```
\dontrun{
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # copy to tempdir and run the model
    idf <- read_idf(idf_path)
    idf$run(epw_path, tempdir(), echo = FALSE)

    # create from local file
    sql <- eplus_sql(file.path(tempdir(), "1ZoneUncontrolled.sql"))
}
}
```

Method `path()`**:** Get the file path of current `EpwSql` object

Usage:

```
EplusSql$path()
```

Details: $path() returns the path of EnergyPlus SQLite file.

Returns: A single string.

Examples:

```
\dontrun{
# get path
sql$path()
}
```

Method `path_idf()`**:** Get the path of corresponding IDF file

Usage:

```
EplusSql$path_idf()
```

Details: $path_idf() returns the IDF file path with same name as the SQLite file in the same folder. `NULL` is returned if no corresponding IDF is found.

Returns: NULL or a single string.

Examples:

```
\dontrun{
# get path
sql$path_idf()
}
```

Method `list_table()`**:** List all table names in current EnergyPlus SQL output

Usage:

```
EplusSql$list_table()
```

Details: $list_table() returns all available table and view names in the EnergyPlus SQLite file.

Returns: A character vector

Examples:

```
\dontrun{
sql$list_table()
}
```

Method `read_table()`**:** Read a single table from current EnergyPlus SQL output

Usage:

```
EplusSql$read_table(name)
```

Arguments:

`name` A single string specifying the name of table to read.

Details: $read_table() takes a valid table `name` of those from `$list_table()` and returns that table data in a `data.table::data.table()` format.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
# read a specific table
sql$read_table("Zones")
}
```

Method `report_data_dict()`**:** Read report data dictionary from current EnergyPlus SQL output

Usage:

```
EplusSql$report_data_dict()
```

Details: $report_data_dict() returns a `data.table::data.table()` which contains all information about report data.
For details on the meaning of each columns, please see "2.20.2.1 ReportDataDictionary Table" in EnergyPlus "Output Details and Examples" documentation.

Returns: A `data.table::data.table()` of 10 columns:

- `report_data_dictionary_index`: The integer used to link the dictionary data to the variable data. Mainly useful when joining different tables
- `is_meter`: Whether report data is a meter data. Possible values: `0` and `1`
- `timestep_type`: Type of data timestep. Possible values: `Zone` and HVAC System
- `key_value`: Key name of the data
- `name`: Actual report data name
- `reporting_frequency`:

- schedule_name: Name of the the schedule that controls reporting frequency.
- units: The data units

Examples:

```
\dontrun{
sql$report_data_dict()
}
```

Method `report_data()`**:** Read report data

Usage:

```
EplusSql$report_data(
  key_value = NULL,
  name = NULL,
  year = NULL,
  tz = "UTC",
  case = "auto",
  all = FALSE,
  wide = FALSE,
  period = NULL,
  month = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  interval = NULL,
  simulation_days = NULL,
  day_type = NULL,
  environment_name = NULL
)
```

Arguments:

key_value A character vector to identify key values of the data. If NULL, all keys of that variable will be returned. key_value can also be data.frame that contains key_value and name columns. In this case, name argument in $report_data() is ignored. All available key_value for current simulation output can be obtained using $report_data_dict(). Default: NULL.

name A character vector to identify names of the data. If NULL, all names of that variable will be returned. If key_value is a data.frame, name is ignored. All available name for current simulation output can be obtained using $report_data_dict(). Default: NULL.

year Year of the date time in column datetime. If NULL, it will calculate a year value that meets the start day of week restriction for each environment. Default: NULL.

tz Time zone of date time in column datetime. Default: "UTC".

case A single string used to add a character column case in the returned results to indicate the case of this simulation. If NULL, no column is added. If "auto", the name of the IDF file without extension is used. Default: "auto".

all If TRUE, extra columns are also included in the returned data.table::data.table().

wide If TRUE, the output is formatted in the same way as standard EnergyPlus csv output file.

`period` A Date or POSIXt vector used to specify which time period to return. The year value does not matter and only month, day, hour and minute value will be used when subsetting. If `NULL`, all time period of data is returned. Default: `NULL`.

`month, day, hour, minute` Each is an integer vector for month, day, hour, minute subsetting of `datetime` column when querying on the SQL database. If `NULL`, no subsetting is performed on those components. All possible `month`, `day`, `hour` and `minute` can be obtained using `$read_table("Time")`. Default: `NULL`.

`interval` An integer vector used to specify which interval length of report to extract. If `NULL`, all interval will be used. Default: `NULL`.

`simulation_days` An integer vector to specify which simulation day data to extract. Note that this number resets after warmup and at the beginning of an environment period. All possible `simulation_days` can be obtained using `$read_table("Time")`. If `NULL`, all simulation days will be used. Default: `NULL`.

`day_type` A character vector to specify which day type of data to extract. All possible day types are: `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Holiday`, `SummerDesignDay`, `WinterDesignDay`, `CustomDay1`, and `CustomDay2`. All possible values for current simulation output can be obtained using `$read_table("Time")`. A few grouped options are also provided:

- `"Weekday"`: All working days, i.e. from Monday to Friday
- `"Weekend"`: Saturday and Sunday
- `"DesignDay"`: Equivalent to `"SummerDesignDay"` plus `"WinterDesignDay"`
- `"CustomDay"`: CustomDay1 and CustomDay2
- `"SpecialDay"`: Equivalent to `"DesignDay"` plus `"CustomDay"`
- `"NormalDay"`: Equivalent to `"Weekday"` and `"Weekend"` plus `"Holiday"`

`environment_name` A character vector to specify which environment data to extract. If `NULL`, all environment data are returned. Default: `NULL`. All possible `environment_name` for current simulation output can be obtained using:

```
$read_table("EnvironmentPeriods")
```

Details: $report_data() extracts the report data in a `data.table::data.table()` using key values, variable names and other specifications.
$report_data() can also directly take all or subset output from $report_data_dict() as input, and extract all data specified.
The returned column numbers varies depending on `all` argument.

- `all` is `FALSE`, the returned `data.table::data.table()` has 6 columns:
  - `case`: Simulation case specified using `case` argument
  - `datetime`: The date time of simulation result
  - `key_value`: Key name of the data
  - `name`: Actual report data name
  - `units`: The data units
  - `value`: The data value
- `all` is `TRUE`, besides columns described above, extra columns are also included:
  - `month`: The month of reported date time
  - `day`: The day of month of reported date time
  - `hour`: The hour of reported date time
  - `minute`: The minute of reported date time

- dst: Daylight saving time indicator. Possible values: 0 and 1
- interval: Length of reporting interval
- simulation_days: Day of simulation
- day_type: The type of day, e.g. Monday, Tuesday and etc.
- environment_period_index: The indices of environment.
- environment_name: A text string identifying the environment.
- is_meter: Whether report data is a meter data. Possible values: 0 and 1
- type: Nature of data type with respect to state. Possible values: Sum and Avg
- index_group: The report group, e.g. Zone, System
- timestep_type: Type of data timestep. Possible values: Zone and HVAC System
- reporting_frequency: The reporting frequency of the variable, e.g. HVAC System Timestep, Zone Timestep.
- schedule_name: Name of the the schedule that controls reporting frequency.

With the datetime column, it is quite straightforward to apply time-series analysis on the simulation output. However, another painful thing is that every simulation run period has its own Day of Week for Start Day. Randomly setting the year may result in a date time series that does not have the same start day of week as specified in the RunPeriod objects.
eplusr provides a simple solution for this. By setting year to NULL, which is the default behavior, eplusr will calculate a year value (from current year backwards) for each run period that compliances with the start day of week restriction.
It is worth noting that EnergyPlus uses 24-hour clock system where 24 is only used to denote midnight at the end of a calendar day. In EnergyPlus output, "00:24:00" with a time interval being 15 mins represents a time period from "00:23:45" to "00:24:00", and similarly "00:15:00" represents a time period from "00:24:00" to "00:15:00" of the next day. This means that if current day is Friday, day of week rule applied in schedule time period "00:23:45" to "00:24:00" (presented as "00:24:00" in the output) is also Friday, but not Saturday. However, if you try to get the day of week of time "00:24:00" in R, you will get Saturday, but not Friday. This introduces inconsistency and may cause problems when doing data analysis considering day of week value.
With wide equals TRUE, $report_data() will format the simulation output in the same way as standard EnergyPlus csv output file. Sometimes this can be useful as there may be existing tools/workflows that depend on this format. When both wide and all are TRUE, columns of runperiod environment names and date time components are also returned, including: environment_period_index", "environment_name, simulation_days, datetime, month, day, hour, minute, day_type.
For convenience, input character arguments matching in $report_data() are **case-insensitive**.

Returns: A data.table::data.table().

Examples:

```
\dontrun{
# read all report data
sql$report_data()

# specify output variables using report data dictionary
dict <- sql$report_data_dict()
sql$report_data(dict[units == "C"])
```

```
# specify output variables using 'key_value' and 'name'
sql$report_data("environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
sql$report_data(dict[1], year = 2020, tz = "Etc/GMT+8")

# explicitly specify case name
sql$report_data(dict[1], case = "example")

# get all possible columns
sql$report_data(dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
sql$report_data(dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
sql$report_data(dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
sql$report_data(dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)

# only get specified run period data
sql$read_table("EnvironmentPeriods") # possible environment name
sql$report_data(dict[1], environment_name = "San Francisco Intl Ap CA USA TMY3 WMO#=724940")
# can also be done using 'environment_period_index' column
sql$report_data(dict[1], all = TRUE)[environment_period_index == 3L]
}
```

Method `tabular_data()`**:** Read tabular data

Usage:

```
EplusSql$tabular_data(
  report_name = NULL,
  report_for = NULL,
  table_name = NULL,
  column_name = NULL,
  row_name = NULL,
  case = "auto",
  wide = FALSE,
  string_value = !wide
)
```

Arguments:

`report_name`, `report_for`, `table_name`, `column_name`, `row_name` Each is a character vector for subsetting when querying the SQL database. For the meaning of each argument, please see the description above.

`case` A single string used to add a character column `case` in the returned results to indicate the case of this simulation. If `NULL`, no column is added. If `"auto"`, the name of the IDF file

without extension is used. Default: "auto".

wide If TRUE, each table will be converted into the similar format as it is shown in EnergyPlus HTML output file. Default: FALSE.

string_value Only applicable when wide is TRUE. If string_value is FALSE, instead of keeping all values as characters, values in possible numeric columns are converted into numbers. Default: the opposite of wide. Possible numeric columns indicate column that:

- columns that have associated units
- columns that contents numbers

Details: $tabular_data() extracts the tabular data in a data.table::data.table() using report, table, column and row name specifications. The returned data.table::data.table() has 9 columns:

- case: Simulation case specified using case argument
- index: Tabular data index
- report_name: The name of the report that the record belongs to
- report_for: The For text that is associated with the record
- table_name: The name of the table that the record belongs to
- column_name: The name of the column that the record belongs to
- row_name: The name of the row that the record belongs to
- units: The units of the record
- value: The value of the record **in string format** by default.

For convenience, input character arguments matching in $tabular_data() are **case-insensitive**.

Returns: A data.table::data.table() with 9 columns (when wide is FALSE) or a named list of data.table::data.table()s where the names are the combination of report_name, report_for and table_name.

Examples:

```
\dontrun{
# read all tabular data
sql$tabular_data()

# explicitly specify data you want
str(sql$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))

# get tabular data in wide format and coerce numeric values
str(sql$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy",
    wide = TRUE, string_value = FALSE
))
```

```
}
```

Method `print()`**:** Print `EplusSql` object

Usage:

```
EplusSql$print()
```

Details: $print() shows the core information of this `EplusSql` object, including the path of the EnergyPlus SQLite file, last modified time of the SQLite file and the path of the IDF file with the same name in the same folder.

Returns: The `EplusSql` object itself, invisibly.

Examples:

```
\dontrun{
sql$print()
}
```

Author(s)

Hongyuan Jia

Examples

```
## ------------------------------------------------
## Method `EplusSql$new`
## ------------------------------------------------

## Not run:
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # copy to tempdir and run the model
    idf <- read_idf(idf_path)
    idf$run(epw_path, tempdir(), echo = FALSE)

    # create from local file
    sql <- eplus_sql(file.path(tempdir(), "1ZoneUncontrolled.sql"))
}

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$path`
## ------------------------------------------------
```

```
## Not run:
# get path
sql$path()

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$path_idf`
## ------------------------------------------------

## Not run:
# get path
sql$path_idf()

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$list_table`
## ------------------------------------------------

## Not run:
sql$list_table()

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$read_table`
## ------------------------------------------------

## Not run:
# read a specific table
sql$read_table("Zones")

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$report_data_dict`
## ------------------------------------------------

## Not run:
sql$report_data_dict()

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$report_data`
## ------------------------------------------------
```

```
## Not run:
# read all report data
sql$report_data()

# specify output variables using report data dictionary
dict <- sql$report_data_dict()
sql$report_data(dict[units == "C"])

# specify output variables using 'key_value' and 'name'
sql$report_data("environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
sql$report_data(dict[1], year = 2020, tz = "Etc/GMT+8")

# explicitly specify case name
sql$report_data(dict[1], case = "example")

# get all possible columns
sql$report_data(dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
sql$report_data(dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
sql$report_data(dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
sql$report_data(dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)

# only get specified run period data
sql$read_table("EnvironmentPeriods") # possible environment name
sql$report_data(dict[1], environment_name = "San Francisco Intl Ap CA USA TMY3 WMO#=724940")
# can also be done using 'environment_period_index' column
sql$report_data(dict[1], all = TRUE)[environment_period_index == 3L]

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$tabular_data`
## ------------------------------------------------

## Not run:
# read all tabular data
sql$tabular_data()

# explicitly specify data you want
str(sql$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
```

```
))

# get tabular data in wide format and coerce numeric values
str(sql$tabular_data(
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy",
    wide = TRUE, string_value = FALSE
))

## End(Not run)


## ------------------------------------------------
## Method `EplusSql$print`
## ------------------------------------------------

## Not run:
sql$print()

## End(Not run)
```

`eplus_sql` *Read an Energy SQLite Output File*

Description

`eplus_sql()` takes an EnergyPlus SQLite output file as input, and returns an `EplusSQL` object for collecting simulation outputs. For more details, please see EplusSql.

Usage

```
eplus_sql(sql)
```

Arguments

`sql`	A path to an local EnergyPlus SQLite output file.

Value

An EplusSql object.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # copy to tempdir and run the model
    idf <- read_idf(idf_path)
    idf$run(epw_path, tempdir(), echo = FALSE)

    # create from local file
    sql <- eplus_sql(file.path(tempdir(), "1ZoneUncontrolled.sql"))
}

## End(Not run)
```

Epw *Read, and modify an EnergyPlus Weather File (EPW)*

Description

Reading an EPW file starts with function `read_epw()`, which parses an EPW file and returns an Epw object. The parsing process is basically the same as [EnergyPlus/WeatherManager.cc] in EnergyPlus, with some simplifications.

Details

An EPW file can be divided into two parts, headers and weather data. The first eight lines of a standard EPW file are normally headers which contains data of location, design conditions, typical/extreme periods, ground temperatures, holidays/daylight savings, data periods and other comments. Epw class provides methods to directly extract those data. For details on the data structure of EPW file, please see "Chapter 2 - Weather Converter Program" in EnergyPlus "Auxiliary Programs" documentation. An online version can be found here.

There are about 35 variables in the core weather data. However, not all of them are used by EnergyPlus. Actually, despite of date and time columns, only 13 columns are used:

1. dry bulb temperature
2. dew point temperature
3. relative humidity
4. atmospheric pressure
5. horizontal infrared radiation intensity from sky
6. direct normal radiation
7. diffuse horizontal radiation

8. wind direction
9. wind speed
10. present weather observation
11. present weather codes
12. snow depth
13. liquid precipitation depth

Note the `hour` column in the core weather data corresponds to the period from **(Hour-1)th** to **(Hour)th**. For instance, if the number of interval per hour is 1, hour of 1 on a certain day corresponds to the period between 00:00:01 to 01:00:00, Hour of 2 corresponds to the period between 01:00:01 to 02:00:00, and etc. Currently, in EnergyPlus the minute column is **not used** to determine currently sub-hour time. For instance, if the number of interval per hour is 2, there is no difference between two rows with following time columns (a) Hour 1, Minute 0; Hour 1, Minute 30 and (b) Hour 1, Minute 30; Hour 1, Minute 60. Only the number of rows count. When EnergyPlus reads the EPW file, both (a) and (b) represent the same time period: 00:00:00 - 00:30:00 and 00:30:00 - 01:00:00. Missing data on the weather file used can be summarized in the eplusout.err file, if `DisplayWeatherMissingDataWarnings` is turned on in `Output:Diagnostics` object. In EnergyPlus, missing data is shown only for fields that EnergyPlus will use. EnergyPlus will fill some missing data automatically during simulation. Likewise out of range values are counted for each occurrence and summarized. However, note that the out of range values will **not be changed** by EnergyPlus and could affect your simulation.

`Epw` class provides methods to easily extract and inspect those abnormal (missing and out of range) weather data and also to know what kind of actions that EnergyPlus will perform on those data.

EnergyPlus energy model calibration often uses actual measured weather data. In order to streamline the error-prone process of creating custom EPW file, `Epw` provides methods to direction add, replace the core weather data.

Methods

Public methods:

- `Epw$new()`
- `Epw$path()`
- `Epw$definition()`
- `Epw$location()`
- `Epw$design_condition()`
- `Epw$typical_extreme_period()`
- `Epw$ground_temperature()`
- `Epw$holiday()`
- `Epw$comment1()`
- `Epw$comment2()`
- `Epw$num_period()`
- `Epw$interval()`
- `Epw$period()`
- `Epw$missing_code()`

- Epw$initial_missing_value()
- Epw$range_exist()
- Epw$range_valid()
- Epw$fill_action()
- Epw$data()
- Epw$abnormal_data()
- Epw$redundant_data()
- Epw$make_na()
- Epw$fill_abnormal()
- Epw$add_unit()
- Epw$drop_unit()
- Epw$purge()
- Epw$add()
- Epw$set()
- Epw$del()
- Epw$is_unsaved()
- Epw$save()
- Epw$print()
- Epw$clone()

Method `new()`**:** Create an Epw object

Usage:

```
Epw$new(path)
```

Arguments:

`path` Either a path, a connection, or literal data (either a single string or a raw vector) to an EnergyPlus Weather File (EPW). If a file path, that file usually has a extension `.epw`.

Details: It takes an EnergyPlus Weather File (EPW) as input and returns an Epw object.

Returns: An Epw object.

Examples:

```
\dontrun{
# read an EPW file from EnergyPlus website
path_base <- "https://energyplus.net/weather-download"
path_region <- "north_and_central_america_wmo_region_4/USA/CA"
path_file <- "USA_CA_San.Francisco.Intl.AP.724940_TMY3/USA_CA_San.Francisco.Intl.AP.724940_TMY3.e
path_epw <- file.path(path_base, path_region, path_file)
epw <- read_epw(path_epw)

# read an EPW file distributed with EnergyPlus
if (is_avail_eplus(8.8)) {
    path_epw <- file.path(
        eplus_config(8.8)$dir,
        "WeatherData",
        "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
```

```
    )
    epw <- read_epw(path_epw)
}
}
```

Method `path()`**:** Get the file path of current `Epw`

Usage:

```
Epw$path()
```

Details: $path() returns the full path of current `Epw` or `NULL` if the `Epw` object is created using a character vector and not saved locally.

Returns: `NULL` or a single string.

Examples:

```
\dontrun{
# get path
epw$path()
}
```

Method `definition()`**:** Get the IddObject object for specified EPW class.

Usage:

```
Epw$definition(class)
```

Arguments:

`class` A single string.

Details: $definition() returns an IddObject of given EPW class. IddObject contains all data used for parsing that EPW class.
Currently, all supported EPW classes are:

- `LOCATION`
- DESIGN CONDITIONS
- TYPICAL/EXTREME PERIODS
- GROUND TEMPERATURES
- HOLIDAYS/DAYLIGHT SAVINGS
- COMMENTS 1
- COMMENTS 2
- DATA PERIODS
- WEATHER DATA

Examples:

```
\dontrun{
# get path
epw$definition("LOCATION")
}
```

Method `location()`**:** Get and modify LOCATION header

Usage:

```
Epw$location(
  city,
  state_province,
  country,
  data_source,
  wmo_number,
  latitude,
  longitude,
  time_zone,
  elevation
)
```

Arguments:

`city` A string of city name recorded in the `LOCATION` header.

`state_province` A string of state or province name recorded in the `LOCATION` header.

`country` A string of country name recorded in the `LOCATION` header.

`data_source` A string of data source recorded in the `LOCATION` header.

`wmo_number` A string of WMO (World Meteorological Organization) number recorded in the `LOCATION` header.

`latitude` A number of latitude recorded in the `LOCATION` header. North latitude is positive and south latitude is negative. Should in range [-90, +90].

`longitude` A number of longitude recorded in the `LOCATION` header. East longitude is positive and west longitude is negative. Should in range [-180, +180].

`time_zone` A number of time zone recorded in the `LOCATION` header. Usually presented as the offset hours from UTC time. Should in range [-12, +14].

`elevation` A number of elevation recorded in the `LOCATION` header. Should in range [-1000, 9999.9).

Details: $location() takes new values for `LOCATION` header fields and returns the parsed values of `LOCATION` header in a list format. If no input is given, current `LOCATION` header value is returned.

Returns: A named list of 9 elements.

Examples:

```
\dontrun{
epw$location()

# modify location data
epw$location(city = "MyCity")
}
```

Method `design_condition()`**:** Get DESIGN CONDITION header

Usage:

```
Epw$design_condition()
```

Details: $design_condition() returns the parsed values of DESIGN CONDITION header in a list format with 4 elements:

- source: A string of source field
- heating: A list, usually of length 16, of the heading design conditions
- cooling: A list, usually of length 32, of the cooling design conditions
- extremes: A list, usually of length 16, of the extreme design conditions

For the meaning of each element, please see ASHRAE Handbook of Fundamentals.

Returns: A named list of 4 elements.

Examples:

```
\dontrun{
epw$design_condition()
}
```

Method `typical_extreme_period()`**:** Get TYPICAL/EXTREME header

Usage:

```
Epw$typical_extreme_period()
```

Details: $typical_extreme_period() returns the parsed values of TYPICAL/EXTREME PERIOD header in a data.table format with 6 columns:

- index: Integer type. The index of typical or extreme period record
- name: Character type. The name of typical or extreme period record
- type: Character type. The type of period. Possible value: typical and extreme
- start_day: Date type with customized formatting. The start day of the period
- start_day: Date type with customized formatting. The end day of the period

Returns: A data.table::data.table() with 6 columns.

Examples:

```
\dontrun{
epw$typical_extreme_period()
}
```

Method `ground_temperature()`**:** Get GROUND TEMPERATURE header

Usage:

```
Epw$ground_temperature()
```

Details: $ground_temperature() returns the parsed values of GROUND TEMPERATURE header in a data.table format with 17 columns:

- index: Integer type. The index of ground temperature record
- depth: Numeric type. The depth of the ground temperature is measured
- soil_conductivity: Numeric type. The soil conductivity at measured depth
- soil_density: Numeric type. The soil density at measured depth
- soil_specific heat: Numeric type. The soil specific heat at measured depth
- January to December: Numeric type. The measured group temperature for each month.

Returns: A data.table::data.table() with 17 columns.

Examples:

```
\dontrun{
epw$ground_temperature()
}
```

Method `holiday()`**:** Get and modify HOLIDAYS/DAYLIGHT SAVINGS header

Usage:

```
Epw$holiday(leapyear, dst, holiday)
```

Arguments:

`leapyear` Either `TRUE` or `FALSE`.

`dst` A length 2 EPW date specifications identifying the start and end of daylight saving time. For example, `c(3.10,10.3)`.

`holiday` a list or a data.frame containing two elements (columns) `name` and `day` where `name` are the holiday names and `day` are valid EPW date specifications. For example:

```
list(name = c("New Year's Day", "Christmas Day"), day = c("1.1", "25 Dec"))
```

Details: $holiday() takes new value for leap year indicator, daylight saving time and holiday specifications, set these new values and returns the parsed values of HOLIDAYS/DAYLIGHT SAVINGS header. If no input is given, current values of HOLIDAYS/DAYLIGHT SAVINGS header is returned. It returns a list of 3 elements:

- `leapyear`: A single logical vector. `TRUE` means that the weather data contains leap year data
- `dst`: A Date vector contains the start and end day of daylight saving time
- `holiday`: A data.table contains 2 columns. If no holiday specified, an empty data.table
  - `name`: Name of the holiday
  - `day`: Date of the holiday

Validation process below is performed when changing the `leapyear` indicator:

- If current record of `leapyear` is `TRUE`, but new input is `FALSE`, the modification is only conducted when all data periods do not cover Feb 29.
- If current record of `leapyear` is `FALSE`, but new input is `TRUE`, the modification is only conducted when TMY data periods do not across Feb, e.g. [01/02, 02/28], [03/01, 12/31]; for AMY data, it is always OK.

The date specifications in `dst` and `holiday` should follow the rules of **"Table 2.14: Weather File Date File Interpretation"** in "AuxiliaryPrograms" documentation. eplusr is able to handle all those kinds of formats automatically. Basically, 5 formats are allowed:

1. A single integer is interpreted as the Julian day of year. For example, `1`, `2`, `3` and `4` will be parsed and presented as 1st day, 2nd day, 3rd day and 4th day.
2. A single number is interpreted as `Month.Day`. For example, `1.2` and `5.6` will be parsed and presented as Jan 02 and May 06.
3. A string giving `MonthName` / `DayNumber`, `DayNumber` / `MonthName`, and `MonthNumber` / `DayNumber`. A year number can be also included. For example, `"Jan/1"`, `"05/Dec"`, `"7/8"`, `"02/10/2019"`, and `"2019/04/05"` will be parsed and presented as Jan 02, Dec 06, Jul 8, `2019-02-10` and `2019-04-15`.
4. A string giving number Weekday in Month. For example, `"2 Sunday in Jan"` will be parsed and presented as 2th Sunday in January.

5. A string giving Last Weekday in Month. For example, `"last Sunday in Dec"` will be parsed and presented as Last Sunday in December.

For convenience, besides all the formats described above, `dst` and days in `holiday` also accept standard Dates input. They will be treated as the same way as No.3 format described above.

Returns: A named list of 3 elements.

Examples:

```
\dontrun{
epw$holiday()

# add daylight saving time
epw$holiday(dst = c(3.10, 11.3))
}
```

Method `comment1()`**:** Get and modify COMMENT1 header

Usage:

```
Epw$comment1(comment)
```

Arguments:

`comment` A string of new comments.

Details: $comment1() takes a single string of new comments and replaces the old comment with input one. If `NULL` is given, the comment is removed. Empty string or a string that contains only spaces will be treated as `NULL`. If no input is given, current comment is returned. If no comments exist, `NULL` is returned.

Returns: A single string.

Examples:

```
\dontrun{
epw$comment1()

epw$comment1("Comment1")
}
```

Method `comment2()`**:** Get and modify COMMENT2 header

Usage:

```
Epw$comment2(comment)
```

Arguments:

`comment` A string of new comments.

Details: $comment2() takes a single string of new comments and replaces the old comment with input one. If `NULL` is given, the comment is removed. Empty string or a string that contains only spaces will be treated as `NULL`. If no input is given, current comment is returned. If no comments exist, `NULL` is returned.

Returns: A single string.

Examples:

```
\dontrun{
epw$comment2()

epw$comment2("Comment2")
}
```

Method `num_period()`**:** Get number of data periods in DATA PERIODS header

Usage:

```
Epw$num_period()
```

Details: $num_period() returns a single positive integer of how many data periods current `Epw` contains.

Returns: A single integer.

Examples:

```
\dontrun{
epw$num_period()
}
```

Method `interval()`**:** Get the time interval in DATA PERIODS header

Usage:

```
Epw$interval()
```

Details: $interval() returns a single positive integer of how many records of weather data exist in one hour.

Returns: A single integer.

Examples:

```
\dontrun{
epw$interval()
}
```

Method `period()`**:** Get and modify data period meta data in DATA PERIODS header

Usage:

```
Epw$period(period, name, start_day_of_week)
```

Arguments:

`period` A positive integer vector identifying the data period indexes.

`name` A character vector used as new names for specified data periods. Should have the same length as `index`.

`start_day_of_week` A character vector or an integer vector used as the new start days of week of specified data periods. Should have the same length as `index`.

Details: $period() takes a data period index, a new period name and start day of week specification, and uses that input to replace the data period's name and start day of week. If no input is given, data periods in current `Epw` is returned.

Returns: A data.table with 5 columns:

- `index`: Integer type. The index of data period.
- `name`: Character type. The name of data period.
- `start_day_of_week`: Character type. The start day of week of data period.
- `start_day`: Date (EpwDate) type. The start day of data period.
- `end_day`: Date (EpwDate) type. The end day of data period.

Examples:

```
\dontrun{
# modify data period name
epw$period(1, name = "test")

# change start day of week
epw$period(1, start_day_of_week = 3)
}
```

Method `missing_code()`**:** Get missing code for weather data variables

Usage:

```
Epw$missing_code()
```

Details: $missing_code() returns a list of 29 elements containing the value used as missing value identifier for all weather data.

Returns: A named list of 29 elements.

Examples:

```
\dontrun{
epw$missing_code()
}
```

Method `initial_missing_value()`**:** Get initial value for missing data of weather data variables

Usage:

```
Epw$initial_missing_value()
```

Details: $initial_missing_value() returns a list of 16 elements containing the initial value used to replace missing values for corresponding weather data.

Returns: A named list of 16 elements.

Examples:

```
\dontrun{
epw$initial_missing_value()
}
```

Method `range_exist()`**:** Get value ranges for existing values of weather data variables

Usage:

```
Epw$range_exist()
```

Details: $range_exist() returns a list of 28 elements containing the range each numeric weather data should fall in. Any values out of this range are treated as missing.

Returns: A named list of 28 elements.

Examples:

```
\dontrun{
epw$range_exist()
}
```

Method `range_valid()`**:** Get value ranges for valid values of weather data variables

Usage:

```
Epw$range_valid()
```

Details: $range_valid() returns a list of 28 elements containing the range each numeric weather data should fall in. Any values out of this range are treated as invalid.

Returns: A named list of 28 elements.

Examples:

```
\dontrun{
epw$range_valid()
}
```

Method `fill_action()`**:** Get fill actions for abnormal values of weather data variables

Usage:

```
Epw$fill_action(type = c("missing", "out_of_range"))
```

Arguments:

`type` What abnormal type of actions to return. Should be one of `"missing"` and `"out_of_range"`. Default: `"missing"`.

Details: $fill_action() returns a list containing `actions` that EnergyPlus will perform when certain abnormal data found for corresponding weather data. There are 3 types of actions in total:

- `do_nothing`: All abnormal values are left as they are.
- `use_zero`: All abnormal values are reset to zeros.
- `use_previous`: The first abnormal values of variables will be set to the initial missing values. All after are set to previous valid one.

Returns: A named list.

Examples:

```
\dontrun{
epw$fill_action("missing")

epw$fill_action("out_of_range")
}
```

Method `data()`**:** Get weather data

Usage:

```
Epw$data(
  period = 1L,
  start_year = NULL,
  align_wday = TRUE,
  tz = "UTC",
  update = FALSE,
  line = FALSE
)
```

Arguments:

`period` A single positive integer identifying the data period index. Data periods information can be obtained using `$period()` described above.

`start_year` A positive integer identifying the year of first date time in specified data period. If `NULL`, the values in the `year` column are used as years of `datetime` column. Default: `NULL`.

`align_wday` Only applicable when `start_year` is `NULL`. If `TRUE`, a year value is automatically calculated for specified data period that compliance with the start day of week value specified in DATA PERIODS header.

`tz` A valid time zone to be assigned to the `datetime` column. All valid time zone names can be obtained using `OlsonNames()`. Default:`"UTC"`.

`update` If `TRUE`, the `year` column are updated according to the newly created `datetime` column using `start_year`. If `FALSE`, original year data in the `Epw` object is kept. Default: `FALSE`.

`line` If `TRUE`, a column named `line` is prepended indicating the line numbers where data occur in the actual EPW file. Default: `FALSE`.

Details: $data() returns weather data of specific data period.
Usually, EPW file downloaded from EnergyPlus website contains TMY weather data. As years of weather data is not consecutive, it may be more convenient to align the year values to be consecutive, which will makes it possible to direct analyze and plot weather data. The `start_year` argument in $data() method can help to achieve this. However, randomly setting the `year` may result in a date time series that does not have the same start day of week as specified in the DATA PERIODS header. eplusr provides a simple solution for this. By setting `year` to `NULL` and `align_wday` to `TRUE`, eplusr will calculate a year value (from current year backwards) for each data period that compliance with the start day of week restriction.
Note that if current data period contains AMY data and `start_year` is given, a warning is given because the actual year values will be overwritten by input `start_year`. An error is given if:

- Using input `start_year` introduces invalid date time. This may happen when weather data contains leap year but input `start_year` is not a leap year.
- Applying specified time zone specified using `tz` introduces invalid date time.

Returns: A `data.table::data.table()` of 36 columns.

Examples:

```
\dontrun{
# get weather data
str(epw$data())

# get weather data but change the year to 2018
# the year column is not changed by default, only the returned datetime column
```

```
head(epw$data(start_year = 2018)$datetime)
str(epw$data(start_year = 2018)$year)
# you can update the year column too
head(epw$data(start_year = 2018, update = TRUE)$year)

# change the time zone of datetime column in the returned weather data
attributes(epw$data()$datetime)
attributes(epw$data(tz = "Etc/GMT+8")$datetime)
}
```

Method `abnormal_data()`**:** Get abnormal weather data

Usage:

```
Epw$abnormal_data(
  period = 1L,
  cols = NULL,
  keep_all = TRUE,
  type = c("both", "missing", "out_of_range")
)
```

Arguments:

`period` A single positive integer identifying the data period index. Data periods information can be obtained using `$period()` described above.

`cols` A character vector identifying what data columns, i.e. all columns except `datetime`, `year`, `month`, `day`, `hour` `minute`, and character columns, to search abnormal values. If `NULL`, all data columns are used. Default: `NULL`.

`keep_all` If `TRUE`, all columns are returned. If `FALSE`, only `line`, `datetime`, `year`, `month`, `day`, `hour` and `minute`, together with columns specified in `cols` are returned. Default: `TRUE`

`type` What abnormal type of data to return. Should be one of `"all"`, `"missing"` and `"out_of_range"`. Default: `"all"`.

Details: $abnormal_data() returns abnormal data of specific data period. Basically, there are 2 types of abnormal data in `Epw`, i.e. missing values and out-of-range values. Sometimes, it may be useful to extract and inspect those data especially when inserting measured weather data. $abnormal_data() does this.
In the returned `data.table::data.table()`, a column named `line` is created indicating the line numbers where abnormal data occur in the actual EPW file.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
epw$abnormal_data()

# only check if there are any abnormal values in air temperature and
# liquid precipitation rate
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"))

# save as above, but only return date time columns plus those 2 columns
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"),
```

```
    keep_all = FALSE
)

# same as above, but only check for missing values
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"),
    type = "missing"
)

# same as above, but only check for out-of-range values
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"),
    type = "out_of_range"
)
}
```

Method `redundant_data()`**:** Get redundant weather data

Usage:

```
Epw$redundant_data()
```

Details: $redundant_data() returns weather data in `Epw` object that do not belong to any data period. This data can be further removed using `$purge()`‘ method described below.
In the returned `data.table::data.table()`, a column named `line` is created indicating the line numbers where redundant data occur in the actual EPW file.

Returns: A `data.table::data.table()` of 37 columns.

Examples:

```
\dontrun{
epw$redundant_data()
}
```

Method `make_na()`**:** Convert abnormal data into NAs

Usage:

```
Epw$make_na(missing = FALSE, out_of_range = FALSE)
```

Arguments:

`missing` If `TRUE`, missing values are included. Default: `FALSE`.

`out_of_range` If `TRUE`, out-of-range values are included. Default: `FALSE`.

Details: $make_na() converts specified abnormal data into `NA`s in specified data period. This makes it easier to find abnormal data directly using `is.na()` instead of using `$missing_code()` $make_na() and `$fill_abnormal()` are reversible, i.e. $make_na() can be used to counteract the effects introduced by `$make_na()`, and vise a versa.
Note that $make_na modify the weather data in-place, meaning that the returned data from `$data()` and `$abnormal_data()` may be different after calling $make_na().

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
# turn all missing values into NAs
summary(epw$data()$liquid_precip_rate)
epw$make_na(missing = TRUE)
summary(epw$data()$liquid_precip_rate)
}
```

Method `fill_abnormal()`**:** Fill abnormal data using prescribed pattern

Usage:

```
Epw$fill_abnormal(missing = FALSE, out_of_range = FALSE, special = FALSE)
```

Arguments:

`missing` If `TRUE`, missing values are included. Default: `FALSE`.

`out_of_range` If `TRUE`, out-of-range values are included. Default: `FALSE`.

`special` If `TRUE`, abnormal data are filled using corresponding actions listed `$fill_action()`. If `FALSE`, all abnormal data are fill with missing code described in `$missing_code()`.

Details: $fill_abnormal() fills specified abnormal data using corresponding actions listed in `$fill_action()`. For what kinds of actions to be performed, please see `$fill_action()`. method described above. Note that only if `special` is `TRUE`, special actions listed in $fill_action() is performed. If `special` is `FALSE`, all abnormal data, including both missing values and out-of-range values, are filled with corresponding missing codes.
`$make_na()` and $fill_abnormal() are reversible, i.e. `$make_na()` can be used to counteract the effects introduced by $fill_abnormal(), and vise a versa.
Note that $fill_abnormal modify the weather data in-place, meaning that the returned data from `$data()` and `$abnormal_data()` may be different after calling $fill_abnormal().

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
# turn all missing values into NAs
summary(epw$data()$liquid_precip_rate)
epw$fill_abnormal(missing = TRUE)
summary(epw$data()$liquid_precip_rate)
}
```

Method `add_unit()`**:** Add units to weather data variables

Usage:

```
Epw$add_unit()
```

Details: $add_unit() assigns units to numeric weather data using `units::set_units()` if applicable.
$add_unit() and `$drop_unit()` are reversible, i.e. $add_unit() can be used to counteract the effects introduced by `$drop_unit()`, and vise a versa.
Note that $add_unit modify the weather data in-place, meaning that the returned data from `$data()` and `$abnormal_data()` may be different after calling $add_unit().

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
# get weather data with units
epw$add_unit()
head(epw$data())

# with units specified, you can easily perform unit conversion using units
# package
t_dry_bulb <- epw$data()$dry_bulb_temperature
units(t_dry_bulb) <- with(units::ud_units, "kelvin")

head(t_dry_bulb)
}
```

Method `drop_unit()`**:** Remove units in weather data variables

Usage:

```
Epw$drop_unit()
```

Details: $drop_unit() removes all units of numeric weather data.
`$add_unit()` and $drop_unit() are reversible, i.e. `$add_unit()` can be used to counteract the effects introduced by $drop_unit(), and vise a versa.
Note that $add_unit modify the weather data in-place, meaning that the returned data from `$data()` and `$abnormal_data()` may be different after calling $add_unit().

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
epw$drop_unit()
epw$data()
}
```

Method `purge()`**:** Delete redundant weather data observations

Usage:

```
Epw$purge()
```

Details: $purge() deletes weather data in `Epw` object that do not belong to any data period.

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
epw$purge()
}
```

Method `add()`**:** Add a data period

Usage:

```
Epw$add(
  data,
  realyear = FALSE,
  name = NULL,
  start_day_of_week = NULL,
  after = 0L
)
```

Arguments:

`data` A `data.table::data.table()` of new weather data to add or set. Validation is performed according to rules described above.

`realyear` Whether input data is AMY data. Default: `FALSE`.

`name` A new string used as name of added or set data period. Should not be the same as existing data period names. If `NULL`, it is generated automatically in format `Data`, `Data_1` and etc., based on existing data period names. Default: `NULL`

`start_day_of_week` A single integer or character specifying start day of week of input data period. If `NULL`, Sunday is used for TMY data and the actual start day of week is used for AMY data. Default: `NULL`.

`after` A single integer identifying the index of data period where input new data period to be inserted after. IF `0`, input new data period will be the first data period. Default: `0`.

Details: $add() adds a new data period into current `Epw` object at specified position.
The validity of input data is checked before adding according to rules following:

- Column `datetime` exists and has type of `POSIXct`. Note that time zone of input date time will be reset to `UTC`.
- It assumes that input data is already sorted, i.e. no further sorting is made during validation. This is because when input data is TMY data, there is no way to properly sort input data rows only using `datetime` column.
- Number of data records per hour should be consistent across input data.
- Input number of data records per hour should be the same as existing data periods.
- The date time of input data should not overlap with existing data periods.
- Input data should have all 29 weather data columns with correct types. The `year`, `month`, `day`, and `minute` column are not compulsory. They will be created according to values in the `datetime` column. Existing values will be overwritten.

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
# will fail since date time in input data has already been covered by
# existing data period
try(epw$add(epw$data()), silent = TRUE)
}
```

Method `set()`**:** Replace a data period

Usage:

```
Epw$set(
  data,
```

```
  realyear = FALSE,
  name = NULL,
  start_day_of_week = NULL,
  period = 1L
)
```

Arguments:

`data` A `data.table::data.table()` of new weather data to add or set. Validation is performed according to rules described above.

`realyear` Whether input data is AMY data. Default: `FALSE`.

`name` A new string used as name of added or set data period. Should not be the same as existing data period names. If `NULL`, it is generated automatically in format `Data`, `Data_1` and etc., based on existing data period names. Default: `NULL`

`start_day_of_week` A single integer or character specifying start day of week of input data period. If `NULL`, Sunday is used for TMY data and the actual start day of week is used for AMY data. Default: `NULL`.

`period` A single integer identifying the index of data period to set.

Details: $set() replaces existing data period using input new weather data.
The validity of input data is checked before replacing according to rules following:

- Column `datetime` exists and has type of `POSIXct`. Note that time zone of input date time will be reset to `UTC`.
- It assumes that input data is already sorted, i.e. no further sorting is made during validation. This is because when input data is TMY data, there is no way to properly sort input data rows only using `datetime` column.
- Number of data records per hour should be consistent across input data.
- Input number of data records per hour should be the same as existing data periods.
- The date time of input data should not overlap with existing data periods.
- Input data should have all 29 weather data columns with right types. The `year`, `month`, `day`, and `minute` column are not compulsory. They will be created according to values in the `datetime` column. Existing values will be overwritten.

Returns: The modified `Epw` object itself, invisibly.

Examples:

```
\dontrun{
# change the weather data
epw$set(epw$data())
}
```

Method `del()`**:** Delete a data period

Usage:

```
Epw$del(period)
```

Arguments:

`period` A single integer identifying the index of data period to set.

Details: $del() removes a specified data period. Note that an error will be given if current `Epw` only contains one data period.

Returns: The modified Epw object itself, invisibly.

Method `is_unsaved()`**:** Check if there are unsaved changes in current Epw

Usage:

```
Epw$is_unsaved()
```

Details: $is_unsaved() returns TRUE if there are modifications on the Epw object since it was read or since last time it was saved, and returns FALSE otherwise.

Returns: A single logical value of TRUE or FALSE.

Examples:

```
\dontrun{
epw$is_unsaved()
}
```

Method `save()`**:** Save Epw object as an EPW file

Usage:

```
Epw$save(path = NULL, overwrite = FALSE, purge = FALSE, format_digit = TRUE)
```

Arguments:

`path` A path where to save the weather file. If NULL, the path of the weather file itself is used. Default: NULL.

`overwrite` Whether to overwrite the file if it already exists. Default is FALSE.

`purge` Whether to remove redundant data when saving. Default: FALSE.

`format_digit` Whether to remove trailing digits in weather data. Default: TRUE.

Details: $save() saves current Epw to an EPW file. Note that if missing values and out-of-range values are converted to NAs using `$make_na()`, they will be filled with corresponding missing codes during saving.

Returns: A length-one character vector, invisibly.

Examples:

```
\dontrun{
# save the weather file
epw$save(file.path(tempdir(), "weather.epw"), overwrite = TRUE)
}
```

Method `print()`**:** Print Idf object

Usage:

```
Epw$print()
```

Details: $print() prints the Epw object, including location, elevation, data source, WMO station, leap year indicator, interval and data periods.

Returns: The Epw object itself, invisibly.

Examples:

```
\dontrun{
epw$print()
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Epw$clone(deep = TRUE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Hongyuan Jia

Examples

```
## ------------------------------------------------
## Method `Epw$new`
## ------------------------------------------------

## Not run:
# read an EPW file from EnergyPlus website
path_base <- "https://energyplus.net/weather-download"
path_region <- "north_and_central_america_wmo_region_4/USA/CA"
path_file <- "USA_CA_San.Francisco.Intl.AP.724940_TMY3/USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
path_epw <- file.path(path_base, path_region, path_file)
epw <- read_epw(path_epw)

# read an EPW file distributed with EnergyPlus
if (is_avail_eplus(8.8)) {
    path_epw <- file.path(
        eplus_config(8.8)$dir,
        "WeatherData",
        "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
    )
    epw <- read_epw(path_epw)
}

## End(Not run)


## ------------------------------------------------
## Method `Epw$path`
## ------------------------------------------------

## Not run:
# get path
epw$path()
```

```
## End(Not run)


## ------------------------------------------------
## Method `Epw$definition`
## ------------------------------------------------

## Not run:
# get path
epw$definition("LOCATION")

## End(Not run)


## ------------------------------------------------
## Method `Epw$location`
## ------------------------------------------------

## Not run:
epw$location()

# modify location data
epw$location(city = "MyCity")

## End(Not run)


## ------------------------------------------------
## Method `Epw$design_condition`
## ------------------------------------------------

## Not run:
epw$design_condition()

## End(Not run)


## ------------------------------------------------
## Method `Epw$typical_extreme_period`
## ------------------------------------------------

## Not run:
epw$typical_extreme_period()

## End(Not run)


## ------------------------------------------------
## Method `Epw$ground_temperature`
## ------------------------------------------------

## Not run:
epw$ground_temperature()
```

```
## End(Not run)


## ------------------------------------------------
## Method `Epw$holiday`
## ------------------------------------------------

## Not run:
epw$holiday()

# add daylight saving time
epw$holiday(dst = c(3.10, 11.3))

## End(Not run)


## ------------------------------------------------
## Method `Epw$comment1`
## ------------------------------------------------

## Not run:
epw$comment1()

epw$comment1("Comment1")

## End(Not run)


## ------------------------------------------------
## Method `Epw$comment2`
## ------------------------------------------------

## Not run:
epw$comment2()

epw$comment2("Comment2")

## End(Not run)


## ------------------------------------------------
## Method `Epw$num_period`
## ------------------------------------------------

## Not run:
epw$num_period()

## End(Not run)


## ------------------------------------------------
## Method `Epw$interval`
```

```
## ------------------------------------------------

## Not run:
epw$interval()

## End(Not run)


## ------------------------------------------------
## Method `Epw$period`
## ------------------------------------------------

## Not run:
# modify data period name
epw$period(1, name = "test")

# change start day of week
epw$period(1, start_day_of_week = 3)

## End(Not run)


## ------------------------------------------------
## Method `Epw$missing_code`
## ------------------------------------------------

## Not run:
epw$missing_code()

## End(Not run)


## ------------------------------------------------
## Method `Epw$initial_missing_value`
## ------------------------------------------------

## Not run:
epw$initial_missing_value()

## End(Not run)


## ------------------------------------------------
## Method `Epw$range_exist`
## ------------------------------------------------

## Not run:
epw$range_exist()

## End(Not run)


## ------------------------------------------------
```

```
## Method `Epw$range_valid`
## ------------------------------------------------

## Not run:
epw$range_valid()

## End(Not run)


## ------------------------------------------------
## Method `Epw$fill_action`
## ------------------------------------------------

## Not run:
epw$fill_action("missing")

epw$fill_action("out_of_range")

## End(Not run)


## ------------------------------------------------
## Method `Epw$data`
## ------------------------------------------------

## Not run:
# get weather data
str(epw$data())

# get weather data but change the year to 2018
# the year column is not changed by default, only the returned datetime column
head(epw$data(start_year = 2018)$datetime)
str(epw$data(start_year = 2018)$year)
# you can update the year column too
head(epw$data(start_year = 2018, update = TRUE)$year)

# change the time zone of datetime column in the returned weather data
attributes(epw$data()$datetime)
attributes(epw$data(tz = "Etc/GMT+8")$datetime)

## End(Not run)


## ------------------------------------------------
## Method `Epw$abnormal_data`
## ------------------------------------------------

## Not run:
epw$abnormal_data()

# only check if there are any abnormal values in air temperature and
# liquid precipitation rate
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"))
```

```
# save as above, but only return date time columns plus those 2 columns
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"),
    keep_all = FALSE
)

# same as above, but only check for missing values
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"),
    type = "missing"
)

# same as above, but only check for out-of-range values
epw$abnormal_data(cols = c("dry_bulb_temperature", "liquid_precip_rate"),
    type = "out_of_range"
)

## End(Not run)


## ------------------------------------------------
## Method `Epw$redundant_data`
## ------------------------------------------------

## Not run:
epw$redundant_data()

## End(Not run)


## ------------------------------------------------
## Method `Epw$make_na`
## ------------------------------------------------

## Not run:
# turn all missing values into NAs
summary(epw$data()$liquid_precip_rate)
epw$make_na(missing = TRUE)
summary(epw$data()$liquid_precip_rate)

## End(Not run)


## ------------------------------------------------
## Method `Epw$fill_abnormal`
## ------------------------------------------------

## Not run:
# turn all missing values into NAs
summary(epw$data()$liquid_precip_rate)
epw$fill_abnormal(missing = TRUE)
summary(epw$data()$liquid_precip_rate)

## End(Not run)
```

```
## ------------------------------------------------
## Method `Epw$add_unit`
## ------------------------------------------------

## Not run:
# get weather data with units
epw$add_unit()
head(epw$data())

# with units specified, you can easily perform unit conversion using units
# package
t_dry_bulb <- epw$data()$dry_bulb_temperature
units(t_dry_bulb) <- with(units::ud_units, "kelvin")

head(t_dry_bulb)

## End(Not run)


## ------------------------------------------------
## Method `Epw$drop_unit`
## ------------------------------------------------

## Not run:
epw$drop_unit()
epw$data()

## End(Not run)


## ------------------------------------------------
## Method `Epw$purge`
## ------------------------------------------------

## Not run:
epw$purge()

## End(Not run)


## ------------------------------------------------
## Method `Epw$add`
## ------------------------------------------------

## Not run:
# will fail since date time in input data has already been covered by
# existing data period
try(epw$add(epw$data()), silent = TRUE)

## End(Not run)
```

```
## ------------------------------------------------
## Method `Epw$set`
## ------------------------------------------------

## Not run:
# change the weather data
epw$set(epw$data())

## End(Not run)


## ------------------------------------------------
## Method `Epw$is_unsaved`
## ------------------------------------------------

## Not run:
epw$is_unsaved()

## End(Not run)


## ------------------------------------------------
## Method `Epw$save`
## ------------------------------------------------

## Not run:
# save the weather file
epw$save(file.path(tempdir(), "weather.epw"), overwrite = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Epw$print`
## ------------------------------------------------

## Not run:
epw$print()

## End(Not run)
```

`format.Idd` *Format an Idd*

Description

Format an Idd into a string.

Usage

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

Arguments

x	An Idd object.
...	Further arguments passed to or from other methods.

Value

A single length character vector.

Examples

```
## Not run:
format(use_idd(8.8, download = "auto"))

## End(Not run)
```

`format.IddObject` *Format an IddObject*

Description

Format an IddObject into a string. It is formatted the same way as `IddObject$print(brief = TRUE)` but with a suffix of current IDD version.

Usage

```
## S3 method for class 'IddObject'
format(x, ver = TRUE, ...)
```

Arguments

x	An IddObject object.
ver	If `TRUE`, a suffix of version string is added. Default: `TRUE`.
...	Further arguments passed to or from other methods.

Value

A single length character vector.

Examples

```
## Not run:
format(use_idd(8.8, download = "auto")$Material)

## End(Not run)
```

`format.Idf` *Format an Idf Object*

Description

Format an Idf object.

Usage

```
## S3 method for class 'Idf'
format(
  x,
  comment = TRUE,
  header = TRUE,
  format = eplusr_option("save_format"),
  leading = 4L,
  sep_at = 29L,
  ...
)
```

Arguments

`x`	An Idf object.
`comment`	If `FALSE`, all comments will not be included. Default: `TRUE`.
`header`	If `FALSE`, the header will not be included. Default: `TRUE`.
`format`	Specific format used when formatting. For details, please see $save(). Default: `eplusr_option("save_format")`
`leading`	Leading spaces added to each field. Default: `4L`.
`sep_at`	The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.
`...`	Further arguments passed to or from other methods.

Value

A single length string.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
idf_path <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")
cat(format(read_idf(idf_path, use_idd(8.8, "auto")), leading = 0))

## End(Not run)
```

`format.IdfObject` *Format an IdfObject*

Description

Format an IddObject into a character vector in the same way as in IDF Editor.

Usage

```
## S3 method for class 'IdfObject'
format(x, comment = TRUE, leading = 4L, sep_at = 29L, all = FALSE, ...)
```

Arguments

`x`	An IdfObject object.
`comment`	If `FALSE`, all comments will not be included. Default: `TRUE`.
`leading`	Leading spaces added to each field. Default: `4L`.
`sep_at`	The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.
`all`	If `TRUE`, values of all possible fields in current class the IdfObject belongs to are returned. Default: `FALSE`
`...`	Further arguments passed to or from other methods.

Value

A character vector.

Examples

```
## Not run:
idf <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"),
    idd = use_idd(8.8, download = "auto"))

# get the IdfObject of material named "C5 - 4 IN HW CONCRETE"
mat <- idf$Material[["C5 - 4 IN HW CONCRETE"]]

cat(format(mat, leading = 0, sep_at = 10))

## End(Not run)
```

Idd *Parse, Query and Modify EnergyPlus Input Data Dictionary (IDD)*

Description

eplusr provides parsing of and programmatic access to EnergyPlus Input Data Dictionary (IDD) files, and objects. It contains all data needed to parse EnergyPlus models. `Idd` class provides parsing and printing while IddObject provides detailed information of curtain class.

Overview

EnergyPlus operates off of text input files written in its own Input Data File (IDF) format. IDF files are similar to XML files in that they are intended to conform to a data schema written using similar syntax. For XML, the schema format is XSD; for IDF, the schema format is IDD. For each release of EnergyPlus, valid IDF files are defined by the "Energy+.idd" file shipped with the release.

eplusr tries to detect all installed EnergyPlus in default installation locations when loading, i.e. C:\\EnergyPlusVX-X-0 on Windows, /usr/local/EnergyPlus-X-Y-0 on Linux, and /Applications/EnergyPlus-X-Y-0 on macOS and stores all found locations internally. This data is used to locate the distributed "Energy+.idd" file of each EnergyPlus version. And also, every time an IDD file is parsed, an `Idd` object is created and cached in an environment.

Parsing an IDD file starts from `use_idd()`. When using `use_idd()`, eplusr will first try to find the cached `Idd` object of that version, if possible. If failed, and EnergyPlus of that version is available (see `avail_eplus()`), the `"Energy+.idd"` distributed with EnergyPlus will be parsed and cached. So each IDD file only needs to be parsed once and can be used when parsing every IDF file of that version.

Internally, the powerful data.table package is used to speed up the whole IDD parsing process and store the results. However, it will still take about 2-3 sec per IDD. Under the hook, eplusr uses a SQL-like structure to store both IDF and IDD data in data.table::data.table format. Every IDD will be parsed and stored in four tables:

- `group`: contains group index and group names.
- `class`: contains class names and properties.
- `field`: contains field names and field properties.
- `reference`: contains cross-reference data of fields.

Methods

Public methods:

- `Idd$new()`
- `Idd$version()`
- `Idd$build()`
- `Idd$group_name()`
- `Idd$from_group()`
- `Idd$class_name()`

- Idd$required_class_name()
- Idd$unique_class_name()
- Idd$extensible_class_name()
- Idd$group_index()
- Idd$class_index()
- Idd$is_valid_group()
- Idd$is_valid_class()
- Idd$object()
- Idd$objects()
- Idd$object_relation()
- Idd$objects_in_relation()
- Idd$objects_in_group()
- Idd$to_table()
- Idd$to_string()
- Idd$print()

Method `new()`**:** Create an `Idd` object

Usage:

```
Idd$new(path)
```

Arguments:

`path` Either a path, a connection, or literal data (either a single string or a raw vector) to an EnergyPlus Input Data Dictionary (IDD). If a file path, that file usually has a extension `.idd`.

Details: It takes an EnergyPlus Input Data Dictionary (IDD) as input and returns an `Idd` object. It is suggested to use helper `use_idd()` which supports to directly take a valid IDD version as input and search automatically the corresponding file path.

Returns: An `Idd` object.

Examples:

```
\dontrun{Idd$new(file.path(eplus_config(8.8)$dir, "Energy+.idd"))

# Preferable way
idd <- use_idd(8.8, download = "auto")
}
```

Method `version()`**:** Get the version of current `Idd`

Usage:

```
Idd$version()
```

Details: $version() returns the version of current `Idd` in a `base::numeric_version()` format. This makes it easy to direction compare versions of different `Idds`, e.g. `idd$version() > 8.6` or `idd1$version() > idd2$version()`.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
# get version
idd$version()
}
```

Method `build()`**:** Get the build tag of current `Idd`

Usage:

```
Idd$build()
```

Details: $build() returns the build tag of current `Idd`. If no build tag is found, `NA` is returned.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
# get build tag
idd$build()
}
```

Method `group_name()`**:** Get names of groups

Usage:

```
Idd$group_name()
```

Details: $group_name() returns names of groups current `Idd` contains.

Returns: A character vector.

Examples:

```
\dontrun{
# get names of all groups Idf contains
idd$group_name()
}
```

Method `from_group()`**:** Get the name of group that specified class belongs to

Usage:

```
Idd$from_group(class)
```

Arguments:

`class` A character vector of valid class names in current `Idd`.

Details: $from_group() returns the name of group that specified class belongs to.

Returns: A character vector.

Examples:

```
\dontrun{
idd$from_group(c("Version", "Schedule:Compact"))
}
```

Method `class_name()`**:** Get names of classes

Usage:

```
Idd$class_name(index = NULL, by_group = FALSE)
```

Arguments:

`index` An integer vector of class indices.

`by_group` If `TRUE`, a list is returned which separates class names by the group they belong to. Default: `FALSE`.

Details: $class_name() returns names of classes current `Idd` contains

Returns: A character vector if `by_group` is `FALSE` and a list of character vectors when `by_group` is `TRUE`.

Examples:

```
\dontrun{
# get names of the 10th to 20th class
idd$class_name(10:20)

# get names of all classes in Idf
idd$class_name()

# get names of all classes grouped by group names in Idf
idd$class_name(by_group = TRUE)
}
```

Method `required_class_name()`**:** Get the names of required classes

Usage:

```
Idd$required_class_name()
```

Details: $required_class_name() returns the names of required classes in current `Idd`. "Require" means that for any Idf there should be at least one object.

Returns: A character vector.

Examples:

```
\dontrun{
idd$required_class_name()
}
```

Method `unique_class_name()`**:** Get the names of unique-object classes

Usage:

```
Idd$unique_class_name()
```

Details: $unique_class_name() returns the names of unique-object classes in current `Idd`. "Unique-object" means that for any Idf there should be at most one object in those classes.

Returns: A character vector.

Examples:

```
\dontrun{
idd$unique_class_name()
}
```

Method `extensible_class_name()`**:** Get the names of classes with extensible fields

Usage:

```
Idd$extensible_class_name()
```

Details: $extensible_class_name() returns the names of classes with extensible fields in current `Idd`. "Extensible fields" indicate fields that can be added dynamically, such like the X, Y and Z vertices of a building surface.

Returns: A character vector.

Examples:

```
\dontrun{
idd$extensible_class_name()
}
```

Method `group_index()`**:** Get the indices of specified groups

Usage:

```
Idd$group_index(group = NULL)
```

Arguments:

`group` A character vector of valid group names.

Details: $group_index() returns the indices of specified groups in current `Idd`. A group index is just an integer indicating its appearance order in the `Idd`.

Returns: An integer vector.

Examples:

```
\dontrun{
idd$group_index()
}
```

Method `class_index()`**:** Get the indices of specified classes

Usage:

```
Idd$class_index(class = NULL, by_group = FALSE)
```

Arguments:

`class` A character vector of valid class names.

`by_group` If `TRUE`, a list is returned which separates class names by the group they belong to. Default: `FALSE`.

Details: $class_index() returns the indices of specified classes in current `Idd`. A class index is just an integer indicating its appearance order in the `Idd`.

Returns: An integer vector.

Examples:

```
\dontrun{
idd$class_index()
}
```

Method `is_valid_group()`**:** Check if elements in input character vector are valid group names.

Usage:

```
Idd$is_valid_group(group)
```

Arguments:

`group` A character vector to check.

Details: $is_valid_group() returns `TRUE`s if given character vector contains valid group names in the context of current `Idd`, and `FALSE`s otherwise.
Note that case-sensitive matching is performed, which means that `"Location and Climate"` is a valid group name but `"location and climate"` is not.

Returns: A logical vector with the same length as input character vector.

Examples:

```
\dontrun{
idd$is_valid_group(c("Schedules", "Compliance Objects"))
}
```

Method `is_valid_class()`**:** Check if elements in input character vector are valid class names.

Usage:

```
Idd$is_valid_class(class)
```

Arguments:

`class` A character vector to check.

Details: $is_valid_class() returns `TRUE`s if given character vector contains valid class names in the context of current `Idd`, and `FALSE`s otherwise.
Note that case-sensitive matching is performed, which means that `"Version"` is a valid class name but `"version"` is not.

Returns: A logical vector with the same length as input character vector.

Examples:

```
\dontrun{
idd$is_valid_class(c("Building", "ShadowCalculation"))
}
```

Method `object()`**:** Extract an IddObject object using class index or name.

Usage:

```
Idd$object(class)
```

Arguments:

`class` A single integer specifying the class index or a single string specifying the class name.

Details: $object() returns an IddObject object specified by a class ID or name.
Note that case-sensitive matching is performed, which means that `"Version"` is a valid class name but `"version"` is not.
For convenience, underscore-style names are allowed, e.g. `Site_Location` is equivalent to `Site:Location`.

Returns: An IddObject object.

Examples:

```
\dontrun{
idd$object(3)

idd$object("Building")
}
```

Method `objects()`**:** Extract multiple IddObject objects using class indices or names.

Usage:

```
Idd$objects(class)
```

Arguments:

`class` An integer vector specifying class indices or a character vector specifying class names.

Details: $objects() returns a named list of IddObject objects using class indices or names. The returned list is named using class names.
Note that case-sensitive matching is performed, which means that `"Version"` is a valid class name but `"version"` is not.
For convenience, underscore-style names are allowed, e.g. `Site_Location` is equivalent to `Site:Location`.

Returns: A named list of IddObject objects.

Examples:

```
\dontrun{
idd$objects(c(3,10))

idd$objects(c("Version", "Material"))
}
```

Method `object_relation()`**:** Extract the relationship between class fields.

Usage:

```
Idd$object_relation(
  which,
  direction = c("all", "ref_to", "ref_by"),
  class = NULL,
  group = NULL,
  depth = 0L
)
```

Arguments:

`which` A single integer specifying the class index or a single string specifying the class name.

`direction` The relation direction to extract. Should be one of `"all"`, `"ref_to"` or `"ref_by"`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

Details: Many fields in Idd can be referred by others. For example, the Outside Layer and other fields in `Construction` class refer to the `Name` field in `Material` class and other material related classes. Here it means that the Outside Layer field **refers to** the `Name` field and the `Name` field is **referred by** the Outside Layer.
$object_relation() provides a simple interface to get this kind of relation. It takes a single class index or name and also a relation direction, and returns an `IddRelation` object which contains data presenting such relation above. For instance, if `idd$object_relation("Construction","ref_to")` gives results below:

```
-- Refer to Others ---------------------------
  Class: <Construction>
  |- Field: <02: Outside Layer>
  |  v~~~~~~~~~~~~~~~~~~~~~~~~
  |  |- Class: <Material>
  |  |  \- Field: <1: Name>
  |  |
  |  |- Class: <Material:NoMass>
  |  |  \- Field: <1: Name>
  |  |
  |  |- Class: <Material:InfraredTransparent>
  |  |  \- Field: <1: Name>
  |  |
  ......
```

This means that the value of field Outside Layer in class `Construction` can be one of values from field `Name` in class `Material`, field `Name` in class `Material:NoMass`, field `Name` in class `Material:InfraredTransparent` and etc. All those classes can be further easily extracted using $objects_in_relation() method described below.

Returns: An `IddRelation` object, which is a list of 3 `data.table::data.table()`s named `ref_to` and `ref_by`. Each `data.table::data.table()` contains 12 columns.

Examples:

```
\dontrun{
# check each construction layer's possible references
idd$object_relation("Construction", "ref_to")

# check where construction being used
idd$object_relation("Construction", "ref_by")
}
```

Method `objects_in_relation()`**:** Extract multiple IddObject objects referencing each others.

Usage:

```
Idd$objects_in_relation(
  which,
  direction = c("ref_to", "ref_by"),
  class = NULL,
  group = NULL,
  depth = 0L
)
```

Arguments:

`which` A single integer specifying the class index or a single string specifying the class name.

`direction` The relation direction to extract. Should be either `"ref_to"` or `"ref_by"`.

`class` A character vector of valid class names in the current Idd. It is used to restrict the classes to be returned. If `NULL`, all possible classes are considered and corresponding IddObject objects are returned if relationships are found. Default: `NULL`.

`group` A character vector of valid group names in the current Idd. It is used to restrict the groups to be returned. If `NULL`, all possible groups are considered and corresponding IddObject objects are returned if relationships are found. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

Details: $objects_in_relation() returns a named list of IddObject objects that have specified relationship with given class. The first element of returned list is always the specified class itself. If that class does not have specified relationship with other classes, a list that only contains specified class itself is returned.
For instance, `idd$objects_in_relation("Construction","ref_by")` will return a named list of an IddObject object named `Construction` and also all other IddObject objects that can refer to field values in class `Construction`. Similarly, `idd$objects_in_relation("Construction","ref_to")` will return a named list of an IddObject object named `Construction` and also all other IddObject objects that `Construction` can refer to.

Returns: An named list of IddObject objects.

Examples:

```
\dontrun{
# get class Construction and all classes that it can refer to
idd$objects_in_relation("Construction", "ref_to")

# get class Construction and all classes that refer to it
idd$objects_in_relation("Construction", "ref_by")
}
```

Method `objects_in_group()`**:** Extract all IddObject objects in one group.

Usage:

```
Idd$objects_in_group(group)
```

Arguments:

`group` A single string of valid group name for current `Idd` object.

Details: $objects_in_group() returns a named list of all IddObject objects in specified group. The returned list is named using class names.

Returns: A named list of IddObject objects.

Examples:

```
\dontrun{
# get all classes in Schedules group
idd$objects_in_group("Schedules")
}
```

Method `to_table()`**:** Format `Idd` classes as a data.frame

Usage:

```
Idd$to_table(class, all = FALSE)
```

Arguments:

`class` A character vector of class names.

`all` If `TRUE`, all available fields defined in IDD for specified class will be returned. Default: `FALSE`.

Details: $to_table() returns a data.table that contains basic data of specified classes. The returned data.table has 3 columns:

- `class`: Character type. Current class name.
- `index`: Integer type. Field indexes.
- `field`: Character type. Field names.

Returns: A data.table with 3 columns.

Examples:

```
\dontrun{
# extract data of class Material
idd$to_table(class = "Material")

# extract multiple class data
idd$to_table(c("Construction", "Material"))
}
```

Method `to_string()`**:** Format `Idf` classes as a character vector

Usage:

```
Idd$to_string(class, leading = 4L, sep_at = 29L, sep_each = 0L, all = FALSE)
```

Arguments:

`class` A character vector of class names.

`leading` Leading spaces added to each field. Default: `4L`.

`sep_at` The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.

`sep_each` A single integer of how many empty strings to insert between different classes. Default: `0`.

`all` If `TRUE`, all available fields defined in IDD for specified class will be returned. Default: `FALSE`.

Details: $to_string() returns the text format of specified classes. The returned character vector can be pasted into an IDF file as empty objects of specified classes.

Returns: A character vector.

Examples:

```
\dontrun{
# get text format of class Material
head(idd$to_string(class = "Material"))

# get text format of multiple class
idd$to_string(c("Material", "Construction"))

# tweak output formatting
idd$to_string(c("Material", "Construction"), leading = 0, sep_at = 0, sep_each = 5)
}
```

Method `print()`**:** Print `Idd` object

Usage:

```
Idd$print()
```

Details: $print() prints the `Idd` object giving the information of version, build tag and total class numbers.

Returns: The `Idd` object itself, invisibly.

Examples:

```
\dontrun{
idd$print()
}
```

Author(s)

Hongyuan Jia

References

IDFEditor, OpenStudio utilities library

See Also

IddObject class which provides detailed information of curtain class

Examples

```
## ------------------------------------------------
## Method `Idd$new`
## ------------------------------------------------

## Not run: Idd$new(file.path(eplus_config(8.8)$dir, "Energy+.idd"))

# Preferable way
idd <- use_idd(8.8, download = "auto")

## End(Not run)


## ------------------------------------------------
## Method `Idd$version`
## ------------------------------------------------

## Not run:
# get version
idd$version()

## End(Not run)


## ------------------------------------------------
## Method `Idd$build`
## ------------------------------------------------

## Not run:
# get build tag
idd$build()

## End(Not run)


## ------------------------------------------------
## Method `Idd$group_name`
## ------------------------------------------------

## Not run:
# get names of all groups Idf contains
idd$group_name()

## End(Not run)


## ------------------------------------------------
## Method `Idd$from_group`
## ------------------------------------------------

## Not run:
```

```
idd$from_group(c("Version", "Schedule:Compact"))

## End(Not run)


## ------------------------------------------------
## Method `Idd$class_name`
## ------------------------------------------------

## Not run:
# get names of the 10th to 20th class
idd$class_name(10:20)

# get names of all classes in Idf
idd$class_name()

# get names of all classes grouped by group names in Idf
idd$class_name(by_group = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idd$required_class_name`
## ------------------------------------------------

## Not run:
idd$required_class_name()

## End(Not run)


## ------------------------------------------------
## Method `Idd$unique_class_name`
## ------------------------------------------------

## Not run:
idd$unique_class_name()

## End(Not run)


## ------------------------------------------------
## Method `Idd$extensible_class_name`
## ------------------------------------------------

## Not run:
idd$extensible_class_name()

## End(Not run)


## ------------------------------------------------
```

```
## Method `Idd$group_index`
## ------------------------------------------------

## Not run:
idd$group_index()

## End(Not run)


## ------------------------------------------------
## Method `Idd$class_index`
## ------------------------------------------------

## Not run:
idd$class_index()

## End(Not run)


## ------------------------------------------------
## Method `Idd$is_valid_group`
## ------------------------------------------------

## Not run:
idd$is_valid_group(c("Schedules", "Compliance Objects"))

## End(Not run)


## ------------------------------------------------
## Method `Idd$is_valid_class`
## ------------------------------------------------

## Not run:
idd$is_valid_class(c("Building", "ShadowCalculation"))

## End(Not run)


## ------------------------------------------------
## Method `Idd$object`
## ------------------------------------------------

## Not run:
idd$object(3)

idd$object("Building")

## End(Not run)


## ------------------------------------------------
## Method `Idd$objects`
```

```
## ------------------------------------------------

## Not run:
idd$objects(c(3,10))

idd$objects(c("Version", "Material"))

## End(Not run)


## ------------------------------------------------
## Method `Idd$object_relation`
## ------------------------------------------------

## Not run:
# check each construction layer's possible references
idd$object_relation("Construction", "ref_to")

# check where construction being used
idd$object_relation("Construction", "ref_by")

## End(Not run)


## ------------------------------------------------
## Method `Idd$objects_in_relation`
## ------------------------------------------------

## Not run:
# get class Construction and all classes that it can refer to
idd$objects_in_relation("Construction", "ref_to")

# get class Construction and all classes that refer to it
idd$objects_in_relation("Construction", "ref_by")

## End(Not run)


## ------------------------------------------------
## Method `Idd$objects_in_group`
## ------------------------------------------------

## Not run:
# get all classes in Schedules group
idd$objects_in_group("Schedules")

## End(Not run)


## ------------------------------------------------
## Method `Idd$to_table`
## ------------------------------------------------
```

```
## Not run:
# extract data of class Material
idd$to_table(class = "Material")

# extract multiple class data
idd$to_table(c("Construction", "Material"))

## End(Not run)


## ------------------------------------------------
## Method `Idd$to_string`
## ------------------------------------------------

## Not run:
# get text format of class Material
head(idd$to_string(class = "Material"))

# get text format of multiple class
idd$to_string(c("Material", "Construction"))

# tweak output formatting
idd$to_string(c("Material", "Construction"), leading = 0, sep_at = 0, sep_each = 5)

## End(Not run)


## ------------------------------------------------
## Method `Idd$print`
## ------------------------------------------------

## Not run:
idd$print()

## End(Not run)
```

`IddObject` *EnergyPlus IDD object*

Description

`IddObject` is an abstraction of a single object in an Idd object. It provides more detail methods to query field properties. `IddObject` can only be created from the parent Idd object, using $object(), $object_in_group() and other equivalent. This is because that initialization of an `IddObject` needs some shared data from parent Idd object.

Details

There are lots of properties for every class and field. For details on the meaning of each property, please see the heading comments in the `Energy+.idd` file in the EnergyPlus installation path.

Methods

Public methods:

- IddObject$new()
- IddObject$version()
- IddObject$parent()
- IddObject$group_name()
- IddObject$group_index()
- IddObject$class_name()
- IddObject$class_index()
- IddObject$class_format()
- IddObject$min_fields()
- IddObject$num_fields()
- IddObject$memo()
- IddObject$num_extensible()
- IddObject$first_extensible_index()
- IddObject$extensible_group_num()
- IddObject$add_extensible_group()
- IddObject$del_extensible_group()
- IddObject$has_name()
- IddObject$is_required()
- IddObject$is_unique()
- IddObject$is_extensible()
- IddObject$field_name()
- IddObject$field_index()
- IddObject$field_type()
- IddObject$field_note()
- IddObject$field_unit()
- IddObject$field_default()
- IddObject$field_choice()
- IddObject$field_range()
- IddObject$field_relation()
- IddObject$field_possible()
- IddObject$is_valid_field_num()
- IddObject$is_extensible_index()
- IddObject$is_valid_field_name()
- IddObject$is_valid_field_index()
- IddObject$is_autosizable_field()
- IddObject$is_autocalculatable_field()
- IddObject$is_numeric_field()
- IddObject$is_real_field()
- IddObject$is_integer_field()
- IddObject$is_required_field()

- IddObject$has_ref()
- IddObject$has_ref_to()
- IddObject$has_ref_by()
- IddObject$to_table()
- IddObject$to_string()
- IddObject$print()

Method `new()`**:** Create an `IddObject` object

Usage:

```
IddObject$new(class, parent)
```

Arguments:

`class` A single integer specifying the class index or a single string specifying the class name.

`parent` An Idd object or a valid input for `use_idd()`.

Details: Note that an `IddObject` can be created from the parent Idd object, using $object(), idd_object and other equivalent.

Returns: An `IddObject` object.

Examples:

```
\dontrun{
surf <- IddObject$new("BuildingSurface:Detailed", use_idd(8.8, download = "auto"))
}
```

Method `version()`**:** Get the version of parent `Idd`

Usage:

```
IddObject$version()
```

Details: $version() returns the version of parent `Idd` in a `base::numeric_version()` format. This makes it easy to direction compare versions of different `IddObjects`, e.g. `iddobj$version() > 8.6` or `iddobj1$version() > iddobj2$version()`.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
# get version
surf$version()
}
```

Method `parent()`**:** Get parent Idd

Usage:

```
IddObject$parent()
```

Details: $parent() returns parent Idd object.

Returns: A Idd object.

Examples:

```
\dontrun{
surf$parent()
}
```

Method `group_name()`**:** Get the group name

Usage:

```
IddObject$group_name()
```

Details: $group_name() returns the group name of current `IddObject`.

Returns: A single string.

Examples:

```
\dontrun{
surf$group_name()
}
```

Method `group_index()`**:** Get the group index

Usage:

```
IddObject$group_index()
```

Details: $group_index() returns the group index of current `IddObject`. A group index is just an integer indicating its appearance order in the Idd.

Returns: A single integer.

Examples:

```
\dontrun{
surf$group_index()
}
```

Method `class_name()`**:** Get the class name of current `IddObject`

Usage:

```
IddObject$class_name()
```

Details: $class_name() returns the class name of current `IddObject`.

Returns: A single string.

Examples:

```
\dontrun{
surf$class_name()
}
```

Method `class_index()`**:** Get the class index

Usage:

```
IddObject$class_index()
```

Details: $class_index() returns the class index of current `IddObject`. A class index is just an integer indicating its appearance order in the Idd.

Returns: A single integer.

Examples:

```
\dontrun{
surf$class_index()
}
```

Method `class_format()`**:** Get the class format

Usage:

```
IddObject$class_format()
```

Details: $class_format() returns the format of this IDD class. This format indicator is currently not used by eplusr.

Returns: A single character.

Examples:

```
\dontrun{
surf$class_format()
}
```

Method `min_fields()`**:** Get the minimum field number of current class

Usage:

```
IddObject$min_fields()
```

Details: $min_fields() returns the minimum fields required for current class. If no required, `0` is returned.

Returns: A single integer.

Examples:

```
\dontrun{
surf$min_fields()
}
```

Method `num_fields()`**:** Get the total field number of current class

Usage:

```
IddObject$num_fields()
```

Details: $num_fields() returns current total number of fields in current class.

Returns: A single integer.

Examples:

```
\dontrun{
surf$num_fields()
}
```

Method `memo()`**:** Get the memo string of current class

Usage:

```
IddObject$memo()
```

Details: $memo() returns memo of current class, usually a brief description of this class.

Returns: A character vector.

Examples:

```
\dontrun{
surf$memo()
}
```

Method `num_extensible()`**:** Get the field number of the extensible group in current class

Usage:

```
IddObject$num_extensible()
```

Details: $num_extensible() returns the field number of the extensible group in current class. An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together.
If there is no extensible group in current class, `0` is returned.

Returns: A single integer.

Examples:

```
\dontrun{
surf$num_extensible()
}
```

Method `first_extensible_index()`**:** Get the minimum field number of current class

Usage:

```
IddObject$first_extensible_index()
```

Details: $first_extensible_index() returns the field index of first extensible field in current class. An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together.
If there is no extensible group in current class, `0` is returned.

Returns: A single integer.

Examples:

```
\dontrun{
surf$first_extensible_index()
}
```

Method `extensible_group_num()`**:** Get the number of extensible groups in current class

Usage:

```
IddObject$extensible_group_num()
```

Details: $extensible_group_num() returns the number of extensible groups in current class.
An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together.
If there is no extensible group in current class, `0` is returned.

Returns: A single integer.

Examples:

```
\dontrun{
surf$extensible_group_num()
}
```

Method `add_extensible_group()`**:** Add extensible groups in current class

Usage:

```
IddObject$add_extensible_group(num = 1L)
```

Arguments:

`num` An integer indicating the number of extensible groups to be added.

Details: $add_extensible_groups() adds extensible groups in this class.
An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together.
An error will be issued if current class contains no extensible group.

Returns: The modified `IddObject` itself.

Examples:

```
\dontrun{
# field number before adding
surf$num_fields()
# extensible group number before adding
surf$extensible_group_num()

# add 2 more extensible groups
surf$add_extensible_group(2)

# field number after adding
surf$num_fields()
# extensible group number after adding
surf$extensible_group_num()
}
```

Method `del_extensible_group()`**:** Delete extensible groups in current class

Usage:

```
IddObject$del_extensible_group(num = 1L)
```

Arguments:

`num` An integer indicating the number of extensible groups to be deleted.

Details: $del_extensible_groups() deletes extensible groups in this class.
An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together.
An error will be issued if current class contains no extensible group.

Returns: The modified `IddObject` itself.

Examples:

```
\dontrun{
# field number before deleting
surf$num_fields()
# extensible group number before deleting
surf$extensible_group_num()

# delete 2 more extensible groups
surf$del_extensible_group(2)

# field number after deleting
surf$num_fields()
# extensible group number after deleting
surf$extensible_group_num()
}
```

Method `has_name()`**:** Check if current class has name attribute

Usage:

```
IddObject$has_name()
```

Details: $has_name() return `TRUE` if current class has name attribute, and `FALSE` otherwise.
A class with name attribute means that objects in this class can have names.

Returns: A single logical value (`TRUE` or `FALSE`).

Examples:

```
\dontrun{
surf$has_name()
}
```

Method `is_required()`**:** Check if current class is required

Usage:

```
IddObject$is_required()
```

Details: $is_required() returns `TRUE` if current class is required and `FALSE` otherwise.
A required class means that for any model, there should be at least one object in this class. One example is `Building` class.

Returns: A single logical value (`TRUE` or `FALSE`).

Examples:

```
\dontrun{
surf$is_required()
}
```

Method `is_unique()`**:** Check if current class is unique

Usage:

```
IddObject$is_unique()
```

Details: $is_unique() returns `TRUE` if current class is unique and `FALSE` otherwise.
A unique class means that for any model, there should be at most one object in this class. One example is `Building` class.

Returns: A single logical value (`TRUE` or `FALSE`).

Examples:

```
\dontrun{
surf$is_unique()
}
```

Method `is_extensible()`**:** Check if current class is extensible

Usage:

```
IddObject$is_extensible()
```

Details: $is_extensible() returns `TRUE` if current class is extensible and `FALSE` otherwise.
A extensible class means that for there are curtain number of fields in this class that can be dynamically added or deleted, such like the X, Y and Z vertices of a building surface.

Returns: A single logical value (`TRUE` or `FALSE`).

Examples:

```
\dontrun{
surf$is_extensible()
}
```

Method `field_name()`**:** Get field names

Usage:

```
IddObject$field_name(
  index = NULL,
  unit = FALSE,
  in_ip = eplusr_option("view_in_ip")
)
```

Arguments:

`index` An integer vector of field indices. If `NULL`, names of all fields in this class are returned. Default: `NULL`.

`unit` If `TRUE`, the units of those fields are also returned. Default: `FALSE`.

`in_ip` If `in_ip`, corresponding imperial units are returned. It only has effect when `unit` is `TRUE`. Default: `eplusr_option("view_in_ip")`.

Details: $field_name() returns a character vector of names of fields specified by field indices in current class.

Returns: A character vector.

Examples:

```
\dontrun{
# get all field names
surf$field_name()

# get field units also
surf$field_name(unit = TRUE)

# get field units in IP
surf$field_name(unit = TRUE)

# change field name to lower-style
surf$field_name(unit = TRUE, in_ip = TRUE)
}
```

Method `field_index()`**:** Get field indices

Usage:

```
IddObject$field_index(name = NULL)
```

Arguments:

`name` A character vector of field names. Can be in "lower-style", i.e. all spaces and dashes is replaced by underscores. If `NULL`, indices of all fields in this class are returned. Default: `NULL`.

Details: $field_index() returns an integer vector of names of fields specified by field names in current class.

Returns: An integer vector.

Examples:

```
\dontrun{
# get all field indices
surf$field_index()

# get field indices for specific fields
surf$field_index(c("number of vertices", "vertex 10 z-coordinate"))
}
```

Method `field_type()`**:** Get field types

Usage:

```
IddObject$field_type(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $field_type() returns a character vector of field types of specified fields in current class. All possible values are:

- `"integer"`
- `"real"`

- "alpha" (arbitrary string)
- "choice" (alpha with specific list of choices)
- "object-list" (link to a list of objects defined elsewhere)
- "external-list" (uses a special list from an external source)
- "node" (name used in connecting HVAC components).

Returns: A character vector.

Examples:

```
\dontrun{
# get all field types
surf$field_type()

# get field types for specific fields
surf$field_type(c("name", "zone name", "vertex 10 z-coordinate"))
}
```

Method `field_note()`**:** Get field notes

Usage:

```
IddObject$field_note(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $field_note() returns a list of character vectors that contains field notes of specified fields in current class, usually serving as field descriptions. If no notes are found for current fields, `NULL` is returned.

Returns: A list of character vectors.

Examples:

```
\dontrun{
# get all field notes
surf$field_note()

# get field types for specific fields
surf$field_note(c("name", "zone name", "vertex 10 z-coordinate"))
}
```

Method `field_unit()`**:** Get field units

Usage:

```
IddObject$field_unit(which = NULL, in_ip = eplusr_option("view_in_ip"))
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

`in_ip` If `in_ip`, corresponding imperial units are returned. Default: `eplusr_option("view_in_ip")`.

Details: $field_unit() returns a character vector that contains units of specified fields in current class. If there is no unit found for current field, NA is returned.

Returns: A character vector.

Examples:

```
\dontrun{
# get all field units
surf$field_unit()

# get field units for specific fields
surf$field_unit(c("name", "zone name", "vertex 10 z-coordinate"))
}
```

Method `field_default()`**:** Get field default value

Usage:

```
IddObject$field_default(which = NULL, in_ip = eplusr_option("view_in_ip"))
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If NULL, all fields in this class are used. Default: NULL.

`in_ip` If in_ip, values in corresponding imperial units are returned. Default: `eplusr_option("view_in_ip")`.

Details: $field_default() returns a list that contains default values of specified fields in current class. If there is no default value found for current field, NA is returned.

Returns: A character vector.

Examples:

```
\dontrun{
# get all field default values
surf$field_default()

# get default values for specific fields
surf$field_default(c("name", "zone name", "vertex 10 z-coordinate"))
}
```

Method `field_choice()`**:** Get choices of field values

Usage:

```
IddObject$field_choice(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If NULL, all fields in this class are used. Default: NULL.

Details: $field_value() returns a list of character vectors that contains choices of specified field values in current class. If there is no choice found for current field, NULL is returned.

Returns: A list of character vectors.

Examples:

```
\dontrun{
# get all field value choices
surf$field_choice()

# get field value choices for specific fields
surf$field_choice(c("name", "sun exposure", "wind exposure"))
}
```

Method `field_range()`**:** Get field value ranges

Usage:

```
IddObject$field_range(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $field_range() returns a list of value ranges of specified fields in current class.
Every range has four components:

- `minimum`: lower limit
- `lower_incbounds`: `TRUE` if the lower limit should be included
- `maximum`: upper limit
- `upper_incbounds`: `TRUE` if the upper limit should be included

For fields of character type,

- `minimum` and `maximum` are always set to `NA`
- `lower_incbounds` and `upper_incbounds` are always set to `FALSE`

For fields of numeric types with no specified ranges,

- `minimum` is set to `-Inf`
- `lower_incbounds` is set to `FALSE`
- `upper` is set to `Inf`
- `upper_incbounds` is set to `FALSE`

The field range is printed in number interval denotation.

Returns: A list of ranges.

Examples:

```
\dontrun{
# get all field value ranges
surf$field_range()

# get value ranges for specific fields
surf$field_range(c("name", "number of vertices", "vertex 10 z-coordinate"))
}
```

Method `field_relation()`**:** Extract the relationship among fields

Usage:

```
IddObject$field_relation(
  which = NULL,
  direction = c("all", "ref_by", "ref_to"),
  class = NULL,
  group = NULL,
  depth = 0L,
  keep = FALSE
)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

`direction` The relation direction to extract. Should be one of `"all"`, `"ref_to"` or `"ref_by"`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

`keep` If `TRUE`, all input fields are returned regardless they have any relations with other objects or not. If `FALSE`, only fields in input that have relations with other objects are returned. Default: `FALSE`.

Details: Many fields in Idd can be referred by others. For example, the Outside Layer and other fields in `Construction` class refer to the `Name` field in `Material` class and other material related classes. Here it means that the Outside Layer field **refers to** the `Name` field and the `Name` field is **referred by** the Outside Layer.
$field_relation() provides a simple interface to get this kind of relation. It takes a field specification and a relation direction, and returns an `IddRelation` object which contains data presenting such relation above.
$field_relation() returns a list of references for those fields that have the `object-list` and/or `reference` and `reference-class-name` attribute. Basically, it is a list of two elements `ref_to` and `ref_by`. Underneath, `ref_to` and `ref_by` are data.tables which contain source field data and reference field data with custom printing method. For instance, if `iddobj$field_relation(c(1,2),"ref_to")` gives results below:

```
-- Refer to Others --------------------
  +- Field: <1: Field 1>
  |  v~~~~~~~~~~~~~~~~~~
  |  \- Class: <Class 2>
  |     \- Field: <2: Field 2>
  |
  \- Field: <2: Field 2>
```

This means that Field 2 in current class does not refer to any other fields. But Field 1 in current class refers to Field 2 in class named Class 2.

Returns: An `IddRelation` object.

Examples:

```
\dontrun{
# get field relation for specific fields
surf$field_relation(c("name", "zone name", "vertex 10 z-coordinate"))
```

```
}
```

Method `field_possible()`**:** Get field possible values

Usage:

```
IddObject$field_possible(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $field_possible() returns all possible values for specified fields, including auto-value (`Autosize`, `Autocalculate`, and `NA` if not applicable), and results from $field_default(), $field_range(), $field_choice(). Underneath, it returns a data.table with custom printing method. For instance, if `iddobj$field_possible(c(4,2))` gives results below:

```
-- 4: Field 4 ----------
* Auto value: <NA>
* Default: <NA>
* Choice:
  - "Key1"
  - "Key2"

-- 2: Field 2 ----------
* Auto value: "Autosize"
* Default: 2
* Choice: <NA>
```

This means that Field 4 in current class cannot be "autosized" or "autocalculated", and it does not have any default value. Its value should be a choice from `"Key1"` or `"Key2"`. For Field 2 in current class, it has a default value of `2` but can also be filled with value `"Autosize"`.

Returns: A `IddFieldPossible` object which is a `data.table::data.table()` with 9 columns.

Examples:

```
\dontrun{
# get field possible values for specific fields
surf$field_possible(6:10)
}
```

Method `is_valid_field_num()`**:** Check if input is a valid field number

Usage:

```
IddObject$is_valid_field_num(num)
```

Arguments:

`num` An integer vector to test.

Details: $is_valid_field_num() returns `TRUE` if input `num` is acceptable as a total number of fields in this class. Extensible property is considered.
For instance, the total number of fields defined in IDD for class `BuildingSurfaces:Detailed` is 390. However, 396 is still a valid field number for this class as the number of field in the extensible group is 3.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_valid_field_num(c(10, 14, 100))
}
```

Method `is_extensible_index()`**:** Check if input field index indicates an extensible field

Usage:

```
IddObject$is_extensible_index(index)
```

Arguments:

`index` An integer vector of field indices.

Details: $is_extensible_index() returns `TRUE` if input `index` indicates an index of extensible field in current class.
Extensible fields mean that these fields can be dynamically added or deleted, such like the X, Y and Z vertices of a building surface.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_extensible_index(c(10, 14, 100))
}
```

Method `is_valid_field_name()`**:** Check if input character is a valid field name

Usage:

```
IddObject$is_valid_field_name(name, strict = FALSE)
```

Arguments:

`name` A character vector to test.

`strict` If `TRUE`, only exact match is accepted. Default: `FALSE`.

Details: $is_valid_field_name() returns `TRUE` if `name` is a valid field name **WITHOUT** unit. Note `name` can be given in underscore style, e.g. `"outside_layer"` is equivalent to `"Outside Layer"`.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_valid_field_name(c("name", "sun_exposure"))

# exact match
surf$is_valid_field_name(c("Name", "Sun_Exposure"), strict = TRUE)
}
```

Method `is_valid_field_index()`**:** Check if input integer is a valid field index

Usage:

```
IddObject$is_valid_field_index(index)
```

Arguments:

`index` An integer vector to test.

Details: $is_valid_field_index() returns `TRUE` if `index` is a valid field index. For extensible class, `TRUE` is always returned.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_valid_field_index(1:10)
}
```

Method `is_autosizable_field()`**:** Check if input field can be autosized

Usage:

```
IddObject$is_autosizable_field(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $is_autosizable_field() returns `TRUE` if input field can be assigned to `autosize`.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_autosizable_field()

surf$is_autosizable_field(c("name", "sun_exposure"))
}
```

Method `is_autocalculatable_field()`**:** Check if input field can be autocalculated

Usage:

```
IddObject$is_autocalculatable_field(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $is_autocalculatable_field() returns `TRUE` if input field can be assigned to `autocalculate`.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_autocalculatable_field()

surf$is_autocalculatable_field(c("name", "sun_exposure"))
}
```

Method `is_numeric_field()`**:** Check if input field value should be numeric

Usage:

```
IddObject$is_numeric_field(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $is_numeric_field() returns `TRUE` if the value of input field should be numeric (an integer or a real number).

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_numeric_field()

surf$is_numeric_field(c("name", "sun_exposure"))
}
```

Method `is_real_field()`**:** Check if input field value should be a real number

Usage:

```
IddObject$is_real_field(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $is_real_field() returns `TRUE` if the field value should be a real number but not an integer.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_real_field()

surf$is_real_field(c("name", "number of vertices"))
}
```

Method `is_integer_field()`**:** Check if input field value should be an integer

Usage:

```
IddObject$is_integer_field(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $is_real_field() returns `TRUE` if the field value should be an integer.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_integer_field()

surf$is_integer_field(c("name", "number of vertices"))
}
```

Method `is_required_field()`**:** Check if input field is required

Usage:

```
IddObject$is_required_field(which = NULL)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

Details: $is_required_field() returns `TRUE` if the field is required.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$is_required_field()

surf$is_required_field(c("name", "number of vertices"))
}
```

Method `has_ref()`**:** Check if input field can refer to or can be referred by other fields

Usage:

```
IddObject$has_ref(which = NULL, class = NULL, group = NULL, depth = 0L)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

Details: $has_ref() returns `TRUE` if input field refers to or can be referred by other fields.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$has_ref()

surf$has_ref(c("name", "zone name"))
}
```

Method `has_ref_to()`**:** Check if input field can refer to other fields

Usage:

```
IddObject$has_ref_to(which = NULL, class = NULL, group = NULL, depth = 0L)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

Details: $has_ref_to() returns `TRUE` if input field can refer to other fields.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$has_ref_to()

surf$has_ref_to(c("name", "zone name"))
}
```

Method `has_ref_by()`**:** Check if input field can be referred by other fields

Usage:

```
IddObject$has_ref_by(which = NULL, class = NULL, group = NULL, depth = 0L)
```

Arguments:

`which` An integer vector of field indices or a character vector of field names in current class. If `NULL`, all fields in this class are used. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

Details: $has_ref_by() returns `TRUE` if input field can be referred by other fields.

Returns: A logical vector.

Examples:

```
\dontrun{
surf$has_ref_by()

surf$has_ref_by(c("name", "zone name"))
}
```

Method `to_table()`**:** Format an `IddObject` as a data.frame

Usage:

```
IddObject$to_table(all = FALSE)
```

Arguments:

`all` If `TRUE`, all available fields defined in IDD for specified class will be returned. If `FALSE`, only the minimum field number is returned. Default: `FALSE`.

Details: $to_table() returns a data.table that contains basic data of current class. The returned data.table has 3 columns:

- `class`: Character type. Current class name.
- `index`: Integer type. Field indexes.
- `field`: Character type. Field names.

Returns: A data.table with 3 columns.

Examples:

```
\dontrun{
surf$to_table()

surf$to_table(TRUE)
}
```

Method `to_string()`**:** Format an `IdfObject` as a character vector

Usage:

```
IddObject$to_string(comment = NULL, leading = 4L, sep_at = 29L, all = FALSE)
```

Arguments:

`comment` A character vector to be used as comments of returned string format object.

`leading` Leading spaces added to each field. Default: `4L`.

`sep_at` The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.

`all` If `TRUE`, all available fields defined in IDD for specified class will be returned. Default: `FALSE`.

Details: $to_string() returns the text format of current class. The returned character vector can be pasted into an IDF file as an empty object of specified class.

Returns: A character vector.

Examples:

```
\dontrun{
# get text format of class BuildingSurface:Detailed
surf$to_string()

# tweak output formatting
surf$to_string(leading = 0, sep_at = 0)

# add comments
surf$to_string(c("This", "will", "be", "comments"))
}
```

Method `print()`**:** Print `IddObject` object

Usage:

```
IddObject$print(brief = FALSE)
```

Arguments:

`brief` If `TRUE`, only class name part is printed. Default: `FALSE`.

Details: $print() prints the `IddObject` object giving the information of class name, class properties, field indices and field names.
$print() prints the IddObject. Basically, the print output can be divided into 4 parts:

- CLASS: IDD class name of current object in format <IddObject: CLASS>.
- MEMO: brief description of the IDD class.
- PROPERTY: properties of the IDD class, including name of group it belongs to, whether it is an unique or required class and current total fields. The fields may increase if the IDD class is extensible, such as `Branch`, `ZoneList` and etc.
- FIELDS: fields of current IDD class. Required fields are marked with stars (`*`). If the class is extensible, only the first extensible group will be printed and two ellipses will be shown at the bottom. Fields in the extensible group will be marked with an arrow down surrounded by angle brackets (<v>).

Returns: The `IddObject` object itself, invisibly.

Examples:

```
\dontrun{
surf

surf$print(brief = TRUE)
}
```

Note

Some classes have special format when saved in the IDFEditor with the special format option enabled. Those special format includes "singleLine", "vertices", "compactSchedule", "fluidProperties", "viewFactors" and "spectral". eplusr can handle all those format when parsing IDF files. However, when saved, all classes are formatted in standard way.

This number may change if the class is extensible and after $add_extensible_group() or $del_extensible_group().

The type of each default value will be consistent with field definition. However, for numeric fields with default values being `"autosize"` or `"autocalculate"`, the type of returned values will be character.

Author(s)

Hongyuan Jia

See Also

Idd Class

Examples

```
## ------------------------------------------------
## Method `IddObject$new`
## ------------------------------------------------

## Not run:
surf <- IddObject$new("BuildingSurface:Detailed", use_idd(8.8, download = "auto"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$version`
## ------------------------------------------------

## Not run:
# get version
surf$version()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$parent`
## ------------------------------------------------

## Not run:
surf$parent()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$group_name`
## ------------------------------------------------

## Not run:
surf$group_name()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$group_index`
## ------------------------------------------------

## Not run:
surf$group_index()

## End(Not run)
```

```
## ------------------------------------------------
## Method `IddObject$class_name`
## ------------------------------------------------

## Not run:
surf$class_name()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$class_index`
## ------------------------------------------------

## Not run:
surf$class_index()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$class_format`
## ------------------------------------------------

## Not run:
surf$class_format()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$min_fields`
## ------------------------------------------------

## Not run:
surf$min_fields()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$num_fields`
## ------------------------------------------------

## Not run:
surf$num_fields()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$memo`
```

```
## ------------------------------------------------

## Not run:
surf$memo()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$num_extensible`
## ------------------------------------------------

## Not run:
surf$num_extensible()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$first_extensible_index`
## ------------------------------------------------

## Not run:
surf$first_extensible_index()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$extensible_group_num`
## ------------------------------------------------

## Not run:
surf$extensible_group_num()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$add_extensible_group`
## ------------------------------------------------

## Not run:
# field number before adding
surf$num_fields()
# extensible group number before adding
surf$extensible_group_num()

# add 2 more extensible groups
surf$add_extensible_group(2)

# field number after adding
surf$num_fields()
```

```
# extensible group number after adding
surf$extensible_group_num()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$del_extensible_group`
## ------------------------------------------------

## Not run:
# field number before deleting
surf$num_fields()
# extensible group number before deleting
surf$extensible_group_num()

# delete 2 more extensible groups
surf$del_extensible_group(2)

# field number after deleting
surf$num_fields()
# extensible group number after deleting
surf$extensible_group_num()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$has_name`
## ------------------------------------------------

## Not run:
surf$has_name()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_required`
## ------------------------------------------------

## Not run:
surf$is_required()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_unique`
## ------------------------------------------------

## Not run:
surf$is_unique()
```

```
## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_extensible`
## ------------------------------------------------

## Not run:
surf$is_extensible()

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_name`
## ------------------------------------------------

## Not run:
# get all field names
surf$field_name()

# get field units also
surf$field_name(unit = TRUE)

# get field units in IP
surf$field_name(unit = TRUE)

# change field name to lower-style
surf$field_name(unit = TRUE, in_ip = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_index`
## ------------------------------------------------

## Not run:
# get all field indices
surf$field_index()

# get field indices for specific fields
surf$field_index(c("number of vertices", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_type`
## ------------------------------------------------

## Not run:
```

```
# get all field types
surf$field_type()

# get field types for specific fields
surf$field_type(c("name", "zone name", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_note`
## ------------------------------------------------

## Not run:
# get all field notes
surf$field_note()

# get field types for specific fields
surf$field_note(c("name", "zone name", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_unit`
## ------------------------------------------------

## Not run:
# get all field units
surf$field_unit()

# get field units for specific fields
surf$field_unit(c("name", "zone name", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_default`
## ------------------------------------------------

## Not run:
# get all field default values
surf$field_default()

# get default values for specific fields
surf$field_default(c("name", "zone name", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_choice`
```

```
## ------------------------------------------------

## Not run:
# get all field value choices
surf$field_choice()

# get field value choices for specific fields
surf$field_choice(c("name", "sun exposure", "wind exposure"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_range`
## ------------------------------------------------

## Not run:
# get all field value ranges
surf$field_range()

# get value ranges for specific fields
surf$field_range(c("name", "number of vertices", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_relation`
## ------------------------------------------------

## Not run:
# get field relation for specific fields
surf$field_relation(c("name", "zone name", "vertex 10 z-coordinate"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$field_possible`
## ------------------------------------------------

## Not run:
# get field possible values for specific fields
surf$field_possible(6:10)

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_valid_field_num`
## ------------------------------------------------

## Not run:
```

```
surf$is_valid_field_num(c(10, 14, 100))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_extensible_index`
## ------------------------------------------------

## Not run:
surf$is_extensible_index(c(10, 14, 100))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_valid_field_name`
## ------------------------------------------------

## Not run:
surf$is_valid_field_name(c("name", "sun_exposure"))

# exact match
surf$is_valid_field_name(c("Name", "Sun_Exposure"), strict = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_valid_field_index`
## ------------------------------------------------

## Not run:
surf$is_valid_field_index(1:10)

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_autosizable_field`
## ------------------------------------------------

## Not run:
surf$is_autosizable_field()

surf$is_autosizable_field(c("name", "sun_exposure"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_autocalculatable_field`
## ------------------------------------------------
```

```
## Not run:
surf$is_autocalculatable_field()

surf$is_autocalculatable_field(c("name", "sun_exposure"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_numeric_field`
## ------------------------------------------------

## Not run:
surf$is_numeric_field()

surf$is_numeric_field(c("name", "sun_exposure"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_real_field`
## ------------------------------------------------

## Not run:
surf$is_real_field()

surf$is_real_field(c("name", "number of vertices"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_integer_field`
## ------------------------------------------------

## Not run:
surf$is_integer_field()

surf$is_integer_field(c("name", "number of vertices"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$is_required_field`
## ------------------------------------------------

## Not run:
surf$is_required_field()

surf$is_required_field(c("name", "number of vertices"))
```

```
## End(Not run)


## ------------------------------------------------
## Method `IddObject$has_ref`
## ------------------------------------------------

## Not run:
surf$has_ref()

surf$has_ref(c("name", "zone name"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$has_ref_to`
## ------------------------------------------------

## Not run:
surf$has_ref_to()

surf$has_ref_to(c("name", "zone name"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$has_ref_by`
## ------------------------------------------------

## Not run:
surf$has_ref_by()

surf$has_ref_by(c("name", "zone name"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$to_table`
## ------------------------------------------------

## Not run:
surf$to_table()

surf$to_table(TRUE)

## End(Not run)


## ------------------------------------------------
```

```
## Method `IddObject$to_string`
## ------------------------------------------------

## Not run:
# get text format of class BuildingSurface:Detailed
surf$to_string()

# tweak output formatting
surf$to_string(leading = 0, sep_at = 0)

# add comments
surf$to_string(c("This", "will", "be", "comments"))

## End(Not run)


## ------------------------------------------------
## Method `IddObject$print`
## ------------------------------------------------

## Not run:
surf

surf$print(brief = TRUE)

## End(Not run)
```

`idd_object` *Create an* `IddObject` *object.*

Description

`idd_object()` takes a parent `Idd` object, a class name, and returns a corresponding IddObject. For details, see IddObject.

Usage

```
idd_object(parent, class)
```

Arguments

`parent`	An Idd object or a valid input for `use_idd()`.
`class`	A valid class name (a string).

Value

An IddObject object.

Examples

```
## Not run:
idd <- use_idd(8.8, download = "auto")

# get an IddObject using class name
idd_object(idd, "Material")
idd_object(8.8, "Material")

## End(Not run)
```

`Idf` *Read, Modify, and Run an EnergyPlus Model*

Description

eplusr provides parsing EnergyPlus Input Data File (IDF) files and strings in a hierarchical structure, which was extremely inspired by OpenStudio utilities library, but with total different data structure under the hook.

Details

eplusr uses `Idf` class to present the whole IDF file and use IdfObject to present a single object in IDF. Both `Idf` and IdfObject contain member functions for helping modify the data in IDF so it complies with the underlying IDD (EnergyPlus Input Data Dictionary).

Under the hook, eplusr uses a SQL-like structure to store both IDF and IDD data in different data.table::data.tables. So to modify an EnergyPlus model in eplusr is equal to change the data in those IDF tables accordingly, in the context of specific IDD data. This means that a corresponding Idd object is needed whenever creating an `Idf` object. eplusr provides several helpers to easily download IDD files and create Idd objects.

All IDF reading process starts with function `read_idf()` which returns an `Idf` object. `Idf` class provides lots of methods to programmatically query and modify EnergyPlus models.

Internally, the powerful data.table package is used to speed up the whole IDF parsing process and store the results. Under the hook, eplusr uses a SQL-like structure to store both IDF and IDD data in data.table::data.table format. Every IDF will be parsed and stored in three tables:

- `object`: contains object IDs, names and comments.
- `value`: contains field values
- `reference`: contains cross-reference data of field values.

Methods

Public methods:

- `Idf$new()`
- `Idf$version()`

- Idf$path()
- Idf$group_name()
- Idf$class_name()
- Idf$is_valid_group()
- Idf$is_valid_class()
- Idf$definition()
- Idf$object_id()
- Idf$object_name()
- Idf$object_num()
- Idf$is_valid_id()
- Idf$is_valid_name()
- Idf$object()
- Idf$objects()
- Idf$object_unique()
- Idf$objects_in_class()
- Idf$objects_in_group()
- Idf$object_relation()
- Idf$objects_in_relation()
- Idf$search_object()
- Idf$dup()
- Idf$add()
- Idf$set()
- Idf$del()
- Idf$purge()
- Idf$duplicated()
- Idf$unique()
- Idf$rename()
- Idf$insert()
- Idf$load()
- Idf$update()
- Idf$paste()
- Idf$search_value()
- Idf$replace_value()
- Idf$validate()
- Idf$is_valid()
- Idf$to_string()
- Idf$to_table()
- Idf$is_unsaved()
- Idf$save()
- Idf$run()
- Idf$last_job()
- Idf$geometry()

- Idf$view()
- Idf$print()
- Idf$clone()

Method `new()`**:** Create an `Idf` object

Usage:

```
Idf$new(path, idd = NULL)
```

Arguments:

`path` Either a path, a connection, or literal data (either a single string or a raw vector) to an EnergyPlus Input Data File (IDF). If a file path, that file usually has a extension `.idf`.

`idd` Any acceptable input of `use_idd()`. If `NULL`, which is the default, the version of IDF will be passed to `use_idd()`. If the input is an `.ddy` file which does not have a version field, the latest version of Idf cached will be used.

Details: It takes an EnergyPlus Input Data File (IDF) as input and returns an `Idf` object. Currently, Imf file is not fully supported. All EpMacro lines will be treated as normal comments of the nearest downwards object. If input is an Imf file, a warning will be given during parsing. It is recommended to convert the Imf file to an Idf file and use ParametricJob class to conduct parametric analysis.

Returns: An `Idf` object.

Examples:

```
\dontrun{
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# If neither EnergyPlus v8.8 nor Idd v8.8 was found, error will
# occur. If Idd v8.8 is found, it will be used automatically.
idf <- Idf$new(path_idf)

# argument `idd` can be specified explicitly using `use_idd()`
idf <- Idf$new(path_idf, idd = use_idd(8.8))

# you can set `download` arugment to "auto" in `use_idd()` if you
# want to automatically download corresponding IDD file when
# necessary
idf <- Idf$new(path_idf, use_idd(8.8, download = "auto"))

# Besides use a path to an IDF file, you can also provide IDF in literal
# string format
string_idf <-
    "
    Version, 8.8;
    Building,
        Building;                !- Name
    "

Idf$new(string_idf, use_idd(8.8, download = "auto"))
```

```
}
```

Method `version()`**:** Get the version of current `Idf`

Usage:

```
Idf$version()
```

Details: $version() returns the version of current `Idf` in a `base::numeric_version()` format. This makes it easy to direction compare versions of different `Idf`s, e.g. `idf$version() > 8.6` or `idf1$version() > idf2$version()`.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
# get version
idf$version()
}
```

Method `path()`**:** Get the file path of current `Idf`

Usage:

```
Idf$path()
```

Details: $path() returns the full path of current `Idf` or `NULL` if the `Idf` object is created using a character vector and not saved locally.

Returns: `NULL` or a single string.

Examples:

```
\dontrun{
# get path
idf$path()

# return `NULL` if Idf is not created from a file
Idf$new("Version, 8.8;\n")$path()
}
```

Method `group_name()`**:** Get names of groups

Usage:

```
Idf$group_name(all = FALSE, sorted = TRUE)
```

Arguments:

`all` If `FALSE`, only names of groups in current `Idf` object will be returned. If `TRUE`, all group names in the underlying Idd will be returned. Default: `FALSE`.

`sorted` Only applicable when `all` is `FALSE`. If `TRUE`, duplications in returned group or class names are removed, and unique names are further sorted according to their occurrences in the underlying Idd. Default: `TRUE`.

Details: $group_name() returns names of groups current `Idf` contains or the underlying Idd object contains.

Returns: A character vector.

Examples:

```
\dontrun{
# get names of all groups Idf contains
idf$group_name()

# get group name of each object in Idf
idf$group_name(sorted = FALSE)

# get names of all available groups in underlying Idd
idf$group_name(all = TRUE)
}
```

Method `class_name()`**:** Get names of classes

Usage:

```
Idf$class_name(all = FALSE, sorted = TRUE, by_group = FALSE)
```

Arguments:

`all` If `FALSE`, only names of classes in current `Idf` object will be returned. If `TRUE`, all class names in the underlying Idd will be returned. Default: `FALSE`.

`sorted` Only applicable when `all` is `FALSE`. If `TRUE`, duplications in returned group or class names are removed, and unique names are further sorted according to their occurrences in the underlying Idd. Default: `TRUE`.

`by_group` Only applicable when `all` or `sorted` is `TRUE`. If `TRUE`, a list is returned which separates class names by the group they belong to.

Details: $class_name() returns names of classes current `Idf` contains or the underlying Idd object contains.

Returns: A character vector if `by_group` is `FALSE` and a list of character vectors when `by_group` is `TRUE`.

Examples:

```
\dontrun{
# get names of all classes in Idf
idf$class_name()

# get names of all classes grouped by group names in Idf
idf$class_name(by_group = TRUE)

# get class name of each object in Idf
idf$class_name(sorted = FALSE)

# get names of all available classes in underlying Idd
idf$class_name(all = TRUE)

# get names of all available classes grouped by group names in
# underlying Idd
```

```
idf$class_name(all = TRUE, by_group = TRUE)
}
```

Method `is_valid_group()`**:** Check if elements in input character vector are valid group names.

Usage:

```
Idf$is_valid_group(group, all = FALSE)
```

Arguments:

`group` A character vector to check.

`all` If `FALSE`, check if input characters are valid group names for current `Idf`. If `TRUE`, check if input characters are valid group names for underlying Idd. Default: FALSE

Details: $is_valid_group() returns `TRUE`s if given character vector contains valid group names in the context of current `Idf` (when `all` is `FALSE`) or current underlying Idd (when `all` is `TRUE`). Note that case-sensitive matching is performed, which means that `"Location and Climate"` is a valid group name but `"location and climate"` is not.

Returns: A logical vector with the same length as input character vector.

Examples:

```
\dontrun{
# check if input is a valid group name in current Idf
idf$is_valid_group(c("Schedules", "Compliance Objects"))

# check if input is a valid group name in underlying Idd
idf$is_valid_group(c("Schedules", "Compliance Objects"), all = TRUE)
}
```

Method `is_valid_class()`**:** Check if elements in input character vector are valid class names.

Usage:

```
Idf$is_valid_class(class, all = FALSE)
```

Arguments:

`class` A character vector to check.

`all` If `FALSE`, check if input characters are valid class names for current `Idf`. If `TRUE`, check if input characters are valid class names for underlying Idd. Default: FALSE

Details: $is_valid_class() returns `TRUE`s if given character vector contains valid class names in the context of current `Idf` (when `all` is `FALSE`) or current underlying Idd (when `all` is `TRUE`), and `FALSE`s otherwise.
Note that case-sensitive matching is performed, which means that `"Version"` is a valid class name but `"version"` is not.

Returns: A logical vector with the same length as input character vector.

Examples:

```
\dontrun{
# check if input is a valid class name in current Idf
idf$is_valid_class(c("Building", "ShadowCalculation"))
```

```
# check if input is a valid class name in underlying Idd
idf$is_valid_class(c("Building", "ShadowCalculation"), all = TRUE)
}
```

Method `definition()`**:** Get the IddObject object for specified class.

Usage:

```
Idf$definition(class = NULL)
```

Arguments:

`class` A **single** string of valid class name in current Idd. If `NULL`, the underlying Idd object is returned. Default: `NULL`.

Details: $definition() returns an IddObject of given class. IddObject contains all data used for parsing and creating an IdfObject. For details, please see IddObject class.

Returns: An IddObject object if class is not `NULL` or an Idd object if class is `NULL`.

Examples:

```
\dontrun{
# get the IddObject object for specified class
idf$definition("Version")
}
```

Method `object_id()`**:** Get the unique ID for each object in specified classes in the `Idf`.

Usage:

```
Idf$object_id(class = NULL, simplify = FALSE)
```

Arguments:

`class` A character vector that contains valid class names for current `Idf` object. If `NULL`, all classes in current `Idf` object are used. Default: `NULL`.

`simplify` If `TRUE`, an integer vector contains object IDs of all specified classes is returned. If `FALSE`, a named list that contains object IDs for each specified class is returned. Default: `FALSE`.

Details: In `Idf`, each object is assigned with an integer as an universally unique identifier (UUID) in the context of current `Idf`. UUID is not reused even if the object associated is deleted.
$object_id() returns an integer vector (when `simplify` is `TRUE`) or a named list (when `simplify` is `FALSE`) of integer vectors that contain object IDs in each specified class. The returned list is named using specified class names.

Returns: An integer vector (when `simplify` is `TRUE`) or a named list of integer vectors (when `simplify` is `FALSE`).

Examples:

```
\dontrun{
# get IDs of all objects in current Idf object
idf$object_id()

# get IDs of all objects in current Idf object, and merge them into a
```

```
# single integer vector
idf$object_id(simplify = TRUE)

# get IDs of objects in class Version and Zone
idf$object_id(c("Version", "Zone"))

# get IDs of objects in class Version and Zone, and merge them into a
# single integer vector
idf$object_id(c("Version", "Zone"), simplify = TRUE)
}
```

Method `object_name()`**:** Get names for objects in specified classes in the `Idf`.

Usage:

```
Idf$object_name(class = NULL, simplify = FALSE)
```

Arguments:

`class` A character vector that contains valid class names for current `Idf`. If `NULL`, all classes in current `Idf` are used. Default: `NULL`.

`simplify` If `TRUE`, a character vector contains object names of all specified classes is returned. If `FALSE`, a named list that contains a character vector for each specified class is returned. Default: `FALSE`.

Details: In `Idf`, each object is assigned with a single string as the name for it, if the class it belongs to has name attribute, e.g. class `RunPeriod`, `Material` and etc. That name should be unique among all objects in that class. EnergyPlus will fail with an error if duplications are found among object names in a class.
$object_name() returns a character vector (when `simplify` is `TRUE`) or a named list (when `simplify` is `FALSE`) of character vectors that contain object IDs in each specified class. The returned list is named using specified class names. If specified class does not have name attribute, `NA`s are returned.

Returns: A character vector (when `simplify` is `TRUE`) or a named list of character vectors (when `simplify` is `FALSE`).

Examples:

```
\dontrun{
# get names of all objects in current Idf object
idf$object_name()

# get names of all objects in current Idf object, and merge them into
# a single character vector
idf$object_name(simplify = TRUE)

# get names of objects in class Version and Zone
idf$object_name(c("Version", "Zone"))

# get names of objects in class Version and Zone, and merge them into
# a single character vector
idf$object_name(c("Version", "Zone"), simplify = TRUE)
```

```
}
```

Method `object_num()`**:** Get number of objects in specified classes in the Idf object.

Usage:

```
Idf$object_num(class = NULL)
```

Arguments:

`class` A character vector that contains valid class names for underlying Idd. If `NULL`, all classes in current `Idf` are used, and the total object number is returned. Default: `NULL`.

Details: $object_num() returns an integer vector of object number in specified classes. `0` is returned if there is no object in that class.

Returns: An integer vector.

Examples:

```
\dontrun{
# get total number of objects
idf$object_num()

# get number of objects in class Zone and Schedule:Compact
idf$object_num(c("Zone", "Schedule:Compact"))
}
```

Method `is_valid_id()`**:** Check if elements in input integer vector are valid object IDs.

Usage:

```
Idf$is_valid_id(id)
```

Arguments:

`id` An integer vector to check.

Details: $is_valid_id() returns `TRUE`s if given integer vector contains valid object IDs in current `Idf` object.

Returns: A logical vector with the same length as input integer vector.

Examples:

```
\dontrun{
idf$is_valid_id(c(51, 1000))
}
```

Method `is_valid_name()`**:** Check if elements in input character vector are valid object names.

Usage:

```
Idf$is_valid_name(name)
```

Arguments:

`name` A character vector to check.

Details: $is_valid_name() returns TRUEs if given character vector contains valid object names in current Idf object.
Note that **case-insensitive** matching is performed, which means that "rOoF" is equivalent to "roof". This behavior is consistent in all methods that take object name(s) as input.

Returns: A logical vector with the same length as input character vector.

Examples:

```
\dontrun{
idf$is_valid_name(c("Simple One Zone (Wireframe DXF)", "ZONE ONE", "a"))

# name matching is case-insensitive
idf$is_valid_name(c("simple one zone (wireframe dxf)", "zone one", "a"))
}
```

Method `object()`**:** Extract an IdfObject object using object ID or name.

Usage:

```
Idf$object(which, class = NULL)
```

Arguments:

`which` A single integer specifying the object ID or a single string specifying the object name.

`class` A character vector that contains valid class names for current Idf object used to locate objects. If NULL, all classes in current Idf object are used. Default: NULL.

Details: $object() returns an IdfObject object specified by an object ID or name.
Note that unlike object ID, which is always unique across the whole Idf object, different objects can have the same name. If the name given matches multiple objects, an error is issued showing what objects are matched by the same name. This behavior is consistent in all methods that take object name(s) as input. In this case, it is suggested to directly use object ID instead of name.
Note that **case-insensitive** matching is performed for object names, which means that "rOoF" is equivalent to "roof". This behavior is consistent in all methods that take object name(s) as input.

Returns: An IdfObject object.

Examples:

```
\dontrun{
# get an object whose ID is 3
idf$object(3)

# get an object whose name is "simple one zone (wireframe dxf)"
# NOTE: object name matching is case-insensitive
idf$object("simple one zone (wireframe dxf)")
}
```

Method `objects()`**:** Extract multiple IdfObject objects using object IDs or names.

Usage:

```
Idf$objects(which)
```

Arguments:

`which` An integer vector specifying object IDs or a character vector specifying object names.

Details: $objects() returns a named list of IdfObject objects using object IDS or names. The returned list is named using object names.
Note that unlike object ID, which is always unique across the whole `Idf` object, different objects can have the same name. If the name given matches multiple objects, an error is issued showing what objects are matched by the same name. This behavior is consistent in all methods that take object name(s) as input. In this case, it is suggested to directly use object ID instead of name.
Note that **case-insensitive** matching is performed for object names, which means that `"rOoF"` is equivalent to `"roof"`. This behavior is consistent in all methods that take object name(s) as input.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# get objects whose IDs are 3 and 10
idf$objects(c(3,10))

# get objects whose names are "Simple One Zone (Wireframe DXF)" and "ZONE ONE"
# NOTE: object name matching is case-insensitive
idf$objects(c("Simple One Zone (Wireframe DXF)", "zone one"))
}
```

Method `object_unique()`**:** Extract the IdfObject in class with `unique-object` attribute.

Usage:

```
Idf$object_unique(class)
```

Arguments:

`class` A single string of valid class name for current `Idf` object.

Details: For each version of an `Idf` object, the corresponding underlying Idd describe how many objects can be defined in each class. Classes that have `unique-object` attribute can only hold a single object, e.g. `Version`, `SimulationControl` and etc. $object_unique() can be used to directly return the IdfObject in one `unique-object` class. An error will be issued if there are multiple objects in that class or input class is not an `unique-object` class. This makes sure that $object_unique() always returns a single IdfObject.
`Idf` class also provides custom S3 method of `$` and `[[` to make it more convenient to get the IdfObject in `unique-object` class. Basically, `idf$ClassName` and `idf[["ClassName"]]`, where `ClassName` is a single valid class name, is equivalent to `idf$object_unique(ClassName)` if `ClassName` is an `unique-object` class. For convenience, underscore-style names are allowed when using `$`, e.g. `Site_Location` is equivalent to `Site:Location`. For instance, `idf$Site_Location` and also `idf[["Site_Location"]]` will both return the IdfObjects in `Site:Location` class. Note that unlike $object_unique(), `idf$ClassName` and `idf[["ClassName"]]` will directly return `NULL` instead of giving an error when `ClassName` is not a valid class name in current `Idf` object. This makes it possible to use `is.null(idf$ClassName)` to check if `ClassName` is a valid class or not.

Returns: An IdfObject object.

Examples:

```
\dontrun{
# get the SimulationColtrol object
idf$object_unique("SimulationControl")

# S3 "[[" and "$" can also be used
idf$SimulationControl
idf[["SimulationControl"]]
}
```

Method `objects_in_class()`**:** Extract all IdfObject objects in one class.

Usage:

```
Idf$objects_in_class(class)
```

Arguments:

`class` A single string of valid class name for current `Idf` object.

Details: $objects_in_class() returns a named list of all IdfObject objects in specified class. The returned list is named using object names.
`Idf` class also provides custom S3 method of `$` and `[[` to make it more convenient to get all IdfObject objects in one class. Basically, `idf$ClassName` and `idf[["ClassName"]]`, where `ClassName` is a single valid class name, is equivalent to `idf$objects_in_class(ClassName)` if `ClassName` is not an `unique-object` class. For convenience, *underscore-style* names are allowed, e.g. `BuildingSurface_Detailed` is equivalent to `BuildingSurface:Detailed` when using `$`. For instance, `idf$BuildingSurface_Detailed` and also `idf[["BuildingSurface:Detailed"]]` will both return all IdfObject objects in `BuildingSurface:Detailed` class. Note that unlike $objects_in_class(), `idf$ClassName` and `idf[["ClassName"]]` will directly return `NULL` instead of giving an error when `ClassName` is not a valid class name in current `Idf` object. This makes it possible to use `is.null(idf$ClassName)` to check if `ClassName` is a valid class or not.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# get all objects in Zone class
idf$objects_in_class("Zone")

# S3 "[[" and "$" can also be used
idf$Zone
idf[["Zone"]]
}
```

Method `objects_in_group()`**:** Extract all IdfObject objects in one group.

Usage:

```
Idf$objects_in_group(group)
```

Arguments:

`group` A single string of valid group name for current `Idf` object.

Details: $objects_in_group() returns a named list of all IdfObject objects in specified group. The returned list is named using object names.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# get all objects in Schedules group
idf$objects_in_group("Schedules")
}
```

Method `object_relation()`**:** Extract the relationship between object field values.

Usage:

```
Idf$object_relation(
  which,
  direction = c("all", "ref_to", "ref_by", "node"),
  object = NULL,
  class = NULL,
  group = NULL,
  depth = 0L,
  keep = FALSE,
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` A single integer specifying object ID or a single string specifying object name.

`direction` The relation direction to extract. Should be either `"all"`, `"ref_to"`, `"ref_by"` and `"node"`.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

`keep` If `TRUE`, all fields of specified object are returned regardless they have any relations with other objects or not. If `FALSE`, only fields in specified object that have relations with other objects are returned. Default: `FALSE`.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in `Branch` objects. Their value refers to other many class names of objects, instaed of referring to specific field values. There are 3 options in total, i.e. `"none"`, `"both"` and `"all"`, with `"both"` being the default. * `"none"`: just ignore class-name-references. It is a reasonable option, as for most cases, class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * `"both"`: only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, only the object

that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * "all": include all class-name-references. For example, if the value of field Component 1 Object Type is Coil:Heating:Water, all objects in Coil:Heating:Water will be treated as referenced by that field. This is the most aggressive option.

Details: Many fields in Idd can be referred by others. For example, the Outside Layer and other fields in Construction class refer to the Name field in Material class and other material related classes. Here it means that the Outside Layer field **refers to** the Name field and the Name field is **referred by** the Outside Layer. In EnergyPlus, there is also a special type of field called Node, which together with Branch, BranchList and other classes define the topography of the HVAC connections. A outlet node of a component can be referred by another component as its inlet node, but can also exists independently, such as zone air node.
$object_relation() provides a simple interface to get this kind of relation. It takes a single object ID or name and also a relation direction, and returns an IdfRelation object which contains data presenting such relation above. For instance, if model$object_relation("WALL-1","ref_to") gives results below:

```
-- Refer to Others ------------------------
  Class: <Construction>
  \- Object [ID:2] <WALL-1>
     \- 2: "WD01";        !- Outside Layer
        v~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        \- Class: <Material>
           \- Object [ID:1] <WD01>
              \- 1: "WD01";        !- Name
```

This means that the value "WD01" of Outside Layer in a construction named WALL-1 refers to a material named WD01. All those objects can be further easily extracted using $objects_in_relation() method described below.

Returns: An IdfRelation object, which is a list of 3 data.table::data.table()s named ref_to, ref_by and node. Each data.table::data.table() contains 24 columns.

Examples:

```
\dontrun{
# check each layer's reference of a construction named FLOOR
idf$object_relation("floor", "ref_to")

# check where is this construction being used
idf$object_relation("floor", "ref_by")
}
```

Method `objects_in_relation()`**:** Extract multiple IdfObject objects referencing each others.

Usage:

```
Idf$objects_in_relation(
  which,
  direction = c("ref_to", "ref_by", "node"),
  object = NULL,
  class = NULL,
  group = NULL,
```

```
  depth = 0L,
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` A single integer specifying object ID or a single string specifying object name.

`direction` The relation direction to extract. Should be one of `"ref_to"`, `"ref_by"` or `"node"`.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of valid class names in the underlying Idd. It is used to restrict the classes to be returned. If `NULL`, all possible classes are considered and corresponding IdfObject objects are returned if relationships are found. Default: `NULL`.

`group` A character vector of valid group names in the underlying Idd. It is used to restrict the groups to be returned. If `NULL`, all possible groups are considered and corresponding IdfObject objects are returned if relationships are found. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in `Branch` objects. Their value refers to other many class names of objects, instaed of refering to specific field values. There are 3 options in total, i.e. `"none"`, `"both"` and `"all"`, with `"both"` being the default. * `"none"`: just ignore class-name-references. It is a reasonable option, as for most cases, class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * `"both"`: only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, only the object that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * `"all"`: include all class-name-references. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, all objects in `Coil:Heating:Water` will be treated as referenced by that field. This is the most aggressive option.

Details: $objects_in_relation() returns a named list of IdfObject objects that have specified relationship with given object. The first element of returned list is always the specified object itself. If that object does not have specified relationship with other objects in specified `class`, a list that only contains specified object itself is returned.

For instance, assuming that `const` is a valid object name in `Construction` class, `idf$objects_in_relation("const",'` will return a named list of an IdfObject object named `const` and also all other IdfObject objects in `BuildingSurface:Detailed` class that refer to field values in `const`. Similarly, `idf$objects_in_relation("const'` will return a named list of an IdfObject object named `const` and also all other IdfObject objects in `Material` class that `const` refers to. This makes it easy to directly extract groups of related objects and then use $insert() method or other methods described below to insert them or extract data.

There are lots of recursive references in a model. For instance, a material can be referred by a construction, that construction can be referred by a building surface, and that building surface can be referred by a window on that surface. These objects related recursively can be extracted by setting `recursive` to `TRUE`.

Returns: An named list of IdfObject objects.

Examples:

```
\dontrun{
# get a construction named FLOOR and all materials it uses
idf$objects_in_relation("floor", "ref_to")

# get a construction named FLOOR and all surfaces that uses it
idf$objects_in_relation("floor", "ref_by", class = "BuildingSurface:Detailed")
}
```

Method `search_object()`: Extract multiple IdfObject objects using regular expression on names.

Usage:

```
Idf$search_object(
  pattern,
  class = NULL,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments:

`pattern`, `ignore.case`, `perl`, `fixed`, `useBytes` All are directly passed to base::grepl.

`class` A character vector of valid class names in the underlying Idd. It is used to restrict the classes to be returned. If `NULL`, all possible classes are considered and corresponding IdfObject objects are returned if `pattern` is met Default: `NULL`.

Details: $search_object() returns a named list of IdfObject objects whose names meet the given regular expression in specified classes.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# get all objects whose names contains "floor"
idf$search_object("floor", ignore.case = TRUE)
}
```

Method `dup()`: Duplicate existing objects.

Usage:

```
Idf$dup(...)
```

Arguments:

`...` Integer vectors of object IDs and character vectors of object names. If input is named, its name will be used as the name of newly created objects.

Details: $dup() takes integer vectors of object IDs and character vectors of object names, duplicates objects specified, and returns a list of newly created IdfObject objects. The names of input are used as new names for created IdfObjects. If input is not named, new names are the

names of duplicated objects with a suffix "_1", "_2" and etc, depending on how many times that object has been duplicated. Note an error will be issued if trying to assign a new name to an object which belongs to a class that does not have name attribute.
Assigning newly added objects with an existing name in current Idf object is prohibited if current validation level includes object name conflicting checking. For details, please see level_checks().

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# duplicate an object named "FLOOR"
idf$dup("floor") # New object name 'FLOOR_1' is auto-generated

# duplicate that object again by specifing object ID
idf$dup(16) # New object name 'FLOOR_2' is auto-generated

# duplicate that object two times and giving new names
idf$dup(new_floor = "floor", new_floor2 = 16)

# duplicate that object multiple times using variable inputs
floors_1 <- c(new_floor3 = "floor", new_floor4 = "floor")
floors_2 <- setNames(rep(16, 5), paste0("flr", 1:5))
idf$dup(floors_1, floors_2)
}
```

Method `add()`**:** Add new objects.

Usage:

```
Idf$add(..., .default = TRUE, .all = FALSE)
```

Arguments:

`...` Lists of object definitions. Each list should be named with a valid class name. There is a special element `.comment` in each list, which will be used as the comments of newly added object.

`.default` If `TRUE`, default values are used for those blank fields if possible. If `FALSE`, empty fields are kept blank. Default: `TRUE`.

`.all` If `TRUE`, all fields are added. If `FALSE`, only minimum required fields are added. Default: `FALSE`.

Details: $add() takes new object definitions in list format, adds corresponding objects in specified classes, returns a list of newly added IdfObject objects. The returned list will be named using newly added object names. Every list should be named using a valid class name. Underscore-style class name is allowed for class name. Names in each list element are treated as field names. Values without names will be inserted according to their position. There is a special element named `.comment` in each list, which will be used as the comments of newly added object.
Empty objects can be added using an empty list, e.g. `idf$add(Building = list())`. All empty fields will be filled with corresponding default value if `.default` is `TRUE`, leaving other fields as blanks. However, adding blank objects may not be allowed if there are required fields in that

class and current validate level includes missing-required-field checking. For what kind of validation components will be performed during adding new objects, please see `level_checks()`.
Field name matching is **case-insensitive**. For convenience, underscore-style field names are also allowed, e.g. `eNd_MoNtH` is equivalent to End Month. This behavior is consistent among all methods that take field names as input.
There is no need to give all field values if only specific fields are interested, as long as other fields are not required. For example, to define a new object in `RunPeriod` class, the following is enough (at least for EnergyPlus v8.8):

```
idf$add(
    RunPeriod = list(
        "my run period",
        begin_month = 1, begin_day_of_month = 1,
        end_month = 1, end_day_of_month = 31
    ),
    .default = TRUE
)
```

If not all field names are given, positions of those values without field names are determined after those values with names. E.g. in `idf$add(Construction = list("out_layer",name = "name"))`, `"out_layer"` will be treated as the value for field Outside Layer in `Construction` class, since the value for field `Name` has been specified using explicit field name.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# add a new Building object with all default values
empty <- empty_idf(8.8) # create an empty Idf
empty$add(Building = list())

# add a new Building object with all default values and comments
empty <- empty_idf(8.8) # create an empty Idf
empty$add(Building = list(.comment = c("this is", "a new building")))

# add a new RunPeriod object with all possible fields
empty <- empty_idf(8.8) # create an empty Idf
empty$add(Building = list(), RunPeriod = list("rp", 1, 1, 1, 31), .all = TRUE)

# add objects using variable inputs
empty <- empty_idf(8.8) # create an empty Idf
objs1 <- list(Schedule_Constant = list("const"), Building = list())
rp <- list(RunPeriod = list("rp", 2, 1, 2, 28))
empty$add(objs1, rp)
}
```

Method `set()`**:** Set values of existing objects.

Usage:

```
Idf$set(..., .default = TRUE, .empty = FALSE)
```

Arguments:

`...` Lists of object definitions. Each list should be named with a valid object name or ID denoted in style `..ID`. There is a special element `.comment` in each list, which will be used as new comments of modified object, overwriting existing comments if any.

`.default` If `TRUE`, default values are used for those blank fields if possible. If `FALSE`, empty fields are kept blank. Default: `TRUE`.

`.empty` If `TRUE`, trailing empty fields are kept. Default: `FALSE`.

Details: $set() takes new field value definitions in list format, sets new values for fields in objects specified, and returns a list of modified IdfObjects. The returned list will be named using names of modified objects. Every list in $set() should be named with a valid object name. Object ID can also be used but have to be combined with prevailing two periods `..`, e.g. `..10` indicates the object with ID `10`. Similar to `$add()`, a special element `.comment` in each list will be used as the **new** comments for modified object, overwriting the old ones. Names in list element are treated as field names.

There is two special syntax in $set(), which is inspired by the data.table package:

- `class := list(field = value)`: Note the use of := instead of =. The main difference is that, unlike =, the left hand side of := should be a valid class name in current `Idf` object. It will set the field of all objects in specified class to specified value.
- `.(object,object) := list(field = value)`: Similar like above, but note the use of `.()` in the left hand side. You can put multiple object ID or names in `.()`. It will set the field of all specified objects to specified value.

You can delete a field by assigning `NULL` to it, e.g. `list(fld = NULL)` means to delete the value of field `fld`, in the condition that `.default` is `FALSE`, `fld` is not a required field and the index of `fld` is larger than the number minimum fields required for that class. If those conditions are not required, `fld` will be left as blank if `.default` is `FALSE` or filled with default value if `.default` is `TRUE`.

By default, trailing empty fields that are not required will be removed and only minimum required fields are kept. For example, if `rp` is an object in `RunPeriod` class in an `Idf` of version 8.8, by default empty field with index larger than 11 will be removed since they are all non-required fields. You can keep the trailing empty fields by setting `.empty` to `TRUE`.

New fields that currently do not exist in that object can also be set. They will be automatically added on the fly.

Field name matching is **case-insensitive**. For convenience, underscore-style field names are also allowed, e.g. `eNd_MoNtH` is equivalent to End Month.

If not all field names are given, positions of those values without field names are determined after those values with names. E.g. in `idf$set(floor = list("out_layer",name = "name"))`, `"out_layer"` will be treated as the value for field Outside Layer in an object named `floor`, since the value for field `Name` has been specified using explicit field name.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# modify an object by name (case-insensitive)
idf$set(r13layer = list(roughness = "smooth"))

# modify an object by ID
idf$set(..12 = list(roughness = "rough"))

# overwrite existing object comments
```

```
idf$set(r13layer = list(.comment = c("New comment")))

# assign default values to fields
idf$set(r13layer = list(solar_absorptance = NULL), .default = TRUE)

# set field values to blanks
idf$set(r13layer = list(solar_absorptance = NULL), .default = FALSE)

# set field values to blank and delete trailing fields
idf$set(r13layer = list(visible_absorptance = NULL), .default = FALSE)

# set field values to blank and keep blank fields
idf$set(r13layer = list(visible_absorptance = NULL), .default = FALSE, .empty = TRUE)

# set all fields in one class
idf$set(Material_NoMass := list(visible_absorptance = 0.9))

# set multiple objects in one class
idf$set(.("r13layer", "r31layer") := list(solar_absorptance = 0.8))
# above is equivalent to
idf$set(r13layer = list(solar_absorptance = 0.8),
        r31layer = list(solar_absorptance = 0.8)
)

# use variable input
sets <- list(r13layer = list(roughness = "smooth"))
idf$set(sets)
}
```

Method `del()`**:** Delete existing objects

Usage:

```
Idf$del(
  ...,
  .ref_by = FALSE,
  .ref_to = FALSE,
  .recursive = FALSE,
  .force = FALSE
)
```

Arguments:

`...` integer vectors of object IDs and character vectors of object names in current `Idf` object.

`.ref_by` If `TRUE`, objects whose fields refer to input objects will also be deleted. Default: `FALSE`.

`.ref_to` If `TRUE`, objects whose fields are referred by input objects will also be deleted. Default: `FALSE`.

`.recursive` If `TRUE`, relation searching is performed recursively, in case that objects whose fields refer to target object are also referred by another object, and also objects whose fields are referred by target object are also referred by another object. Default: `FALSE`.

`.force` If `TRUE`, objects are deleted even if they are referred by other objects.

Details: $del() takes integer vectors of object IDs and character vectors of object names, and deletes objects specified.
If current validate level includes reference checking, objects will not be allowed to be deleted if they are referred by other objects. For example, an error will be issued if you want to delete one material that is referred by other constructions, because doing so will result in invalid field value references. You may bypass this if you really want to by setting `.force` to `TRUE`.
When `.ref_by` or `.ref_to` is `TRUE`, objects will be deleted only when they have and only have relation with input objects but not any other objects. For example, a construction `const` consist of 4 different materials. If `.ref_to` is `TRUE`, that 4 materials will only be deleted when they are only used in `const`, but not used in any other objects.
There are recursively reference relations in `Idf` object. For example, one material's name is referenced by one construction, and that construction's name can be referred by another surface. You can delete all of them by setting `.recursive` to `TRUE`.
If `.ref_by` is `TRUE`, objects whose fields refer to input objects will also be deleted.
IF `.ref_to` is `TRUE`, objects whose fields are referred by input objects will also be deleted.

Returns: The modified `Idf` object itself, invisibly.

Examples:

```
\dontrun{
# delete objects using names
idf$object("Fraction") # ScheduleTypeLimits
idf$del("Fraction")

# delete objects using IDs
idf$objects(c(39, 40)) # Output:Variable
idf$del(39, 40)

# cannot delete objects that are referred by others
level_checks()$reference # reference-checking is enable by default
idf$del("r13layer") # error

# force to delete objects even thay are referred by others
idf$del("r13layer", .force = TRUE)

# delete objects and also objects that refer to them
idf$del("r31layer", .ref_by = TRUE) # Construction 'ROOF31' will be kept

# delete objects and also objects that they refer to
idf$del("extlights", .ref_to = TRUE) # Schedule 'AlwaysOn' will be kept

# delete objects and also other objects that refer to them recursively
idf$del("roof31", .ref_by = TRUE, .recursive = TRUE)

# delete objects using variable inputs
ids <- idf$object_id("Output:Variable", simplify = TRUE)
idf$del(ids)
}
```

Method `purge()`**:** Purge resource objects that are not used

Usage:

```
Idf$purge(object = NULL, class = NULL, group = NULL)
```

Arguments:

`object` an integer vector of object IDs or a character vector of object names in current `Idf` object. Default: `NULL`.

`class` A character vector of valid class names in current `Idf` object. Default: `NULL`.

`group` A character vector of valid group names in current `Idf` object. Default: `NULL`.

Details: $purge() takes an integer vector of object IDs or a character vectors of object names, and deletes resource objects specified that are not used by any objects.
Here resource objects indicate all objects that can be referenced by other objects, e.g. all schedules. $purge() will ignore any inputs that are not resources. If inputs contain objects from multiple classes, references among them are also taken into account, which means purging is performed hierarchically. If both materials and constructions are specified, the latter will be purged first, because it is possible that input constructions reference input materials.

Returns: The modified `Idf` object itself, invisibly.

Examples:

```
\dontrun{
# purge unused "Fraction" schedule type
idf$purge("on/off") # ScheduleTypeLimits

# purge all unused schedule types
idf$purge(class = "ScheduleTypeLimits")

# purge all unused schedule related objects
idf$purge(group = "Schedules")
}
```

Method `duplicated()`**:** Determine duplicated objects

Usage:

```
Idf$duplicated(object = NULL, class = NULL, group = NULL)
```

Arguments:

`object` an integer vector of object IDs or a character vector of object names in current `Idf` object. Default: `NULL`.

`class` A character vector of valid class names in current `Idf` object. Default: `NULL`.

`group` A character vector of valid group names in current `Idf` object. Default: `NULL`.
If all `object`, `class` and `group` are `NULL`, duplication checking is performed on the whole `Idf`.

Details: $duplicated() takes an integer vector of object IDs or a character vectors of object names, and returns a `data.table::data.table()` to show whether input objects contain duplications or not.
Here duplicated objects refer to objects whose field values are the same except the names. Object comments are just ignored during comparison.

Returns: A data.table::data.table() of 4 columns:

- class: Character. Names of classes that input objects belong to
- id: Integer. Input object IDs
- name: Character. Input object names
- duplicate: Integer. The IDs of objects that input objects duplicate. If input object is not a duplication, NA is returned

Examples:

```
\dontrun{
# check if there are any duplications in the Idf
idf$duplicated(class = "ScheduleTypeLimits")

# check if there are any duplications in the schedule types
idf$duplicated(class = "ScheduleTypeLimits")

# check if there are any duplications in the schedule groups and
# material class
idf$duplicated(class = "Material", group = "Schedules")
}
```

Method `unique()`**:** Remove duplicated objects

Usage:

```
Idf$unique(object = NULL, class = NULL, group = NULL)
```

Arguments:

`object` an integer vector of object IDs or a character vector of object names in current `Idf` object. Default: `NULL`.

`class` A character vector of valid class names in current `Idf` object. Default: `NULL`.

`group` A character vector of valid group names in current `Idf` object. Default: `NULL`.
If all `object`, `class` and `group` are `NULL`, duplication checking is performed on the whole `Idf`.

Details: $unique() takes an integer vector of object IDs or a character vectors of object names, and remove duplicated objects.
Here duplicated objects refer to objects whose field values are the same except the names. Object comments are just ignored during comparison.
$unique() will only keep the first unique object and remove all redundant objects. Value referencing the redundant objects will be redirected into the unique object.

Returns: The modified `Idf` object itself, invisibly.

Examples:

```
\dontrun{
# remove duplications in the Idf
idf$unique(class = "ScheduleTypeLimits")

# remove duplications in the schedule types
idf$unique(class = "ScheduleTypeLimits")
```

```
# remove duplications in the schedule groups and material class
idf$unique(class = "Material", group = "Schedules")
}
```

Method `rename()`**:** Rename existing objects

Usage:

```
Idf$rename(...)
```

Arguments:

`...` Integer vectors of valid object IDs and character vectors of valid object names in current `Idf` object. Each element should be named. Names of input vectors are used as the new object names

Details: $rename() takes named character vectors of object names and named integer vectors of object IDs, renames specified objects to names of input vectors and returns a list of renamed IdfObjects. The returned list will be named using names of modified objects. An error will be issued if trying to "rename" an object which does not have name attribute. When renaming an object that is referred by other objects, corresponding fields that refer to that object's name will also be changed accordingly.

Returns: A named list of renamed IdfObject objects.

Examples:

```
\dontrun{
idf$objects(c("on/off", "test 352a"))
idf$rename(on_off = "on/off", test_352a = 51)
}
```

Method `insert()`**:** Insert new objects from IdfObjects

Usage:

```
Idf$insert(..., .unique = TRUE, .empty = FALSE)
```

Arguments:

`...` IdfObjects or lists of IdfObjects from same version as current `Idf` object.

`.unique` If there are duplications in input IdfObjects or there is same object in current `Idf` object, duplications in input are removed. Default: `TRUE`.

`.empty` If `TRUE`, trailing empty fields are kept. Default: `FALSE`.

Details: $insert() takes IdfObjects or lists of IdfObjects as input, inserts them into current `Idf` objects, and returns a list of inserted IdfObjects. The returned list will be named using names of inserted objects.
$insert() is quite useful to insert objects from other `Idf` objects. However, you cannot insert an IdfObject which comes from a different version than current `Idf` object.
$insert() will skip IdfObjects that have exactly same fields in current `Idf` object. If input IdfObject has the same name as one IdfObject in current `Idf` object but field values are not equal, an error will be issued if current validate level includes conflicted-name checking.
By default, trailing empty fields that are not required will be removed and only minimum required fields are kept. You can keep the trailing empty fields by setting `.empty` to `TRUE`.

Returns: A named list of inserted IdfObject objects.

Examples:

```
\dontrun{
# insert all material from another IDF
path_idf2 <- file.path(eplus_config(8.8)$dir, "ExampleFiles/5ZoneTDV.idf")
idf2 <- Idf$new(path_idf2)
idf$insert(idf2$Material)

# insert objects from same Idf is equivalent to using Idf$dup()
idf$insert(idf$SizingPeriod_DesignDay)
}
```

Method `load()`**:** Load new objects from characters or data.frames

Usage:

```
Idf$load(..., .unique = TRUE, .default = TRUE, .empty = FALSE)
```

Arguments:

`...` Character vectors or data.frames of object definitions.

`.unique` If `TRUE`, and there are duplications in input IdfObjects or there is same object in current `Idf` object, duplications in input are removed. Default: `TRUE`.

`.default` If `TRUE`, default values are filled for those blank fields if possible. Default: `TRUE`.

`.empty` If `TRUE`, trailing empty fields are kept. Default: `FALSE`.

Details: $load() is similar to `$insert()`, except it takes directly character vectors or data.frames as IdfObject definitions, insert corresponding objects into current `Idf` object and returns a named list of newly added IdfObjects. The returned list will be named using names of added objects. This makes it easy to create objects using the output from$to_string() and $to_table() method from `Idd`, `IddObject`, also from `Idf`, and `IdfObject`, class.
For object definitions in character vector format, they follow the same rules as a normal IDF file:

- Each object starts with a class name and a comma (,);
- Separates each values with a comma (,);
- Ends an object with a semicolon (;) for the last value.

Each character vector can contain:

- One single object, e.g. `c("Building,","MyBuilding;")`, or "Building, MyBuilding;".
- Multiple objects, e.g. `c("Building,MyBuilding;","SimulationControl,Yes")`.

You can also provide an option header to indicate if input objects are presented in IP units, using !-Option ViewInIPunits. If this header does not exist, then all values are treated as in SI units.
For object definitions in data.frame format, it is highly recommended to use $to_table() method in `Idd`, Idd, `IddObject`, IddObject, `Idf`, and `IdfObject`, class to create an acceptable data.frame template. A valid definition requires at least three columns described below. Note that column order does not matter.

- `class`:Character type. Valid class names in the underlying Idd object.
- `index`:Integer type. Valid field indices for each class.
- `value`:Character type or list type. Value for each field to be added.
  - If character type, usually when `string_value` is `TRUE` in method $to_table() in `Idf` and `IdfObject` class. Note that each value should be given as a string even if the corresponding field is a numeric type.

  - If list type, usually when `string_value` is set to `FALSE` in method$to_table() in `Idf` and `IdfObject` class. Each value should have the right type as the corresponding field definition. Otherwise, errors will be issued if current validation level includes invalid-type checking.
- `id`: **Optional**. Integer type. If input data.frame includes multiple object definitions in a same class, values in `id` column will be used to distinguish each definition. If `id` column does not exists, it assumes that each definition is separated by `class` column and will issue an error if there is any duplication in the `index` column.

Note that $load() assumes all definitions are from the same version as current `Idf` object. If input definition is from different version, parsing error may occur.
By default, trailing empty fields that are not required will be removed and only minimum required fields are kept. You can keep the trailing empty fields by setting `.empty` to `TRUE`.

Returns: A named list of loaded IdfObject objects.

Examples:

```
\dontrun{
# load objects from character vectors
idf$load(
    c("Material,",
      "    mat,                    !- Name",
      "    MediumSmooth,           !- Roughness",
      "    0.667,                  !- Thickness {m}",
      "    0.115,                  !- Conductivity {W/m-K}",
      "    513,                    !- Density {kg/m3}",
      "    1381;                   !- Specific Heat {J/kg-K}"),

    "Construction, const, mat;"
)

# load objects from data.frame definitions
dt <- idf$to_table(class = "Material")
dt[field == "Name", value := paste(value, 1)]
dt[field == "Thickness", value := "0.5"]
idf$load(dt)

# by default, duplications are removed
idf$load(idf$to_table(class = "Material"))

# keep empty fields as they are
idf$load("Material, mat1, smooth, 0.5, 0.2, 500, 1000,,, 0.5;", .default = FALSE)

# keep trailing empty fields
idf$load("Material, mat2, smooth, 0.5, 0.2, 500, 1000,,,;",
    .default = FALSE, .empty = TRUE
)
}
```

Method `update()`**:** Update existing object values from characters or data.frames

Usage:

```
Idf$update(..., .default = TRUE, .empty = FALSE)
```

Arguments:

`...` Character vectors or data.frames of object definitions.

`.default` If `TRUE`, default values are filled for those blank fields if possible. Default: `TRUE`.

`.empty` If `TRUE`, trailing empty fields are kept. Default: `FALSE`.

Details: $update() is similar to `$set()`, except it takes directly character vectors or data.frames as IdfObject definitions, updates new values for fields in objects specified, and returns a named list of modified IdfObjects. The returned list will be named using names of modified objects. This makes it easy to update object values using the output from $to_string() and $to_table method from `Idf`, and `IdfObject`, class.
The format of object definitions is similar to $load().
For object definitions in character vector format, object names are used to locate which objects to update. Objects that have name attribute should have valid names. This means that there is no way to update object names using character vector format, but this can be achieved using data.frame format as it uses object IDs instead of object names to locate objects. The format of acceptable characters follows the same rules as a normal IDF file:

- Each object starts with a class name and a comma (,);
- Separates each values with a comma (,);
- Ends an object with a semicolon (;) for the last value.

Each character vector can contain:

- One single object, e.g. `c("Building,","MyBuilding;")`, or "Building, MyBuilding;".
- Multiple objects, e.g. `c("Building,MyBuilding;","SimulationControl,Yes")`.

You can also provide an option header to indicate if input objects are presented in IP units, using !-Option ViewInIPunits. If this header does not exist, then all values are treated as in SI units.
For object definitions in data.frame format, it is highly recommended to use $to_table() method in `Idf`, and `IdfObject`, class to create an acceptable data.frame template. A valid definition requires three columns described below. Note that column order does not matter.

- `id`: Integer type. Valid IDs of objects to update.
- `index`:Integer type. Valid field indices for each object.
- `value`:Character type or list type. Value for each field to be added.
  - If character type, usually when `string_value` is `TRUE` in method $to_table() in `Idf` and `IdfObject` class. Note that each value should be given as a string even if the corresponding field is a numeric type.
  - If list type, usually when `string_value` is set to `FALSE` in method $to_table() in `Idf` and `IdfObject` class. Each value should have the right type as the corresponding field definition. Otherwise, errors will be issued if current validation level includes invalid-type checking.

Note that $update() assumes all definitions are from the same version as current `Idf` object. If input definition is from different version, parsing error may occur.
By default, trailing empty fields that are not required will be removed and only minimum required fields are kept. You can keep the trailing empty fields by setting `.empty` to `TRUE`.

Returns: A named list of updated IdfObject objects.

Examples:

```
\dontrun{
# update objects from string definitions:
str <- idf$to_string("zone one", header = FALSE, format = "new_top")
str[8] <- "2," # Multiplier
idf$update(str)

# update objects from data.frame definitions:
dt <- idf$to_table("zone one")
dt[field == "Multiplier", value := "1"]
idf$update(dt)
}
```

Method `paste()`**:** Paste new objects from IDF Editor

Usage:

```
Idf$paste(in_ip = FALSE, ver = NULL, unique = TRUE, empty = FALSE)
```

Arguments:

`in_ip` Set to `TRUE` if the IDF file is open with `Inch-Pound` view option toggled. Numeric values will automatically converted to SI units if necessary. Default: `FALSE`.

`ver` The version of IDF file open by IDF Editor, e.g. `8.6`, `"8.8.0"`. If `NULL`, assume that the file has the same version as current Idf object. Default: `NULL`.

`unique` If `TRUE`, and there are duplications in copied objects from IDF Editor or there is same object in current Idf, duplications in input are removed. Default: `TRUE`.

`empty` If `TRUE`, trailing empty fields are kept. Default: `FALSE`.

Details: $paste() reads the contents (from clipboard) of copied objects from IDF Editor (after hitting Copy Obj button), inserts corresponding objects into current `Idf` object and returns a named list of newly added IdfObjects. The returned list will be named using names of added objects. As IDF Editor is only available on Windows platform, $paste() only works on Windows too.
There is no version data copied to the clipboard when copying objects in IDF Editor. $paste() assumes the file open in IDF Editor has the same version as current `Idf` object. This may not be always true. Please check the version before running $paste(), or explicitly specify the version of file opened by IDF Editor using `ver` parameter. Parsing error may occur if there is a version mismatch.
By default, trailing empty fields that are not required will be removed and only minimum required fields are kept. You can keep the trailing empty fields by setting `.empty` to `TRUE`.

Returns: A named list of loaded IdfObject objects.

Method `search_value()`**:** Search objects by field values using regular expression

Usage:

```
Idf$search_value(
  pattern,
  class = NULL,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
```

```
  useBytes = FALSE
)
```

Arguments:

`pattern`, `ignore.case`, `perl`, `fixed`, `useBytes` All of them are directly passed to base::grepl and base::gsub.

`class` A character vector of invalid class names in current `Idf` object to search for values. If `NULL`, all classes are used. Default: `NULL`.

Details: $search_value() returns a list of IdfObjects that contain values which match the given pattern. If no matched found, `NULL` is returned invisibly. The returned list will be named using names of matched objects.
Note that during matching, all values are treated as characters, including numeric values.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# search values that contains "floor"
idf$search_value("floor", ignore.case = TRUE)

# search values that contains "floor" in class Construction
idf$search_value("floor", "Construction", ignore.case = TRUE)
}
```

Method `replace_value()`**:** Replace object field values using regular expression

Usage:

```
Idf$replace_value(
  pattern,
  replacement,
  class = NULL,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments:

`pattern`, `replacement`, `ignore.case`, `perl`, `fixed`, `useBytes` All of them are directly passed to base::grepl and base::gsub.

`class` A character vector of invalid class names in current `Idf` object to search for values. If `NULL`, all classes are used. Default: `NULL`.

Details: $replace_value() returns a list of IdfObjects whose values have been replaced using given pattern. If no matched found, `NULL` is returned invisibly. The returned list will be named using names of matched objects.
Note that during matching, all values are treated as characters, including numeric values.
Modifying object values using regular expression is not recommended. Consider to use `$set()` and `$update()` if possible. Validation rules also apply during replacing.

Returns: A named list of IdfObject objects.

Examples:

```
\dontrun{
# search values that contains "win" and replace them with "windows"
idf$replace_value("win", "windows")
}
```

Method `validate()`**:** Check possible object field value errors

Usage:

```
Idf$validate(level = eplusr_option("validate_level"))
```

Arguments:

`level` One of `"none"`, `"draft"`, `"final"` or a list of 10 elements with same format as `custom_validate()` output.

Details: $validate() checks if there are errors in current `Idf` object under specified validation level and returns an `IdfValidity` object. $validate() is useful to help avoid some common errors before running the model. By default, validation is performed when calling all methods that modify objects, e.g. `$dup()` `$add()`, `$set()`, `$del()`, and etc.
In total, there are 10 different validate checking components:

- `required_object`: Check if required objects are missing in current `Idf`.
- `unique_object`: Check if there are multiple objects in one unique-object class. An unique-object class means that there should be at most only one object existing in that class.
- `unique_name`: Check if all objects in each class have unique names.
- `extensible`: Check if all fields in an extensible group have values. An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together. `extensible` component checks if there are some, but not all, fields in an extensible group are empty.
- `required_field`: Check if all required fields have values.
- `auto_field`: Check if all fields filled with value `"Autosize"` and `"Autocalculate"` are actual autosizable and autocalculatable fields or not.
- `type`: Check if all fields have value types complied with their definitions, i.e. character, numeric and integer fields should be filled with corresponding type of values.
- `choice`: Check if all choice fields are filled with valid choice values.
- `range`: Check if all numeric fields have values within prescibed ranges.
- `reference`: Check if all fields whose values refer to other fields are valid.

The `level` argument controls what checkings should be performed. `level` here is just a list of 10 element which specify the toggle status of each component. You can use helper `custom_validate()` to get that list and pass it directly to `level`.
There are 3 predefined validate level that indicates different combinations of checking components, i.e. `none`, `draft` and `final`. Basically, `none` level just does not perform any checkings; `draft` includes 5 components, i.e. `auto_field`, `type`, `unique_name`, `choice` and `range`; and `final` level includes all 10 components. You can always get what components each level contains using `level_checks()`. By default, the result from `eplusr_option("validate_level")` is passed to `level`. If not set, `final` level is used.
Underneath, an `IdfValidity` object which $validate() returns is a list of 13 element as shown below. Each element or several elements represents the results from a single validation checking component.

- `missing_object`: Result of `required_object` checking.
- `duplicate_object`: Result of `unique_object` checking.
- `conflict_name`: Result of `unique_name` checking.
- `incomplete_extensible`: Result of `extensible` checking.
- `missing_value`: Result of `required_field` checking.
- `invalid_autosize`: Result of `auto_field` checking for invalid `Autosize` field values.
- `invalid_autocalculate`: Result of `auto_field` checking for invalid `Autocalculate` field values.
- `invalid_character`: Result of `type` checking for invalid character field values.
- `invalid_numeric`: Result of `type` checking for invalid numeric field values.
- `invalid_integer`: Result of `type` checking for invalid integer field values.
- `invalid_choice`: Result of `choice` checking.
- `invalid_range`: Result of `range` checking.
- `invalid_reference`: Result of `reference` checking.

Except `missing_object`, which is a character vector of class names that are missing, all other elements are data.table with 9 columns containing data of invalid field values:

- `object_id`: IDs of objects that contain invalid values
- `object_name`: names of objects that contain invalid values
- `class_id`: indexes of classes that invalid objects belong to
- `class_name`: names of classes that invalid objects belong to
- `field_id`: indexes (at Idd level) of object fields that are invalid
- `field_index`: indexes of object fields in corresponding that are invalid
- `field_name`: names (without units) of object fields that are invalid
- `units`: SI units of object fields that are invalid
- `ip_units`: IP units of object fields that are invalid
- `type_enum`: An integer vector indicates types of invalid fields
- `value_id`: indexes (at Idf level) of object field values that are invalid
- `value_chr`: values (converted to characters) of object fields that are invalid
- `value_num`: values (converted to numbers in SI units) of object fields that are invalid

Knowing the internal structure of `IdfValidity`, it is easy to extract invalid IdfObjects you interested in. For example, you can get all IDs of objects that contain invalid value references using `model$validate()$invalid_reference$object_id`. Then using `$set()` method to correct them.

Different validate result examples are shown below:

- No error is found:

  ```
  v No error found.
  ```

  Above result shows that there is no error found after conducting all validate checks in specified validate level.

- Errors are found:

  ```
   x [2] Errors found during validation.
  ================================================================================

  -- [2] Invalid Autocalculate Field ---------------------------------------
     Fields below cannot be `autocalculate`:
  ```

```
    Class: <AirTerminal:SingleDuct:VAV:Reheat>
    \- Object [ID:176] <SPACE5-1 VAV Reheat>
     +- 17: AUTOCALCULATE, !- Maximum Flow per Zone Floor Area During Reheat {m3/s-m2}
        \- 18: AUTOCALCULATE; !- Maximum Flow Fraction During Reheat
```

Above result shows that after all validate components performed under current validate level, 2 invalid field values are found. All of them are in a object named SPACE5-1 VAV Reheat with ID 176. They are invalid because those two fields do not have an autocalculatable attribute but are given AUTOCALCULATE value. Knowing this info, one simple way to fix the error is to correct those two fields by doing:

```
idf$set(..176 =
    list(`Maximum Flow per Zone Floor Area During Reheat` = "autosize",
         `Maximum Flow Fraction During Reheat` = "autosize"
    )
)
```

Returns: An IdfValidity object.

Examples:

```
\dontrun{
idf$validate()

# check at predefined validate level
idf$validate("none")
idf$validate("draft")
idf$validate("final")

# custom validate checking components
idf$validate(custom_validate(auto_field = TRUE, choice = TRUE))
}
```

Method `is_valid()`**:** Check if there is any error in current `Idf`

Usage:

```
Idf$is_valid(level = eplusr_option("validate_level"))
```

Arguments:

`level` One of `"none"`, `"draft"`, `"final"` or a list of 10 elements with same format as `custom_validate()` output.

Details: $is_valid() checks if there are errors in current `Idf` object under specified validation level and returns `TRUE` or `FALSE` accordingly. For detailed description on validate checking, see `$validate()` documentation above.

Returns: A single logical value of `TRUE` or `FALSE`.

Examples:

```
\dontrun{
idf$is_valid()

# check at predefined validate level
```

```
idf$is_valid("none")
idf$is_valid("draft")
idf$is_valid("final")

# custom validate checking components
idf$is_valid(custom_validate(auto_field = TRUE, choice = TRUE))
}
```

Method `to_string()`**:** Format `Idf` as a character vector

Usage:

```
Idf$to_string(
  which = NULL,
  class = NULL,
  comment = TRUE,
  header = TRUE,
  format = eplusr_option("save_format"),
  leading = 4L,
  sep_at = 29L
)
```

Arguments:

`which` Either an integer vector of valid object IDs or a character vector of valid object names. If `NULL`, the whole `Idf` object is converted. Default: `NULL`.

`class` A character vector of class names. If `NULL`, all classed in current `Idf` object is converted. Default: `NULL`.

`comment` If `FALSE`, all comments will not be included. Default: `TRUE`.

`header` If `FALSE`, the header will not be included. Default: `TRUE`.

`format` Specific format used when formatting. Should be one of `"asis"`, `"sorted"`, `"new_top"`, and `"new_bot"`.

- If `"asis"`, `Idf` object will be formatted in the same way as it was when first read. If `Idf` object does not contain any format saving option, which is typically the case when the model was not saved using eplusr or IDFEditor, `"sorted"` will be used.
- `"sorted"`, `"new_top"` and `"new_bot"` are the same as the save options `"Sorted"`, `"Original with New at Top"`, and `"Original with New at Bottom"` in IDFEditor. Default: `eplusr_option("save_format"`

`leading` Leading spaces added to each field. Default: `4L`.

`sep_at` The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.

Details: $to_string() returns the text format of parts or whole `Idf` object.

Returns: A character vector.

Examples:

```
\dontrun{
# get text format of the whole Idf
head(idf$to_string())

# get text format of the whole Idf, excluding the header and all comments
```

```
head(idf$to_string(comment = FALSE, header = FALSE))

# get text format of all objects in class Material
head(idf$to_string(class = "Material", comment = FALSE, header = FALSE))

# get text format of some objects
head(idf$to_string(c("floor", "zone one")))

# tweak output formatting
head(idf$to_string("floor", leading = 0, sep_at = 0))
}
```

Method `to_table()`**:** Format `Idf` as a data.frame

Usage:

```
Idf$to_table(
  which = NULL,
  class = NULL,
  string_value = TRUE,
  unit = FALSE,
  wide = FALSE,
  align = FALSE,
  all = FALSE,
  group_ext = c("none", "group", "index"),
  force = FALSE,
  init = FALSE
)
```

Arguments:

`which` Either an integer vector of valid object IDs or a character vector of valid object names. If `NULL`, the whole `Idf` object is converted. Default: `NULL`.

`class` A character vector of class names. If `NULL`, all classed in current `Idf` object is converted. Default: `NULL`.

`string_value` If `TRUE`, all field values are returned as character. If `FALSE`, `value` column in returned data.table is a list column with each value stored as corresponding type. Note that if the value of numeric field is set to `"Autosize"` or `"Autocalculate"`, it is left as it is, leaving the returned type being a string instead of a number. Default: `TRUE`.

`unit` Only applicable when `string_value` is `FALSE`. If `TRUE`, values of numeric fields are assigned with units using `units::set_units()` if applicable. Default: `FALSE`.

`wide` Only applicable if target objects belong to a same class. If `TRUE`, a wide table will be returned, i.e. first three columns are always `id`, `name` and `class`, and then every field in a separate column. Note that this requires all objects specified must from the same class. Default: `FALSE`.

`align` If `TRUE`, all objects in the same class will have the same field number. The number of fields is the same as the object that have the most fields among objects specified. Default: `FALSE`.

`all` If `TRUE`, all available fields defined in IDD for the class that objects belong to will be returned. Default: `FALSE`.

`group_ext` Should be one of `"none"`, `"group"` or `"index"`. If not `"none"`, `value` column in returned `data.table::data.table()` will be converted into a list. If `"group"`, values from extensible fields will be grouped by the extensible group they belong to. For example, coordinate values of each vertex in class `BuildingSurface:Detailed` will be put into a list. If `"index"`, values from extensible fields will be grouped by the extensible field indice they belong to. For example, coordinate values of all x coordinates will be put into a list. If `"none"`, nothing special will be done. Default: `"none"`.

`force` If `TRUE`, `wide` can be `TRUE` even though there are multiple classes in input. This can result in a data.table with lots of columns. But may be useful when you know that target classes have the exact same fields, e.g. `Ceiling:Adiabatic` and `Floor:Adiabatic`. Default: `FALSE`.

`init` If `TRUE`, a table for new object input will be returned with all values filled with defaults. In this case, `object` input will be ignored. The `id` column will be filled with possible new object IDs. Default: `FALSE`.

Details: $to_table() returns a data.table that contains core data of specified objects. The returned data.table has 5 columns:

- `id`: Integer type. Object IDs.
- `name`: Character type. Object names.
- `class`: Character type. Current class name.
- `index`: Integer type. Field indexes.
- `field`: Character type. Field names.
- `value`: Character type if `string_value` is `TRUE` or list type if `string_value` is `FALSE` or `group_ext` is not `"none"`. Field values.

Note that when `group_ext` is not `"none"`, `index` and `field` values will not match the original field indices and names. In this case, `index` will only indicate the indices of sequences. For `field` column, specifically:

- When `group_ext` is `"group"`, each field name in a extensible group will be abbreviated using `abbreviate()` with `minlength` being `10L` and all abbreviated names will be separated by `|` and combined together. For example, field names in the extensible group (Vertex 1 X-coordinate, Vertex 1 Y-coordinate, Vertex 1 Z-coordinate) in class `BuildiBuildingSurface:Detailed` will be merged into one name `Vrtx1X-crd|Vrtx1Y-crd|Vrtx1Z-crd`.
- When `group_ext` is `"index"`, the extensible group indicator in field names will be removed. Take the same example as above, the resulting field names will be Vertex X-coordinate, Vertex Y-coordinate, and Vertex Z-coordinate.

Returns: A data.table with 6 columns (if `wide` is `FALSE`) or at least 6 columns (if `wide` is `TRUE`).

Examples:

```
\dontrun{
# extract whole Idf data
idf$to_table()

# extract all data from class Material
idf$to_table(class = "Material")

# extract multiple object data
idf$to_table(c("FLOOR", "ZONE ONE"))
```

```
# keep value types and put actual values into a list column
idf$to_table(c("FLOOR", "ZONE ONE"), string_value = FALSE)$value

# add the unit to each value
idf$to_table(c("FLOOR", "ZONE ONE"), string_value = FALSE, unit = TRUE)

# get all possible fields
idf$to_table("ZONE ONE", all = TRUE)

# make sure all objects in same class have the same number of fields
idf$to_table(class = "Construction", align = TRUE)

# get a wide table with string values
idf$to_table(class = "Construction", wide = TRUE)

# get a wide table with actual values
idf$to_table(class = "OtherEquipment", wide = TRUE, string_value = FALSE)

# group extensible by extensible group number
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "group")

# group extensible by extensible group number and convert into a wide table
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "group", wide = TRUE)

# group extensible by extensible field index
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "index")

# group extensible by extensible field index and convert into a wide table
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "index", wide = TRUE)

# when grouping extensible, 'string_value' and 'unit' still take effect
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "index",
    wide = TRUE, string_value = FALSE, unit = TRUE
)

# create table for new object input
idf$to_table(class = "BuildingSurface:Detailed", init = TRUE)
}
```

Method `is_unsaved()`**:** Check if there are unsaved changes in current `Idf`

Usage:

```
Idf$is_unsaved()
```

Details: $is_unsaved() returns `TRUE` if there are modifications on the model since it was read or since last time it was saved, and returns `FALSE` otherwise.

Returns: A single logical value of `TRUE` or `FALSE`.

Examples:

```
\dontrun{
idf$is_unsaved()
}
```

Method `save()`**:** Save `Idf` object as an IDF file

Usage:

```
Idf$save(
  path = NULL,
  format = eplusr_option("save_format"),
  overwrite = FALSE,
  copy_external = TRUE
)
```

Arguments:

`path` A path where to save the IDF file. If `NULL`, the path of the `Idf` itself, i.e. `$path()`, will be used.

`format` Specific format used when formatting. Should be one of `"asis"`, `"sorted"`, `"new_top"`, and `"new_bot"`.

- If `"asis"`, `Idf` object will be formatted in the same way as it was when first read. If `Idf` object does not contain any format saving option, which is typically the case when the model was not saved using eplusr or IDFEditor, `"sorted"` will be used.
- `"sorted"`, `"new_top"` and `"new_bot"` are the same as the save options `"Sorted"`, `"Original with New at Top"`, and `"Original with New at Bottom"` in IDFEditor. Default: `eplusr_option("save_format'`

`overwrite` Whether to overwrite the file if it already exists. Default: `FALSE`.

`copy_external` If `TRUE`, the external files that current `Idf` object depends on will also be copied into the same directory. The values of file paths in the `Idf` will be changed into relative path automatically. This makes it possible to create fully reproducible simulation conditions. Currently, only `Schedule:File` class is supported. Default: `FALSE`.

Details: $save() formats current `Idf` object, saves it as an IDF file and returns the path of saved file invisibly. After saving, `$path()` will also be updated to return the path of saved file.

Returns: A length-one character vector, invisibly.

Examples:

```
\dontrun{
# save Idf as a new file
idf$save(tempfile(fileext = ".idf"))

# save and overwrite current file
idf$save(overwrite = TRUE)

# save the model with newly created and modified objects at the top
idf$save(overwrite = TRUE, format = "new_top")

# save the model to a new file and copy all external csv files used in
# "Schedule:File" class into the same folder
idf$save(path = file.path(tempdir(), "test1.idf"), copy_external = TRUE)
}
```

Method `run()`: Run simulation using EnergyPlus

Usage:

```
Idf$run(
  weather,
  dir = NULL,
  wait = TRUE,
  force = FALSE,
  copy_external = FALSE,
  echo = wait
)
```

Arguments:

`weather` A path to an `.epw` file or an Epw object. `weather` can also be `NULL` which will force design-day-only simulation. Note that this needs at least one `Sizing:DesignDay` object exists in the `Idf`.

`dir` The directory to save the simulation results. If `NULL`, the folder of `Idf` path will be used. Default: `NULL`.

`wait` Whether to wait until the simulation completes and print the standard output and error of EnergyPlus. If `FALSE`, the simulation will run in the background. Default is `TRUE`.

`force` Only applicable when the last simulation runs with `wait` equals to `FALSE` and is still running. If `TRUE`, current running job is forced to stop and a new one will start. Default: `FALSE`.

`copy_external` If `TRUE`, the external files that current `Idf` object depends on will also be copied into the simulation output directory. The values of file paths in the Idf will be changed automatically. This ensures that the output directory will have all files needed for the model to run. Default is `FALSE`.

`echo` Only applicable when `wait` is `TRUE`. Whether to show standard output and error from EnergyPlus. Default: same as `wait`.

Details: $run() calls corresponding version of EnergyPlus to run the current `Idf` object together with specified weather. The model and the weather used will be copied into the output directory. An EplusJob object is returned which provides detailed info of the simulation and methods to collect simulation results. Please see EplusJob for details.
eplusr uses the EnergyPlus command line interface which was introduced since EnergyPlus 8.3.0. So $run() only supports models with version no lower than 8.3.0.
When calling $run(), eplusr will do steps below to make sure the output collecting methods work as expected. Please note that this may result in an IDF file that may not be exactly same as your current `Idf` object.

- eplusr uses EnergyPlus SQL output for extracting simulation results. In order to do so, an object in `Output:SQLite` class with Option Type value being `SimpleAndTabular` will be automatically created if it does not exists.
- In order to make sure `.rdd` (Report Data Dictionary) and `.mdd` (Meter Data Dictionary) files are created during simulation, an object in `Output:VariableDictionary` class with Key Field value being `IDF` will be automatically created if it does not exists.

Returns: An EplusJob object of current simulation.

Examples:

```
\dontrun{
idf <- Idf$new(path_idf)
# save the model to tempdir()
idf$save(file.path(tempdir(), "test_run.idf"))

# use the first epw file in "WeatherData" folder in EnergyPlus v8.8
# installation path
epw <- list.files(file.path(eplus_config(8.8)$dir, "WeatherData"),
    pattern = "\\.epw$", full.names = TRUE)[1]

# if `dir` is NULL, the directory of IDF file will be used as simulation
# output directory
job <- idf$run(epw, dir = NULL)

# run simulation in the background
idf$run(epw, dir = tempdir(), wait = FALSE)

# copy all external files into the directory run simulation
idf$run(epw, dir = tempdir(), copy_external = TRUE)

# check for simulation errors
job$errors()

# get simulation status
job$status()

# get output directory
job$output_dir()

# re-run the simulation
job$run()

# get simulation results
job$report_data()
}
```

Method `last_job()`**:** Get the last simulation job

Usage:

```
Idf$last_job()
```

Details: $last_job() returns the last EplusJob object that was created using `$run()`. If the `Idf` hasn't been run yet, `NULL` is returned.

Returns: `NULL` or an EplusJob object.

Examples:

```
\dontrun{
idf$last_job()
}
```

Method `geometry()`**:** Extract `Idf` geometries

Usage:

```
Idf$geometry()
```

Details: $geometry() extracts all geometry objects into an IdfGeometry object. `IdfGeometry` is an abstraction of a collection of geometry in an Idf. It provides more detail methods to query geometry properties, update geometry vertices and visualize geometry in 3D using the rgl package.

Returns: An IdfGeometry object.

Examples:

```
\dontrun{
idf$geometry()
}
```

Method `view()`**:** View 3D `Idf` geometry

Usage:

```
Idf$view(
  new = FALSE,
  render_by = "surface_type",
  wireframe = TRUE,
  x_ray = FALSE,
  axis = TRUE
)
```

Arguments:

`new` If `TRUE`, a new rgl window will be open using `rgl::rgl.open()`. If `FALSE`, existing rgl window will be reused if possible. Default: `FALSE`.

`render_by` A single string specifying the way of rendering the geometry. Possible values are:

- `"surface_type"`: Default. Render the model by surface type model. Walls, roofs, windows, doors, floors, and shading surfaces will have unique colors.
- `"boundary"`: Render the model by outside boundary condition. Only surfaces that have boundary conditions will be rendered with a color. All other surfaces will be white.
- `"construction"`: Render the model by surface constructions.
- `"zone"`: Render the model by zones assigned.
- `"normal"`: Render the model by surface normal. The outside face of a heat transfer face will be rendered as white and the inside face will be rendered as red.

`wireframe` If `TRUE`, the wireframe of each surface will be shown. Default: `TRUE`.

`x_ray` If `TRUE`, all surfaces will be rendered translucently. Default: `FALSE`.

`axis` If `TRUE`, the X, Y and Z axes will be drawn at the global origin. Default: `TRUE`.

Details: $view() uses the rgl package to visualize the IDF geometry in 3D in a similar way as OpenStudio.
$view() returns an IdfViewer object which can be used to further tweak the viewer scene.
In the rgl window, you can control the view using your mouse:

- Left button: Trackball
- Right button: Pan

- Middle button: Field-of-view (FOV). '0' means orthographic projection.
- Wheel: Zoom

Returns: An IdfViewer object

Examples:

```
\dontrun{
idf$view()
idf$view(render_by = "zone")
idf$view(render_by = "construction")
}
```

Method `print()`**:** Print `Idf` object

Usage:

```
Idf$print(zoom = "class", order = TRUE)
```

Arguments:

`zoom` Control how detailed of the Idf object should be printed. Should be one of `"group"`, `"class"`, `"object"` and `"field"`. Default: `"group"`.

- `"group"`: all group names current existing are shown with prevailing square bracket showing how many **C**lasses existing in that group.
- `"class"`: all class names are shown with prevailing square bracket showing how many **O**bjects existing in that class, together with parent group name of each class.
- `"object"`: all object IDs and names are shown, together with parent class name of each object.
- `"field"`: all object IDs and names, field names and values are shown, together with parent class name of each object.

`order` Only applicable when `zoom` is `"object"` or `"field"`. If `TRUE`, objects are shown as the same order in the IDF. If `FALSE`, objects are grouped and ordered by classes. Default: `TRUE`.

Details: $print() prints the `Idf` object according to different detail level specified using the `zoom` argument.
With the default `zoom` level `object`, contents of the `Idf` object is printed in a similar style as you see in IDF Editor, with additional heading lines showing `Path`, `Version` of the `Idf` object. Class names of objects are ordered by group and the number of objects in classes are shown in square bracket.

Returns: The `Idf` object itself, invisibly.

Examples:

```
\dontrun{
idf$print("group")
idf$print("class")
idf$print("object")
idf$print("field")

# order objects by there classes
idf$print("object", order = FALSE)
idf$print("field", order = FALSE)
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Idf$clone(deep = TRUE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Hongyuan Jia

See Also

IdfObject class for a single object in an IDF.

Examples

```
## ------------------------------------------------
## Method `Idf$new`
## ------------------------------------------------

## Not run:
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# If neither EnergyPlus v8.8 nor Idd v8.8 was found, error will
# occur. If Idd v8.8 is found, it will be used automatically.
idf <- Idf$new(path_idf)

# argument `idd` can be specified explicitly using `use_idd()`
idf <- Idf$new(path_idf, idd = use_idd(8.8))

# you can set `download` arugment to "auto" in `use_idd()` if you
# want to automatically download corresponding IDD file when
# necessary
idf <- Idf$new(path_idf, use_idd(8.8, download = "auto"))

# Besides use a path to an IDF file, you can also provide IDF in literal
# string format
string_idf <-
    "
    Version, 8.8;
    Building,
        Building;                !- Name
    "

Idf$new(string_idf, use_idd(8.8, download = "auto"))

## End(Not run)
```

```
## ------------------------------------------------
## Method `Idf$version`
## ------------------------------------------------

## Not run:
# get version
idf$version()

## End(Not run)


## ------------------------------------------------
## Method `Idf$path`
## ------------------------------------------------

## Not run:
# get path
idf$path()

# return `NULL` if Idf is not created from a file
Idf$new("Version, 8.8;\n")$path()

## End(Not run)


## ------------------------------------------------
## Method `Idf$group_name`
## ------------------------------------------------

## Not run:
# get names of all groups Idf contains
idf$group_name()

# get group name of each object in Idf
idf$group_name(sorted = FALSE)

# get names of all available groups in underlying Idd
idf$group_name(all = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$class_name`
## ------------------------------------------------

## Not run:
# get names of all classes in Idf
idf$class_name()

# get names of all classes grouped by group names in Idf
idf$class_name(by_group = TRUE)
```

```
# get class name of each object in Idf
idf$class_name(sorted = FALSE)

# get names of all available classes in underlying Idd
idf$class_name(all = TRUE)

# get names of all available classes grouped by group names in
# underlying Idd
idf$class_name(all = TRUE, by_group = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$is_valid_group`
## ------------------------------------------------

## Not run:
# check if input is a valid group name in current Idf
idf$is_valid_group(c("Schedules", "Compliance Objects"))

# check if input is a valid group name in underlying Idd
idf$is_valid_group(c("Schedules", "Compliance Objects"), all = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$is_valid_class`
## ------------------------------------------------

## Not run:
# check if input is a valid class name in current Idf
idf$is_valid_class(c("Building", "ShadowCalculation"))

# check if input is a valid class name in underlying Idd
idf$is_valid_class(c("Building", "ShadowCalculation"), all = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$definition`
## ------------------------------------------------

## Not run:
# get the IddObject object for specified class
idf$definition("Version")

## End(Not run)


## ------------------------------------------------
```

```
## Method `Idf$object_id`
## ------------------------------------------------

## Not run:
# get IDs of all objects in current Idf object
idf$object_id()

# get IDs of all objects in current Idf object, and merge them into a
# single integer vector
idf$object_id(simplify = TRUE)

# get IDs of objects in class Version and Zone
idf$object_id(c("Version", "Zone"))

# get IDs of objects in class Version and Zone, and merge them into a
# single integer vector
idf$object_id(c("Version", "Zone"), simplify = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$object_name`
## ------------------------------------------------

## Not run:
# get names of all objects in current Idf object
idf$object_name()

# get names of all objects in current Idf object, and merge them into
# a single character vector
idf$object_name(simplify = TRUE)

# get names of objects in class Version and Zone
idf$object_name(c("Version", "Zone"))

# get names of objects in class Version and Zone, and merge them into
# a single character vector
idf$object_name(c("Version", "Zone"), simplify = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$object_num`
## ------------------------------------------------

## Not run:
# get total number of objects
idf$object_num()

# get number of objects in class Zone and Schedule:Compact
idf$object_num(c("Zone", "Schedule:Compact"))
```

```
## End(Not run)


## ------------------------------------------------
## Method `Idf$is_valid_id`
## ------------------------------------------------

## Not run:
idf$is_valid_id(c(51, 1000))

## End(Not run)


## ------------------------------------------------
## Method `Idf$is_valid_name`
## ------------------------------------------------

## Not run:
idf$is_valid_name(c("Simple One Zone (Wireframe DXF)", "ZONE ONE", "a"))

# name matching is case-insensitive
idf$is_valid_name(c("simple one zone (wireframe dxf)", "zone one", "a"))

## End(Not run)


## ------------------------------------------------
## Method `Idf$object`
## ------------------------------------------------

## Not run:
# get an object whose ID is 3
idf$object(3)

# get an object whose name is "simple one zone (wireframe dxf)"
# NOTE: object name matching is case-insensitive
idf$object("simple one zone (wireframe dxf)")

## End(Not run)


## ------------------------------------------------
## Method `Idf$objects`
## ------------------------------------------------

## Not run:
# get objects whose IDs are 3 and 10
idf$objects(c(3,10))

# get objects whose names are "Simple One Zone (Wireframe DXF)" and "ZONE ONE"
# NOTE: object name matching is case-insensitive
idf$objects(c("Simple One Zone (Wireframe DXF)", "zone one"))
```

```
## End(Not run)


## ------------------------------------------------
## Method `Idf$object_unique`
## ------------------------------------------------

## Not run:
# get the SimulationColtrol object
idf$object_unique("SimulationControl")

# S3 "[[" and "$" can also be used
idf$SimulationControl
idf[["SimulationControl"]]

## End(Not run)


## ------------------------------------------------
## Method `Idf$objects_in_class`
## ------------------------------------------------

## Not run:
# get all objects in Zone class
idf$objects_in_class("Zone")

# S3 "[[" and "$" can also be used
idf$Zone
idf[["Zone"]]

## End(Not run)


## ------------------------------------------------
## Method `Idf$objects_in_group`
## ------------------------------------------------

## Not run:
# get all objects in Schedules group
idf$objects_in_group("Schedules")

## End(Not run)


## ------------------------------------------------
## Method `Idf$object_relation`
## ------------------------------------------------

## Not run:
# check each layer's reference of a construction named FLOOR
idf$object_relation("floor", "ref_to")
```

```
# check where is this construction being used
idf$object_relation("floor", "ref_by")

## End(Not run)


## ------------------------------------------------
## Method `Idf$objects_in_relation`
## ------------------------------------------------

## Not run:
# get a construction named FLOOR and all materials it uses
idf$objects_in_relation("floor", "ref_to")

# get a construction named FLOOR and all surfaces that uses it
idf$objects_in_relation("floor", "ref_by", class = "BuildingSurface:Detailed")

## End(Not run)


## ------------------------------------------------
## Method `Idf$search_object`
## ------------------------------------------------

## Not run:
# get all objects whose names contains "floor"
idf$search_object("floor", ignore.case = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$dup`
## ------------------------------------------------

## Not run:
# duplicate an object named "FLOOR"
idf$dup("floor") # New object name 'FLOOR_1' is auto-generated

# duplicate that object again by specifing object ID
idf$dup(16) # New object name 'FLOOR_2' is auto-generated

# duplicate that object two times and giving new names
idf$dup(new_floor = "floor", new_floor2 = 16)

# duplicate that object multiple times using variable inputs
floors_1 <- c(new_floor3 = "floor", new_floor4 = "floor")
floors_2 <- setNames(rep(16, 5), paste0("flr", 1:5))
idf$dup(floors_1, floors_2)

## End(Not run)
```

```
## ------------------------------------------------
## Method `Idf$add`
## ------------------------------------------------

## Not run:
# add a new Building object with all default values
empty <- empty_idf(8.8) # create an empty Idf
empty$add(Building = list())

# add a new Building object with all default values and comments
empty <- empty_idf(8.8) # create an empty Idf
empty$add(Building = list(.comment = c("this is", "a new building")))

# add a new RunPeriod object with all possible fields
empty <- empty_idf(8.8) # create an empty Idf
empty$add(Building = list(), RunPeriod = list("rp", 1, 1, 1, 31), .all = TRUE)

# add objects using variable inputs
empty <- empty_idf(8.8) # create an empty Idf
objs1 <- list(Schedule_Constant = list("const"), Building = list())
rp <- list(RunPeriod = list("rp", 2, 1, 2, 28))
empty$add(objs1, rp)

## End(Not run)


## ------------------------------------------------
## Method `Idf$set`
## ------------------------------------------------

## Not run:
# modify an object by name (case-insensitive)
idf$set(r13layer = list(roughness = "smooth"))

# modify an object by ID
idf$set(..12 = list(roughness = "rough"))

# overwrite existing object comments
idf$set(r13layer = list(.comment = c("New comment")))

# assign default values to fields
idf$set(r13layer = list(solar_absorptance = NULL), .default = TRUE)

# set field values to blanks
idf$set(r13layer = list(solar_absorptance = NULL), .default = FALSE)

# set field values to blank and delete trailing fields
idf$set(r13layer = list(visible_absorptance = NULL), .default = FALSE)

# set field values to blank and keep blank fields
idf$set(r13layer = list(visible_absorptance = NULL), .default = FALSE, .empty = TRUE)

# set all fields in one class
```

```
idf$set(Material_NoMass := list(visible_absorptance = 0.9))

# set multiple objects in one class
idf$set(.("r13layer", "r31layer") := list(solar_absorptance = 0.8))
# above is equivalent to
idf$set(r13layer = list(solar_absorptance = 0.8),
        r31layer = list(solar_absorptance = 0.8)
)

# use variable input
sets <- list(r13layer = list(roughness = "smooth"))
idf$set(sets)

## End(Not run)


## ------------------------------------------------
## Method `Idf$del`
## ------------------------------------------------

## Not run:
# delete objects using names
idf$object("Fraction") # ScheduleTypeLimits
idf$del("Fraction")

# delete objects using IDs
idf$objects(c(39, 40)) # Output:Variable
idf$del(39, 40)

# cannot delete objects that are referred by others
level_checks()$reference # reference-checking is enable by default
idf$del("r13layer") # error

# force to delete objects even thay are referred by others
idf$del("r13layer", .force = TRUE)

# delete objects and also objects that refer to them
idf$del("r31layer", .ref_by = TRUE) # Construction 'ROOF31' will be kept

# delete objects and also objects that they refer to
idf$del("extlights", .ref_to = TRUE) # Schedule 'AlwaysOn' will be kept

# delete objects and also other objects that refer to them recursively
idf$del("roof31", .ref_by = TRUE, .recursive = TRUE)

# delete objects using variable inputs
ids <- idf$object_id("Output:Variable", simplify = TRUE)
idf$del(ids)

## End(Not run)


## ------------------------------------------------
```

```
## Method `Idf$purge`
## ------------------------------------------------

## Not run:
# purge unused "Fraction" schedule type
idf$purge("on/off") # ScheduleTypeLimits

# purge all unused schedule types
idf$purge(class = "ScheduleTypeLimits")

# purge all unused schedule related objects
idf$purge(group = "Schedules")

## End(Not run)


## ------------------------------------------------
## Method `Idf$duplicated`
## ------------------------------------------------

## Not run:
# check if there are any duplications in the Idf
idf$duplicated(class = "ScheduleTypeLimits")

# check if there are any duplications in the schedule types
idf$duplicated(class = "ScheduleTypeLimits")

# check if there are any duplications in the schedule groups and
# material class
idf$duplicated(class = "Material", group = "Schedules")

## End(Not run)


## ------------------------------------------------
## Method `Idf$unique`
## ------------------------------------------------

## Not run:
# remove duplications in the Idf
idf$unique(class = "ScheduleTypeLimits")

# remove duplications in the schedule types
idf$unique(class = "ScheduleTypeLimits")

# remove duplications in the schedule groups and material class
idf$unique(class = "Material", group = "Schedules")

## End(Not run)


## ------------------------------------------------
## Method `Idf$rename`
```

```
## ------------------------------------------------

## Not run:
idf$objects(c("on/off", "test 352a"))
idf$rename(on_off = "on/off", test_352a = 51)

## End(Not run)


## ------------------------------------------------
## Method `Idf$insert`
## ------------------------------------------------

## Not run:
# insert all material from another IDF
path_idf2 <- file.path(eplus_config(8.8)$dir, "ExampleFiles/5ZoneTDV.idf")
idf2 <- Idf$new(path_idf2)
idf$insert(idf2$Material)

# insert objects from same Idf is equivalent to using Idf$dup()
idf$insert(idf$SizingPeriod_DesignDay)

## End(Not run)


## ------------------------------------------------
## Method `Idf$load`
## ------------------------------------------------

## Not run:
# load objects from character vectors
idf$load(
    c("Material,",
      "    mat,                     !- Name",
      "    MediumSmooth,            !- Roughness",
      "    0.667,                   !- Thickness {m}",
      "    0.115,                   !- Conductivity {W/m-K}",
      "    513,                     !- Density {kg/m3}",
      "    1381;                    !- Specific Heat {J/kg-K}"),

    "Construction, const, mat;"
)

# load objects from data.frame definitions
dt <- idf$to_table(class = "Material")
dt[field == "Name", value := paste(value, 1)]
dt[field == "Thickness", value := "0.5"]
idf$load(dt)

# by default, duplications are removed
idf$load(idf$to_table(class = "Material"))

# keep empty fields as they are
```

```
idf$load("Material, mat1, smooth, 0.5, 0.2, 500, 1000,,, 0.5;", .default = FALSE)

# keep trailing empty fields
idf$load("Material, mat2, smooth, 0.5, 0.2, 500, 1000,,,;",
    .default = FALSE, .empty = TRUE
)

## End(Not run)


## ------------------------------------------------
## Method `Idf$update`
## ------------------------------------------------

## Not run:
# update objects from string definitions:
str <- idf$to_string("zone one", header = FALSE, format = "new_top")
str[8] <- "2," # Multiplier
idf$update(str)

# update objects from data.frame definitions:
dt <- idf$to_table("zone one")
dt[field == "Multiplier", value := "1"]
idf$update(dt)

## End(Not run)


## ------------------------------------------------
## Method `Idf$search_value`
## ------------------------------------------------

## Not run:
# search values that contains "floor"
idf$search_value("floor", ignore.case = TRUE)

# search values that contains "floor" in class Construction
idf$search_value("floor", "Construction", ignore.case = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$replace_value`
## ------------------------------------------------

## Not run:
# search values that contains "win" and replace them with "windows"
idf$replace_value("win", "windows")

## End(Not run)
```

```
## ------------------------------------------------
## Method `Idf$validate`
## ------------------------------------------------

## Not run:
idf$validate()

# check at predefined validate level
idf$validate("none")
idf$validate("draft")
idf$validate("final")

# custom validate checking components
idf$validate(custom_validate(auto_field = TRUE, choice = TRUE))

## End(Not run)


## ------------------------------------------------
## Method `Idf$is_valid`
## ------------------------------------------------

## Not run:
idf$is_valid()

# check at predefined validate level
idf$is_valid("none")
idf$is_valid("draft")
idf$is_valid("final")

# custom validate checking components
idf$is_valid(custom_validate(auto_field = TRUE, choice = TRUE))

## End(Not run)


## ------------------------------------------------
## Method `Idf$to_string`
## ------------------------------------------------

## Not run:
# get text format of the whole Idf
head(idf$to_string())

# get text format of the whole Idf, excluding the header and all comments
head(idf$to_string(comment = FALSE, header = FALSE))

# get text format of all objects in class Material
head(idf$to_string(class = "Material", comment = FALSE, header = FALSE))

# get text format of some objects
head(idf$to_string(c("floor", "zone one")))
```

```
# tweak output formatting
head(idf$to_string("floor", leading = 0, sep_at = 0))

## End(Not run)


## ------------------------------------------------
## Method `Idf$to_table`
## ------------------------------------------------

## Not run:
# extract whole Idf data
idf$to_table()

# extract all data from class Material
idf$to_table(class = "Material")

# extract multiple object data
idf$to_table(c("FLOOR", "ZONE ONE"))

# keep value types and put actual values into a list column
idf$to_table(c("FLOOR", "ZONE ONE"), string_value = FALSE)$value

# add the unit to each value
idf$to_table(c("FLOOR", "ZONE ONE"), string_value = FALSE, unit = TRUE)

# get all possible fields
idf$to_table("ZONE ONE", all = TRUE)

# make sure all objects in same class have the same number of fields
idf$to_table(class = "Construction", align = TRUE)

# get a wide table with string values
idf$to_table(class = "Construction", wide = TRUE)

# get a wide table with actual values
idf$to_table(class = "OtherEquipment", wide = TRUE, string_value = FALSE)

# group extensible by extensible group number
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "group")

# group extensible by extensible group number and convert into a wide table
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "group", wide = TRUE)

# group extensible by extensible field index
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "index")

# group extensible by extensible field index and convert into a wide table
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "index", wide = TRUE)

# when grouping extensible, 'string_value' and 'unit' still take effect
idf$to_table(class = "BuildingSurface:Detailed", group_ext = "index",
    wide = TRUE, string_value = FALSE, unit = TRUE
```

```
)

# create table for new object input
idf$to_table(class = "BuildingSurface:Detailed", init = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$is_unsaved`
## ------------------------------------------------

## Not run:
idf$is_unsaved()

## End(Not run)


## ------------------------------------------------
## Method `Idf$save`
## ------------------------------------------------

## Not run:
# save Idf as a new file
idf$save(tempfile(fileext = ".idf"))

# save and overwrite current file
idf$save(overwrite = TRUE)

# save the model with newly created and modified objects at the top
idf$save(overwrite = TRUE, format = "new_top")

# save the model to a new file and copy all external csv files used in
# "Schedule:File" class into the same folder
idf$save(path = file.path(tempdir(), "test1.idf"), copy_external = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `Idf$run`
## ------------------------------------------------

## Not run:
idf <- Idf$new(path_idf)
# save the model to tempdir()
idf$save(file.path(tempdir(), "test_run.idf"))

# use the first epw file in "WeatherData" folder in EnergyPlus v8.8
# installation path
epw <- list.files(file.path(eplus_config(8.8)$dir, "WeatherData"),
    pattern = "\\.epw$", full.names = TRUE)[1]
```

```
# if `dir` is NULL, the directory of IDF file will be used as simulation
# output directory
job <- idf$run(epw, dir = NULL)

# run simulation in the background
idf$run(epw, dir = tempdir(), wait = FALSE)

# copy all external files into the directory run simulation
idf$run(epw, dir = tempdir(), copy_external = TRUE)

# check for simulation errors
job$errors()

# get simulation status
job$status()

# get output directory
job$output_dir()

# re-run the simulation
job$run()

# get simulation results
job$report_data()

## End(Not run)


## ------------------------------------------------
## Method `Idf$last_job`
## ------------------------------------------------

## Not run:
idf$last_job()

## End(Not run)


## ------------------------------------------------
## Method `Idf$geometry`
## ------------------------------------------------

## Not run:
idf$geometry()

## End(Not run)


## ------------------------------------------------
## Method `Idf$view`
## ------------------------------------------------

## Not run:
```

```
idf$view()
idf$view(render_by = "zone")
idf$view(render_by = "construction")

## End(Not run)


## ------------------------------------------------
## Method `Idf$print`
## ------------------------------------------------

## Not run:
idf$print("group")
idf$print("class")
idf$print("object")
idf$print("field")

# order objects by there classes
idf$print("object", order = FALSE)
idf$print("field", order = FALSE)

## End(Not run)
```

`IdfGeometry` *Modify and Visualize an EnergyPlus Model Geometry*

Description

`IdfGeometry` is an abstraction of a collection of geometry in an Idf. It provides more detail methods to query geometry properties, update geometry vertices and visualize geometry in 3D using the rgl package.

Usage

```
idf_geometry(parent, object = NULL)
```

Arguments

`parent`	A path to an IDF file or an Idf object.
`object`	A character vector of valid names or an integer vector of valid IDs of objects to extract. If `NULL`, all objects in geometry classes will be extracted.

Value

An `IdfGeometry` object.

Methods

Public methods:

- IdfGeometry$new()
- IdfGeometry$parent()
- IdfGeometry$rules()
- IdfGeometry$convert()
- IdfGeometry$coord_system()
- IdfGeometry$round_digits()
- IdfGeometry$area()
- IdfGeometry$azimuth()
- IdfGeometry$tilt()
- IdfGeometry$view()
- IdfGeometry$print()

Method `new()`**:** Create an `IdfGeometry` object

Usage:

```
IdfGeometry$new(parent, object = NULL)
```

Arguments:

`parent` A path to an IDF file or an Idf object.

`object` A character vector of valid names or an integer vector of valid IDs of objects to extract. If `NULL`, all objects in geometry classes will be extracted.

Returns: An `IdfGeometry` object.

Examples:

```
\dontrun{
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# create from an Idf object
idf <- read_idf(path_idf, use_idd(8.8, "auto"))
geom <- idf$geometry()
geom <- IdfGeometry$new(idf)

# create from an IDF file
geom <- idf_geometry(path_idf)
geom <- IdfGeometry$new(path_idf)
}
```

Method `parent()`**:** Get parent Idf object

Usage:

```
IdfGeometry$parent()
```

Details: $parent() returns the parent Idf object of current `IdfGeometry` object.

Returns: An Idf object.

Examples:

```
\dontrun{
geom$parent()
}
```

Method `rules()`**:** Get global geometry rules

Usage:

```
IdfGeometry$rules()
```

Details: $rules() returns global geometry rules.

Returns: An Idf object.

Examples:

```
\dontrun{
geom$rules()
}
```

Method `convert()`**:** Convert simple geometry objects

Usage:

```
IdfGeometry$convert(type = c("surface", "subsurface", "shading"))
```

Arguments:

`type` A character vector giving what types of simplified geometries should be converted. Should be a subset of `"surface"`, `"subsurface"` and `"shading"`. Default is set to all of them.

Details: EnergyPlus provides several classes that allow for simplified entry of geometries, such as `Wall:Exterior`, `Window` and etc. $convert() will generate detailed vertices from simplified geometry specifications and replace the original object with its corresponding detailed class, including:

- `BuildingSurface:Detailed`
- `FenestrationSurface:Detailed`
- `Shading:Site:Detailed`
- `Shading:Building:Detailed`
- `Shading:Zone:Detailed`

Returns: The modified Idf object.

Examples:

```
\dontrun{
geom$convert()
}
```

Method `coord_system()`**:** Convert vertices to specified coordinate systems

Usage:

```
IdfGeometry$coord_system(detailed = NULL, simple = NULL, daylighting = NULL)
```

Arguments:

`detailed`, `simple`, `daylighting` A string specifying the coordinate system for detailed geometries, simple (rectangular surface) geometries, and daylighting reference points. Should be one of `"relative"` and `"absolute"`.

Details: $coord_system() converts all vertices of geometries into specified coordinate systems, e.g. from absolute to relative, and vice versa. Besides, it also updates the `GlobalGeometryRules` in parent Idf accordingly.

Returns: The modified Idf object.

Examples:

```
\dontrun{
geom$coord_system("absolute", "absolute", "absolute")
}
```

Method `round_digits()`**:** Round digits on geometry vertices

Usage:

```
IdfGeometry$round_digits(digits = 4L)
```

Arguments:

`digits` An integer giving the number of decimal places to be used. Default: `4`.

Details: $round_digits() performs number rounding on vertices of detailed geometry object vertices, e.g. `BuildingSurface:Detailed`, `FenestrationSurface:Detailed` and etc. $round_digits() may be useful for clean up IDF files generated using OpenStudio which often gives vertices with long trailing digits.

Returns: The modified Idf object.

Examples:

```
\dontrun{
geom$round_digits()
}
```

Method `area()`**:** Get area

Usage:

```
IdfGeometry$area(class = NULL, object = NULL, net = FALSE)
```

Arguments:

`class` A character vector of valid geometry class names. Default: `NULL`.

`object` A character vector of valid names or an integer vector of valid IDs of targeting objects. Default: `NULL`.

`net` If `TRUE`, the gross area is returned. If `FALSE`, the net area is returned. Default: `FALSE`.

Details: $area() returns the area of surfaces in square meters.

Returns: A `data.table::data.table()` of 6 columns:

- `id`: Integer type. Object IDs.
- `name`: Character type. Object names.
- `class`: Character type. Class names.
- `zone`: Character type. Zone names that specified objects belong to.
- `type`: Character type. Surface types.
- `area`: Numeric type. Surface Area in m2.

Examples:

```
\dontrun{
geom$area()
}
```

Method `azimuth()`**:** Get azimuth

Usage:

```
IdfGeometry$azimuth(class = NULL, object = NULL)
```

Arguments:

`class` A character vector of valid geometry class names. Default: `NULL`.

`object` A character vector of valid names or an integer vector of valid IDs of targeting objects. Default: `NULL`.

Details: $azimuth() returns the azimuth of surfaces in degree.

Returns: A `data.table::data.table()` of 6 columns:

- `id`: Integer type. Object IDs.
- `name`: Character type. Object names.
- `class`: Character type. Class names.
- `zone`: Character type. Zone names that specified objects belong to.
- `type`: Character type. Surface types.
- `azimuth`: Numeric type. Azimuth in degree.

Examples:

```
\dontrun{
geom$azimuth()
}
```

Method `tilt()`**:** Get tilt

Usage:

```
IdfGeometry$tilt(class = NULL, object = NULL)
```

Arguments:

`class` A character vector of valid geometry class names. Default: `NULL`.

`object` A character vector of valid names or an integer vector of valid IDs of targeting objects. Default: `NULL`.

Details: $tilt() returns the tilt of surfaces in degree.

Returns: A `data.table::data.table()` of 6 columns:

- `id`: Integer type. Object IDs.
- `name`: Character type. Object names.
- `class`: Character type. Class names.
- `zone`: Character type. Zone names that specified objects belong to.
- `type`: Character type. Surface types.
- `tilt`: Numeric type. Azimuth in degree.

Examples:

```
\dontrun{
geom$tilt()
}
```

Method `view()`: View 3D geometry

Usage:

```
IdfGeometry$view(
  new = FALSE,
  render_by = "surface_type",
  wireframe = TRUE,
  x_ray = FALSE,
  axis = TRUE
)
```

Arguments:

`new` If `TRUE`, a new rgl window will be open using `rgl::rgl.open()`. If `FALSE`, existing rgl window will be reused if possible. Default: `FALSE`.

`render_by` A single string specifying the way of rendering the geometry. Possible values are:

- `"surface_type"`: Default. Render the model by surface type model. Walls, roofs, windows, doors, floors, and shading surfaces will have unique colors.
- `"boundary"`: Render the model by outside boundary condition. Only surfaces that have boundary conditions will be rendered with a color. All other surfaces will be white.
- `"construction"`: Render the model by surface constructions.
- `"zone"`: Render the model by zones assigned.
- `"normal"`: Render the model by surface normal. The outside face of a heat transfer face will be rendered as white and the inside face will be rendered as red.

`wireframe` If `TRUE`, the wireframe of each surface will be shown. Default: `TRUE`.

`x_ray` If `TRUE`, all surfaces wll be rendered translucently. Default: `FALSE`.

`axis` If `TRUE`, the X, Y and Z axes will be drawn at the global origin. Default: `TRUE`.

Details: $view() uses the rgl package to visualize the IDF geometry in 3D in a similar way as OpenStudio.
$view() returns an IdfViewer object which can be used to further tweak the viewer scene.
In the rgl window, you can control the view using your mouse:

- Left button: Trackball
- Right button: Pan projection.
- Wheel: Zoom

For more detailed control on the scene, see IdfViewer.

Returns: An IdfViewer object

Examples:

```
\dontrun{
idf$view()
idf$view(render_by = "zone")
idf$view(new = TRUE, render_by = "construction")
}
```

Method `print()`: Print an `IdfGeometry` object

Usage:

```
IdfGeometry$print()
```

Returns: The `IdfGeometry` itself, invisibly.

Examples:

```
\dontrun{
geom$print()
}
```

Author(s)

Hongyuan Jia

See Also

Idf class

Examples

```
## ------------------------------------------------
## Method `IdfGeometry$new`
## ------------------------------------------------

## Not run:
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# create from an Idf object
idf <- read_idf(path_idf, use_idd(8.8, "auto"))
geom <- idf$geometry()
geom <- IdfGeometry$new(idf)

# create from an IDF file
geom <- idf_geometry(path_idf)
geom <- IdfGeometry$new(path_idf)

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$parent`
## ------------------------------------------------

## Not run:
geom$parent()

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$rules`
## ------------------------------------------------

## Not run:
geom$rules()

## End(Not run)
```

```
## ------------------------------------------------
## Method `IdfGeometry$convert`
## ------------------------------------------------

## Not run:
geom$convert()

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$coord_system`
## ------------------------------------------------

## Not run:
geom$coord_system("absolute", "absolute", "absolute")

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$round_digits`
## ------------------------------------------------

## Not run:
geom$round_digits()

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$area`
## ------------------------------------------------

## Not run:
geom$area()

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$azimuth`
## ------------------------------------------------

## Not run:
geom$azimuth()

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$tilt`
## ------------------------------------------------

## Not run:
geom$tilt()
```

```
## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$view`
## ------------------------------------------------

## Not run:
idf$view()
idf$view(render_by = "zone")
idf$view(new = TRUE, render_by = "construction")

## End(Not run)

## ------------------------------------------------
## Method `IdfGeometry$print`
## ------------------------------------------------

## Not run:
geom$print()

## End(Not run)
```

`IdfObject` *Create and Modify an EnergyPlus Object*

Description

`IdfObject` is an abstraction of a single object in an Idf. It provides more detail methods to modify object values and comments. An `IdfObject` object can also be created using function `idf_object()` or from methods of a parent Idf object, using $object(), $objects_in_class() and equivalent.

Methods

Public methods:

- `IdfObject$new()`
- `IdfObject$version()`
- `IdfObject$parent()`
- `IdfObject$id()`
- `IdfObject$name()`
- `IdfObject$group_name()`
- `IdfObject$class_name()`
- `IdfObject$definition()`
- `IdfObject$comment()`
- `IdfObject$value()`
- `IdfObject$set()`
- `IdfObject$value_possible()`

- IdfObject$validate()
- IdfObject$is_valid()
- IdfObject$value_relation()
- IdfObject$ref_to_object()
- IdfObject$ref_by_object()
- IdfObject$ref_to_node()
- IdfObject$has_ref_to()
- IdfObject$has_ref_by()
- IdfObject$has_ref_node()
- IdfObject$has_ref()
- IdfObject$to_table()
- IdfObject$to_string()
- IdfObject$print()
- IdfObject$clone()

Method `new()`**:** Create an `IdfObject` object

Usage:

```
IdfObject$new(object, class = NULL, parent)
```

Arguments:

`object` An integer specifying an object ID.

`class` An integer specifying a class index.

`parent` An Idf object specifying the parent object.

Details: It is not recommended to directly use $new() method to create an `IdfObject` object, instead considering to use idf_object, `Idf$object()` and other equivalent to create `IdfObject` objects. They provide more user-friendly interfaces. $new() is a lower level API which is mainly used inside methods in other classes.

Returns: An `IdfObject` object.

Examples:

```
\dontrun{
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8
idf <- read_idf(path_idf, use_idd(8.8, "auto"))

roof <- IdfObject$new(26, parent = idf)

# get the IdfObject of material named "C5 - 4 IN HW CONCRETE"
mat <- idf$Material[["C5 - 4 IN HW CONCRETE"]]
}
```

Method `version()`**:** Get the version of parent `Idf`

Usage:

```
IdfObject$version()
```

Details: $version() returns the version of parent `Idf` in a `base::numeric_version()` format. This makes it easy to direction compare versions of different `IdfObjects`, e.g. `idfobj$version() > 8.6` or `idfobj1$version() > idfobj2$version()`.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
# get version
roof$version()
}
```

Method `parent()`**:** Get parent Idf

Usage:

```
IdfObject$parent()
```

Details: $parent() returns parent Idf object.

Returns: A Idf object.

Examples:

```
\dontrun{
roof$parent()
}
```

Method `id()`**:** Get the unique ID for current object

Usage:

```
IdfObject$id()
```

Details: In Idf, each object is assigned with an integer as an universally unique identifier (UUID) in the context of current Idf. UUID is not reused even if the object associated is deleted. $id() returns an integer of current object unique ID.

Returns: A single integer.

Examples:

```
\dontrun{
roof$id()
}
```

Method `name()`**:** Get the name for current object.

Usage:

```
IdfObject$name()
```

Details: In `Idf`, each object is assigned with a single string as the name for it, if the class it belongs to has name attribute, e.g. class `RunPeriod`, `Material` and etc. That name should be unique among all objects in that class. EnergyPlus will fail with an error if duplications are found among object names in a class.
$name() returns a single string of current object name. If specified class does not have name attribute, `NA` is returned.

Returns: A single string.

Examples:

```
\dontrun{
roof$name()

# NA will be returned if the class does not have name attribute. For example,
# "Version" class
idf$Version$name()
}
```

Method `group_name()`**:** Get name of group for current object.

Usage:

```
IdfObject$group_name()
```

Details: $group_name() returns a single string of group name current `IdfObject` belongs to.

Returns: A single string.

Examples:

```
\dontrun{
roof$group_name()
}
```

Method `class_name()`**:** Get name of class for current object.

Usage:

```
IdfObject$class_name()
```

Details: $class_name() returns a single string of class name current `IdfObject` belongs to.

Returns: A single string.

Examples:

```
\dontrun{
roof$class_name()
}
```

Method `definition()`**:** Get the IddObject object for current class.

Usage:

```
IdfObject$definition()
```

Details: $definition() returns an IddObject of current class. IddObject contains all data used for parsing and creating current `IdfObject`. For details, please see IddObject class.

Returns: An IddObject object.

Examples:

```
\dontrun{
roof$definition()
}
```

Method `comment()`**:** Get and modify object comments

Usage:

```
IdfObject$comment(comment, append = TRUE, width = 0L)
```

Arguments:

`comment` A character vector.

- If missing, current comments are returned. If there is no comment in current `IdfObject`, `NULL` is returned.
- If `NULL`, all comments in current `IdfObject` is deleted.
- If a character vector, it is inserted as comments depending on the `append` value.

`append` Only applicable when `commment` is a character vector. Default: `FALSE`.

- If `NULL`, existing comments is deleted before adding `comment`.
- If `TRUE`, comment will be appended to existing comments.
- If `FALSE`, `comment` is prepended to existing currents.

`width` A positive integer giving the target width for wrapping inserted `comment`.

Details: $comment() returns current `IdfObject` comments if `comment` is not given, or modifies current `IdfObject` comments if `comment` is given. If no comments found, `NULL` is returned.

Returns: If calling without any argument, a character vector or `NULL` (if no comments) is return. Otherwise, the modified object itself.

Examples:

```
\dontrun{
# get object comments
roof$comment()

# add new object comments
roof$comment(c("This is a material named `WD01`", "This object has an ID of 47"))
roof$comment()

# append new comments
roof$comment("This is an appended comment")
roof$comment()

# prepend new comments
roof$comment("This is a prepended comment", append = FALSE)
roof$comment()

# wrap long comments
roof$comment("This is a very long comment that is needed to be wrapped.", width = 30)
roof$comment()

# delete old comments and add new one
roof$comment("This is the only comment", append = NULL)
roof$comment()

# delete all comments
roof$comment(NULL)
```

```
roof$comment()
}
```

Method `value()`**:** Get object field values.

Usage:

```
IdfObject$value(which = NULL, all = FALSE, simplify = FALSE, unit = FALSE)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`all` If `TRUE`, values of all possible fields in current class the `IdfObject` belongs to are returned. Default: `FALSE`

`simplify` If `TRUE`, values of fields are converted into characters and the converted character vector is returned.

`unit` If `TRUE`, values of numeric fields are assigned with units using `units::set_units()` if applicable. Default: `FALSE`.

Details: $value() takes an integer vector of valid field indexes or a character vector of valid field names, and returns a named list containing values of specified fields when `simplify` is `FALSE` and a character vector when `simplify` is `TRUE`.
eplusr also provides custom S3 method of `$` and `[[` which make it more convenient to get a single value of current `IdfObject`. Basically, `idfobj$FieldName` and `idfobj[[Field]]` is equivalent to `idfobj$value(FieldName)[[1]]` and `idfobj$value(Field)[[1]]`.

Returns: A named list.

Examples:

```
\dontrun{
# get all existing field values
str(mat$value())

# get values of field 1, 3, 5
str(mat$value(c(1, 3, 5)))

# get character format values instead of a named list
mat$value(c(1, 3, 5), simplify = TRUE)

# get values of all field even those that are not set
str(roof$value())
str(roof$value(all = TRUE))

# get field values using shortcuts
mat$Roughness
mat[["Specific_Heat"]]
mat[c(1,2)]
mat[c("Name", "Density")]
}
```

Method `set()`**:** Modify object field values.

Usage:

```
IdfObject$set(..., .default = TRUE, .empty = FALSE)
```

Arguments:

`...` New field value definitions in `field = value` format or a single list in format:
`list(field1 = value1, field2 = value2)`

`.default` If `TRUE`, default values are used for those blank fields if possible. Default: `TRUE`.

`.empty` If `TRUE`, trailing empty fields are kept. Default: `FALSE`.

Details: $set() takes new field value definitions in `field = value` format or in a single list format, sets new values for fields specified, and returns the modified `IdfObject`. Unlike $set() method in Idf class, the special element `.comment` is **not allowed**. To modify object comments, please use $comment().

Examples:

```
\dontrun{
# set field values
mat$set(name = "new_name", Thickness = 0.02)
mat[c("Name", "Thickness")]

# When `default` argument is set to TRUE and input field values are empty, i.e.
# NULL, the field values will be reset to defaults.
mat[c("Thermal Absorptance", "Solar Absorptance")]

mat$set(visible_absorptance = NULL, Solar_Absorptance = NULL, .default = TRUE)
mat[c("Visible Absorptance", "Solar Absorptance")]

# set field values using shortcuts
mat$Name <- "another_name"
mat$Name
mat[["Thickness"]] <- 0.019
mat$Thickness
}
```

Method `value_possible()`**:** Get possible object field values.

Usage:

```
IdfObject$value_possible(
  which = NULL,
  type = c("auto", "default", "choice", "range", "source")
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`type` A character vector. What types of possible values should be returned. Should be one of or a combination of `"auto"`, `"default"`, `"choice"`, `"range"` and `"source"`. Default: All of those.

Details: $value_possible() takes an integer vector of valid field indexes or a character vector of valid field names, and returns all possible values for specified fields. For a specific field, there are 5 types of possible values:

- auto: Whether the field can be filled with Autosize and Autocalculate. This field attribute can also be retrieved using:
  idfobj$definition()$is_autosizable_field()
  idfobj$definition()$is_autocalculatable_field()
- default: The default value. This value can also be retrieved using idfobj$defintion()$field_default().
- choice: The choices which the field can be set. This value can also be retrieved using idfobj$definition()$field_choice().
- range: The range which the field value should fall in. This range can also be retrieved using idfobj$definition()$field_range().
- source: All values from other objects that current field can refer to.

Returns: $value_possible() returns an IdfValuePossible object which is a data.table with at most 15 columns:

- class_id: index of class that current IdfObject belongs to
- class_name: name of class that current IdfObject belongs to
- object_id: ID of current IdfObject
- object_name: name of current IdfObject
- field_id: indexes (at Idd level) of object fields specified
- field_index: indexes of object fields specified
- field_name: names (without units) of object fields specified
- value_id: value indexes (at Idf level) of object fields specified
- value_chr: values (converted to characters) of object fields specified
- value_num: values (converted to numbers in SI units) of object fields specified.
- auto: Exists only when "auto" is one of type. Character type. Possible values are: "Autosize", "Autocalculate" and NA (if current field is neither autosizable nor autocalculatable).
- default: Exists only when "default" is one of type. List type. The default value of current field. The value is converted into number if corresponding field type yells so. Note that if current field is a numeric field but the default value is "Autosize" or "Autocalculate", it is left as it is, leaving the returned type being a string instead of a number.
- range: Exists only when "range" is one of type. List type. The range that field value should fall in. Every range has four components: minimum (lower limit), lower_incbounds (TRUE if the lower limit should be included), maximum (upper limit), and upper_incbounds (TRUE if the upper limit should be included). For fields of character type, empty lists are returned. For fields of numeric types with no specified ranges, minimum is set to -Inf, lower_incbounds is set to FALSE, upper is set to Inf, and upper_incbounds is set to FALSE. The field range is printed in number interval denotation.
- source: Exists only when "source" is one of type. List type. Each element is a character vector which includes all values from other objects that current field can use as sources and refers to.

Examples:

```
\dontrun{
mat$value_possible()
}
```

Method validate()**:** Check possible object field value errors

Usage:

```
IdfObject$validate(level = eplusr_option("validate_level"))
```

Arguments:

`level` One of `"none"`, `"draft"`, `"final"` or a list of 10 elements with same format as `custom_validate()` output.

Details: $validate() checks if there are errors in current `IdfObject` object under specified validation level and returns an `IdfValidity` object.
$validate() is useful to help avoid some common errors before running the model. By default, validation is performed when calling all methods that modify objects, e.g. `$set()` and etc.
In total, there are 10 different validate checking components:

- `required_object`: Check if required objects are missing in current `Idf`.
- `unique_object`: Check if there are multiple objects in one unique-object class. An unique-object class means that there should be at most only one object existing in that class.
- `unique_name`: Check if all objects in each class have unique names.
- `extensible`: Check if all fields in an extensible group have values. An extensible group is a set of fields that should be treated as a whole, such like the X, Y and Z vertices of a building surfaces. An extensible group should be added or deleted together. `extensible` component checks if there are some, but not all, fields in an extensible group are empty.
- `required_field`: Check if all required fields have values.
- `auto_field`: Check if all fields filled with value `"Autosize"` and `"Autocalculate"` are actual autosizable and autocalculatable fields or not.
- `type`: Check if all fields have value types complied with their definitions, i.e. character, numeric and integer fields should be filled with corresponding type of values.
- `choice`: Check if all choice fields are filled with valid choice values.
- `range`: Check if all numeric fields have values within prescibed ranges.
- `reference`: Check if all fields whose values refer to other fields are valid.

The `level` argument controls what checkings should be performed. `level` here is just a list of 10 element which specify the toggle status of each component. You can use helper `custom_validate()` to get that list and pass it directly to `level`.
There are 3 predefined validate level that indicates different combinations of checking components, i.e. `none`, `draft` and `final`. Basically, `none` level just does not perform any checkings; `draft` includes 5 components, i.e. `auto_field`, `type`, `unique_name`, `choice` and `range`; and `final` level includes all 10 components. You can always get what components each level contains using `level_checks()`. By default, the result from `eplusr_option("validate_level")` is passed to `level`. If not set, `final` level is used.
Underneath, an `IdfValidity` object which $validate() returns is a list of 13 element as shown below. Each element or several elements represents the results from a single validation checking component.

- `missing_object`: Result of `required_object` checking.
- `duplicate_object`: Result of `unique_object` checking.
- `conflict_name`: Result of `unique_name` checking.
- `incomplete_extensible`: Result of `extensible` checking.
- `missing_value`: Result of `required_field` checking.
- `invalid_autosize`: Result of `auto_field` checking for invalid `Autosize` field values.

- `invalid_autocalculate`: Result of `auto_field` checking for invalid `Autocalculate` field values.
- `invalid_character`: Result of `type` checking for invalid character field values.
- `invalid_numeric`: Result of `type` checking for invalid numeric field values.
- `invalid_integer`: Result of `type` checking for invalid integer field values.
- `invalid_choice`: Result of `choice` checking.
- `invalid_range`: Result of `range` checking.
- `invalid_reference`: Result of `reference` checking.

Except `missing_object`, which is a character vector of class names that are missing, all other elements are data.table with 9 columns containing data of invalid field values:

- `object_id`: IDs of objects that contain invalid values
- `object_name`: names of objects that contain invalid values
- `class_id`: indexes of classes that invalid objects belong to
- `class_name`: names of classes that invalid objects belong to
- `field_id`: indexes (at Idd level) of object fields that are invalid
- `field_index`: indexes of object fields in corresponding that are invalid
- `field_name`: names (without units) of object fields that are invalid
- `units`: SI units of object fields that are invalid
- `ip_units`: IP units of object fields that are invalid
- `type_enum`: An integer vector indicates types of invalid fields
- `value_id`: indexes (at Idf level) of object field values that are invalid
- `value_chr`: values (converted to characters) of object fields that are invalid
- `value_num`: values (converted to numbers in SI units) of object fields that are invalid

Knowing the internal structure of `IdfValidity`, it is easy to extract invalid IdfObjects you interested in. For example, you can get all IDs of objects that contain invalid value references using `model$validate()$invalid_reference$object_id`. Then using `$set()` method to correct them.

Different validate result examples are shown below:

- No error is found:

  ```
  v No error found.
  ```

  Above result shows that there is no error found after conducting all validate checks in specified validate level.

- Errors are found:

  ```
   x [2] Errors found during validation.
  ==========================================================================

  -- [2] Invalid Autocalculate Field ---------------------------------------
     Fields below cannot be `autocalculate`:

      Class: <AirTerminal:SingleDuct:VAV:Reheat>
      \- Object [ID:176] <SPACE5-1 VAV Reheat>
       +- 17: AUTOCALCULATE, !- Maximum Flow per Zone Floor Area During Reheat {m3/s-m2}
          \- 18: AUTOCALCULATE; !- Maximum Flow Fraction During Reheat
  ```

Above result shows that after all validate components performed under current validate level, 2 invalid field values are found. All of them are in a object named SPACE5-1 VAV Reheat with ID 176. They are invalid because those two fields do not have an autocalculatable attribute but are given AUTOCALCULATE value. Knowing this info, one simple way to fix the error is to correct those two fields by doing:

```
idf$set(..176 =
    list(`Maximum Flow per Zone Floor Area During Reheat` = "autosize",
         `Maximum Flow Fraction During Reheat` = "autosize"
    )
)
```

Returns: An IdfValidity object.

Examples:

```
\dontrun{
mat$validate()

# check at predefined validate level
mat$validate("none")
mat$validate("draft")
mat$validate("final")

# custom validate checking components
mat$validate(custom_validate(auto_field = TRUE, choice = TRUE))
}
```

Method `is_valid()`**:** Check if there is any error in current object

Usage:

```
IdfObject$is_valid(level = eplusr_option("validate_level"))
```

Arguments:

`level` One of `"none"`, `"draft"`, `"final"` or a list of 10 elements with same format as `custom_validate()` output.

Details: $is_valid() returns TRUE if there is no error in current IdfObject object under specified validation level and FALSE otherwise.
$is_valid() checks if there are errors in current IdfObject object under specified validation level and returns TRUE or FALSE accordingly. For detailed description on validate checking, see `$validate()` documentation above.

Returns: A single logical value of TRUE or FALSE.

Examples:

```
\dontrun{
mat$is_valid()

mat$definition()$field_range("Density")
eplusr_option(validate_level = "none") # have to set validate to "none" to do so
mat$Density <- -1
eplusr_option(validate_level = "final") # change back to "final" validate level
```

```
mat$is_valid()

# check at predefined validate level
mat$is_valid("none")
mat$is_valid("draft")
mat$is_valid("final")

# custom validate checking components
mat$is_valid(custom_validate(auto_field = TRUE, choice = TRUE))
}
```

Method `value_relation()`**:** Get value relations

Usage:

```
IdfObject$value_relation(
  which = NULL,
  direction = c("all", "ref_to", "ref_by", "node"),
  object = NULL,
  class = NULL,
  group = NULL,
  depth = 0L,
  keep = FALSE,
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`direction` The relation direction to extract. Should be either `"all"`, `"ref_to"` or "ref_by".

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

`keep` If `TRUE`, all input fields are returned regardless they have any relations with other objects or not. If `FALSE`, only fields in input that have relations with other objects are returned. Default: `FALSE`.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in `Branch` objects. Their value refers to other many class names of objects, instaed of referring to specific field values. There are 3 options in total, i.e. `"none"`, `"both"` and `"all"`, with `"both"` being the default. * `"none"`: just ignore class-name-references. It is a reasonable option, as for most cases, class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * `"both"`: only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, only the object

that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * `"all"`: include all class-name-references. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, all objects in `Coil:Heating:Water` will be treated as referenced by that field. This is the most aggressive option.

Details: Many fields in Idd can be referred by others. For example, the Outside Layer and other fields in `Construction` class refer to the `Name` field in `Material` class and other material related classes. Here it means that the Outside Layer field **refers to** the `Name` field and the `Name` field is **referred by** the Outside Layer. In EnergyPlus, there is also a special type of field called `Node`, which together with `Branch` and `BranchList` define the topography of the HVAC connections. A outlet node of a component can be referred by another component as its inlet node, but can also exists independently, such as zone air node.
$value_relation() provides a simple interface to get this kind of relation. It takes field indexes or field names, together a relation direction, and returns an `IdfRelation` object which contains data presenting such relation described above. For instance, if `idfobj$value_relation("Name","ref_by")` gives results below:

```
-- Referred by Others ------------------------
  \- 1: "WALL-1";      !- Name
     ^~~~~~~~~~~~~~~~~~~~~~~
     \- Class: <BuildingSurface:Detailed>
        \- Object [ID:3] <WALL-1PF>
           \- 3: "WALL-1";      !- Construction Name
```

This means that the value `"WALL-1"` of field `Name` is referred by field Construction Name in a surface named WALL-1PF. All those objects can be further easily extracted using $ref_by_object() method.
Note that $value_relation() shows all fields specified, even some of them may do not have relation.

Returns: An `IdfRelation` object, which is a list of 3 `data.table::data.table()`s named `ref_to`, `ref_by` and node. Each `data.table::data.table()` contains 24 columns.

Examples:

```
\dontrun{
# check each layer's reference of a construction named FLOOR
roof$value_relation("zone name", "ref_to")

# check where is this construction being used
roof$value_relation("name", direction = "ref_by")
}
```

Method `ref_to_object()`**:** Extract multiple `IdfObject` objects referred by specified field values

Usage:

```
IdfObject$ref_to_object(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
```

```
  depth = 0L,
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in `Branch` objects. Their value refers to other many class names of objects, instaed of referring to specific field values. There are 3 options in total, i.e. `"none"`, `"both"` and `"all"`, with `"both"` being the default. * `"none"`: just ignore class-name-references. It is a reasonable option, as for most cases, class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * `"both"`: only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, only the object that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * `"all"`: include all class-name-references. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, all objects in `Coil:Heating:Water` will be treated as referenced by that field. This is the most aggressive option.

Details: For details on field value relations, see `$value_relation()`.
$ref_to_object() takes an integer vector of field indexes or a character vector of field names, and returns a list of `IdfObjects` that specified fields refer to.

Returns: A named list of `IdfObject` objects.

Examples:

```
\dontrun{
# get other objects that this object refereces
mat$ref_to_object() # not referencing other objects
}
```

Method `ref_by_object()`**:** Extract multiple `IdfObject` objects referring to specified field values

Usage:

```
IdfObject$ref_by_object(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
  depth = 0L,
```

```
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in `Branch` objects. Their value refers to other many class names of objects, instaed of referring to specific field values. There are 3 options in total, i.e. `"none"`, `"both"` and `"all"`, with `"both"` being the default. * `"none"`: just ignore class-name-references. It is a reasonable option, as for most cases, class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * `"both"`: only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, only the object that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * `"all"`: include all class-name-references. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, all objects in `Coil:Heating:Water` will be treated as referenced by that field. This is the most aggressive option.

Details: For details on field value relations, see `$value_relation()`.
$ref_by_object() takes an integer vector of field indexes or a character vector of field names, and returns a list of `IdfObject`s that refer to specified fields.

Returns: A named list of `IdfObject` objects.

Examples:

```
\dontrun{
# get other objects that reference this object
mat$ref_by_object() # referenced by construction "FLOOR"
}
```

Method `ref_to_node()`**:** Extract multiple `IdfObject` objects referring to same nodes

Usage:

```
IdfObject$ref_to_node(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
  depth = 0L
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`depth` If > 0, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. If `NULL`, all possible recursive relations are returned. Default: `0`.

Details: For details on field value relations, see `$value_relation()`.
$ref_to_node() takes an integer vector of field indexes or a character vector of field names, and returns a list of `IdfObjects` whose nodes are referred by specified fields.

Returns: A named list of `IdfObject` objects.

Examples:

```
\dontrun{
if (is_avail_eplus(8.8)) {
   path <- file.path(eplus_config(8.8)$dir, "ExampleFiles/5Zone_Transformer.idf")
    idf_5z <- read_idf(path)
    idf_5z$NodeList$OutsideAirInletNodes$ref_to_node()
}
}
```

Method `has_ref_to()`**:** Check if object field values refer to others

Usage:

```
IdfObject$has_ref_to(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
  recursive = FALSE,
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`recursive` If `TRUE`, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. Default: `FALSE`.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in `Branch` objects. Their value refers to other many class names of objects, instaed of referring to specific field values. There

are 3 options in total, i.e. "none", "both" and "all", with "both" being the default. * "none": just ignore class-name-references. It is a reasonable option, as for most cases, class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * "both": only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is Coil:Heating:Water, only the object that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * "all": include all class-name-references. For example, if the value of field Component 1 Object Type is Coil:Heating:Water, all objects in Coil:Heating:Water will be treated as referenced by that field. This is the most aggressive option.

Details: For details on field value relations, see $value_relation().
$has_ref_to() takes an integer vector of field indexes or a character vector of field names, and returns a logical vector showing whether specified fields refer to other object values or not.

Returns: A logical vector with the same length as specified field.

Examples:

```
\dontrun{
mat$has_ref_to()
}
```

Method `has_ref_by()`**:** Check if object field values are referred by others

Usage:

```
IdfObject$has_ref_by(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
  recursive = FALSE,
  class_ref = c("both", "none", "all")
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: NULL.

`class` A character vector of class names used for searching relations. Default: NULL.

`group` A character vector of group names used for searching relations. Default: NULL.

`recursive` If TRUE, the relation is searched recursively. A simple example of recursive reference: one material named mat is referred by a construction named const, and const is also referred by a surface named surf. Default: FALSE.

`class_ref` Specify how to handle class-name-references. Class name references refer to references in like field Component 1 Object Type in Branch objects. Their value refers to other many class names of objects, instaed of referring to specific field values. There are 3 options in total, i.e. "none", "both" and "all", with "both" being the default. * "none": just ignore class-name-references. It is a reasonable option, as for most cases,

class-name-references always come along with field value references. Ignoring class-name-references will not impact the most part of the relation structure. * `"both"`: only include class-name-references if this object also reference field values of the same one. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, only the object that is referenced in the next field Component 1 Name is treated as referenced by Component 1 Object Type. This is the default option. * `"all"`: include all class-name-references. For example, if the value of field Component 1 Object Type is `Coil:Heating:Water`, all objects in `Coil:Heating:Water` will be treated as referenced by that field. This is the most aggressive option.

Details: For details on field value relations, see `$value_relation()`.
$has_ref_by() takes an integer vector of field indexes or a character vector of field names, and returns a logical vector showing whether there are other object values ref to specified fields.

Returns: A logical vector with the same length as specified field.

Examples:

```
\dontrun{
mat$has_ref_by()
}
```

Method `has_ref_node()`**:** Check if object field values refer to other nodes

Usage:

```
IdfObject$has_ref_node(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
  recursive = FALSE
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`recursive` If `TRUE`, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. Default: `FALSE`.

Details: For details on field value relations, see `$value_relation()`.
$has_ref_node() takes an integer vector of field indexes or a character vector of field names, and returns a logical vector showing whether specified fields refer to other objects' nodes.

Returns: A logical vector with the same length as specified field.

Examples:

```
\dontrun{
mat$has_ref_node()
}
```

Method `has_ref()`**:** Check if object field values refer to or are referred by others

Usage:

```
IdfObject$has_ref(
  which = NULL,
  object = NULL,
  class = NULL,
  group = NULL,
  recursive = FALSE
)
```

Arguments:

`which` An integer vector of field indexes or a character vector of field names.

`object` A character vector of object names or an integer vector of object IDs used for searching relations. Default: `NULL`.

`class` A character vector of class names used for searching relations. Default: `NULL`.

`group` A character vector of group names used for searching relations. Default: `NULL`.

`recursive` If `TRUE`, the relation is searched recursively. A simple example of recursive reference: one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`. Default: `FALSE`.

Details: For details on field value relations, see `$value_relation()`.
$has_ref() takes an integer vector of field indexes or a character vector of field names, and returns a logical vector showing whether there are other object values ref to specified field values or specified field values refer to other object values or specified field values refer to other objects' nodes.

Returns: A logical vector with the same length as specified field.

Examples:

```
\dontrun{
# check if having any referenced objects or is referenced by other objects
mat$has_ref()
}
```

Method `to_table()`**:** Format `IdfObject` as a data.frame

Usage:

```
IdfObject$to_table(
  string_value = TRUE,
  unit = TRUE,
  wide = FALSE,
  all = FALSE,
  group_ext = c("none", "group", "index")
)
```

Arguments:

`string_value` If `TRUE`, all field values are returned as character. If `FALSE`, `value` column in returned data.table is a list column with each value stored as corresponding type. Note that if the value of numeric field is set to `"Autosize"` or `"Autocalculate"`, it is left as it is, leaving the returned type being a string instead of a number. Default: `TRUE`.

`unit` Only applicable when `string_value` is `FALSE`. If `TRUE`, values of numeric fields are assigned with units using `units::set_units()` if applicable. Default: `FALSE`.

`wide` Only applicable if target objects belong to a same class. If `TRUE`, a wide table will be returned, i.e. first three columns are always `id`, `name` and `class`, and then every field in a separate column. Note that this requires all objects specified must from the same class. Default: `FALSE`.

`all` If `TRUE`, all available fields defined in IDD for the class that objects belong to will be returned. Default: `FALSE`.

`group_ext` Should be one of `"none"`, `"group"` or `"index"`. If not `"none"`, `value` column in returned `data.table::data.table()` will be converted into a list. If `"group"`, values from extensible fields will be grouped by the extensible group they belong to. For example, coordinate values of each vertex in class `BuildingSurface:Detailed` will be put into a list. If `"index"`, values from extensible fields will be grouped by the extensible field indice they belong to. For example, coordinate values of all x coordinates will be put into a list. If `"none"`, nothing special will be done. Default: `"none"`.

Details: $to_table() returns a data.table that contains core data of current `IdfObject`. It has 6 columns:

- `id`: Integer type. Object IDs.
- `name`: Character type. Object names.
- `class`: Character type. Current class name.
- `index`: Integer type. Field indexes.
- `field`: Character type. Field names.
- `value`: Character type if `string_value` is `TRUE` or list type if `string_value` is `FALSE` or `group_ext` is not `"none"`. Field values.

Note that when `group_ext` is not `"none"`, `index` and `field` values will not match the original field indices and names. In this case, `index` will only indicate the indices of sequences. For `field` column, specifically:

- When `group_ext` is `"group"`, each field name in a extensible group will be abbreviated using `abbreviate()` with `minlength` being `10L` and all abbreviated names will be separated by `|` and combined together. For example, field names in the extensible group (Vertex 1 X-coordinate, Vertex 1 Y-coordinate, Vertex 1 Z-coordinate) in class `BuildiBuildingSurface:Detailed` will be merged into one name `Vrtx1X-crd|Vrtx1Y-crd|Vrtx1Z-crd`.
- When `group_ext` is `"index"`, the extensible group indicator in field names will be removed. Take the same example as above, the resulting field names will be Vertex X-coordinate, Vertex Y-coordinate, and Vertex Z-coordinate.

Returns: A data.table with 6 columns (if `wide` is `FALSE`) or at least 6 columns (if `wide` is `TRUE`).

Examples:

```
\dontrun{
# get all object data in a data.table format without field units
str(mat$to_table(unit = FALSE))

# get all object data in a data.table format where all field values are put in a
# list column and field names without unit
str(mat$to_table(string_value = FALSE, unit = FALSE))
```

```
# get all object data in a data.table format, including tailing empty fields
str(idf$Zone$`ZONE ONE`$to_table(all = TRUE))

# get all object data in a data.table format where each field becomes a column
str(mat$to_table(wide = TRUE))

# group extensible by extensible group number
surf <- idf$BuildingSurface_Detailed[["Zn001:Roof001"]]
surf$to_table(group_ext = "group")

# group extensible by extensible group number and convert into a wide table
surf$to_table(group_ext = "group", wide = TRUE)

# group extensible by extensible field index
surf$to_table(group_ext = "index")

# group extensible by extensible field index and convert into a wide table
surf$to_table(group_ext = "index", wide = TRUE)

# when grouping extensible, 'string_value' and 'unit' still take effect
surf$to_table(group_ext = "index", wide = TRUE, string_value = FALSE, unit = TRUE)
}
```

Method `to_string()`**:** Format current object as a character vector

Usage:

```
IdfObject$to_string(comment = TRUE, leading = 4L, sep_at = 29L, all = FALSE)
```

Arguments:

`comment` If `FALSE`, all comments will not be included. Default: `TRUE`.

`leading` Leading spaces added to each field. Default: `4L`.

`sep_at` The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.

`all` If `TRUE`, all available fields defined in IDD for the class that objects belong to will be returned. Default: `FALSE`.

Details: $to_string() returns the text format of current object.

Returns: A character vector.

Examples:

```
\dontrun{
# get string format object
mat$to_string()

# get string format of object, and decrease the space between field values and
# field names
mat$to_string(sep_at = 15)

# get string format of object, and decrease the leading space of field values
```

```
mat$to_string(leading = 0)
}
```

Method `print()`**:** Print `IdfObject` object

Usage:

```
IdfObject$print(comment = TRUE, auto_sep = TRUE, brief = FALSE, all = FALSE)
```

Arguments:

`comment` If `FALSE`, all comments are not included.

`auto_sep` If `TRUE`, values and field names are separated at the largest character length of values. Default: `FALSE`.

`brief` If `TRUE`, only OBJECT part is printed. Default: `FALSE`.

`all` If `TRUE`, all fields defined in Idd are printed even they do not exist in current object. Default: `FALSE`.

Details: $print() prints the `IdfObject`. Basically, the print output can be divided into 3 parts:

- OBJECT: Class name, object id and name (if applicable).
- COMMENTS: Object comments if exist.
- VALUES: fields and values of current `IdfObject`. Required fields are marked with start `*`. String values are quoted. Numeric values are printed as they are. Blank string values are printed as <"Blank"> and blank number values are printed as <Blank>.

Returns: The `IdfObject` itself, invisibly.

Examples:

```
\dontrun{
# print the object without comment
mat$print(comment = FALSE)

# print the object, and auto separate field values and field names at the
# largetst character length of field values
mat$print(auto_sep = TRUE)
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
IdfObject$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

- Only one single list is allowed, e.g. `idfobj$set(lst1)` where `lst1 <-list(field1 = value1)` is allowed, but `idfobj$set(lst1,lst2)` is not.

- You can delete a field by assigning NULL to it, e.g. `iddobj$set(fld = NULL)` means to delete the value of field `fld`. If `.default` is FALSE, also `fld` is not a required field and the index of `fld` is larger than the number minimum fields required for that class, it will be deleted. Otherwise it will be left as blank. If `.default` is TRUE, that field will be filled with default value if applicable and left as blank if not.
- By default, trailing empty fields that are not required will be removed and only minimum required fields are kept. You can keep the trailing empty fields by setting `.empty` to TRUE.
- New fields that currently do not exist in that object can also be set. They will be automatically added on the fly.
- Field name matching is **case-insensitive**. For convenience, underscore-style field names are also allowed, e.g. `eNd_MoNtH` is equivalent to End Month.
- If not all field names are given, positions of those values without field names are determined after those values with names. E.g. in `model$set(Construction = list("out_layer",name = "name"))`, `"out_layer"` will be treated as the value of field Outside Layer in `Construction`, as value of field `Name` has been given as `"name"`.

eplusr also provides custom S3 method of $<- and [[<- which makes it more convenient to set a single field value of an `IdfObject`. Basically, `idfobj$FieldName <-value` and `idfobj[[Field]] <-value` is equivalent to `idfobj$set(FieldName = value)` and `idfobjset(Field = value)`.

Author(s)

Hongyuan Jia

See Also

Idf class

Examples

```
## ------------------------------------------------
## Method `IdfObject$new`
## ------------------------------------------------

## Not run:
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8
idf <- read_idf(path_idf, use_idd(8.8, "auto"))

roof <- IdfObject$new(26, parent = idf)

# get the IdfObject of material named "C5 - 4 IN HW CONCRETE"
mat <- idf$Material[["C5 - 4 IN HW CONCRETE"]]

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$version`
```

```
## ------------------------------------------------

## Not run:
# get version
roof$version()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$parent`
## ------------------------------------------------

## Not run:
roof$parent()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$id`
## ------------------------------------------------

## Not run:
roof$id()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$name`
## ------------------------------------------------

## Not run:
roof$name()

# NA will be returned if the class does not have name attribute. For example,
# "Version" class
idf$Version$name()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$group_name`
## ------------------------------------------------

## Not run:
roof$group_name()

## End(Not run)
```

```
## ------------------------------------------------
## Method `IdfObject$class_name`
## ------------------------------------------------

## Not run:
roof$class_name()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$definition`
## ------------------------------------------------

## Not run:
roof$definition()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$comment`
## ------------------------------------------------

## Not run:
# get object comments
roof$comment()

# add new object comments
roof$comment(c("This is a material named `WD01`", "This object has an ID of 47"))
roof$comment()

# append new comments
roof$comment("This is an appended comment")
roof$comment()

# prepend new comments
roof$comment("This is a prepended comment", append = FALSE)
roof$comment()

# wrap long comments
roof$comment("This is a very long comment that is needed to be wrapped.", width = 30)
roof$comment()

# delete old comments and add new one
roof$comment("This is the only comment", append = NULL)
roof$comment()

# delete all comments
roof$comment(NULL)
roof$comment()

## End(Not run)
```

```
## ------------------------------------------------
## Method `IdfObject$value`
## ------------------------------------------------

## Not run:
# get all existing field values
str(mat$value())

# get values of field 1, 3, 5
str(mat$value(c(1, 3, 5)))

# get character format values instead of a named list
mat$value(c(1, 3, 5), simplify = TRUE)

# get values of all field even those that are not set
str(roof$value())
str(roof$value(all = TRUE))

# get field values using shortcuts
mat$Roughness
mat[["Specific_Heat"]]
mat[c(1,2)]
mat[c("Name", "Density")]

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$set`
## ------------------------------------------------

## Not run:
# set field values
mat$set(name = "new_name", Thickness = 0.02)
mat[c("Name", "Thickness")]

# When `default` argument is set to TRUE and input field values are empty, i.e.
# NULL, the field values will be reset to defaults.
mat[c("Thermal Absorptance", "Solar Absorptance")]

mat$set(visible_absorptance = NULL, Solar_Absorptance = NULL, .default = TRUE)
mat[c("Visible Absorptance", "Solar Absorptance")]

# set field values using shortcuts
mat$Name <- "another_name"
mat$Name
mat[["Thickness"]] <- 0.019
mat$Thickness

## End(Not run)
```

```
## ------------------------------------------------
## Method `IdfObject$value_possible`
## ------------------------------------------------

## Not run:
mat$value_possible()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$validate`
## ------------------------------------------------

## Not run:
mat$validate()

# check at predefined validate level
mat$validate("none")
mat$validate("draft")
mat$validate("final")

# custom validate checking components
mat$validate(custom_validate(auto_field = TRUE, choice = TRUE))

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$is_valid`
## ------------------------------------------------

## Not run:
mat$is_valid()

mat$definition()$field_range("Density")
eplusr_option(validate_level = "none") # have to set validate to "none" to do so
mat$Density <- -1
eplusr_option(validate_level = "final") # change back to "final" validate level
mat$is_valid()

# check at predefined validate level
mat$is_valid("none")
mat$is_valid("draft")
mat$is_valid("final")

# custom validate checking components
mat$is_valid(custom_validate(auto_field = TRUE, choice = TRUE))

## End(Not run)
```

```
## ------------------------------------------------
## Method `IdfObject$value_relation`
## ------------------------------------------------

## Not run:
# check each layer's reference of a construction named FLOOR
roof$value_relation("zone name", "ref_to")

# check where is this construction being used
roof$value_relation("name", direction = "ref_by")

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$ref_to_object`
## ------------------------------------------------

## Not run:
# get other objects that this object refereces
mat$ref_to_object() # not referencing other objects

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$ref_by_object`
## ------------------------------------------------

## Not run:
# get other objects that reference this object
mat$ref_by_object() # referenced by construction "FLOOR"

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$ref_to_node`
## ------------------------------------------------

## Not run:
if (is_avail_eplus(8.8)) {
    path <- file.path(eplus_config(8.8)$dir, "ExampleFiles/5Zone_Transformer.idf")
    idf_5z <- read_idf(path)
    idf_5z$NodeList$OutsideAirInletNodes$ref_to_node()
}

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$has_ref_to`
## ------------------------------------------------
```

```
## Not run:
mat$has_ref_to()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$has_ref_by`
## ------------------------------------------------

## Not run:
mat$has_ref_by()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$has_ref_node`
## ------------------------------------------------

## Not run:
mat$has_ref_node()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$has_ref`
## ------------------------------------------------

## Not run:
# check if having any referenced objects or is referenced by other objects
mat$has_ref()

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$to_table`
## ------------------------------------------------

## Not run:
# get all object data in a data.table format without field units
str(mat$to_table(unit = FALSE))

# get all object data in a data.table format where all field values are put in a
# list column and field names without unit
str(mat$to_table(string_value = FALSE, unit = FALSE))

# get all object data in a data.table format, including tailing empty fields
str(idf$Zone$`ZONE ONE`$to_table(all = TRUE))
```

```
# get all object data in a data.table format where each field becomes a column
str(mat$to_table(wide = TRUE))

# group extensible by extensible group number
surf <- idf$BuildingSurface_Detailed[["Zn001:Roof001"]]
surf$to_table(group_ext = "group")

# group extensible by extensible group number and convert into a wide table
surf$to_table(group_ext = "group", wide = TRUE)

# group extensible by extensible field index
surf$to_table(group_ext = "index")

# group extensible by extensible field index and convert into a wide table
surf$to_table(group_ext = "index", wide = TRUE)

# when grouping extensible, 'string_value' and 'unit' still take effect
surf$to_table(group_ext = "index", wide = TRUE, string_value = FALSE, unit = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$to_string`
## ------------------------------------------------

## Not run:
# get string format object
mat$to_string()

# get string format of object, and decrease the space between field values and
# field names
mat$to_string(sep_at = 15)

# get string format of object, and decrease the leading space of field values
mat$to_string(leading = 0)

## End(Not run)


## ------------------------------------------------
## Method `IdfObject$print`
## ------------------------------------------------

## Not run:
# print the object without comment
mat$print(comment = FALSE)

# print the object, and auto separate field values and field names at the
# largetst character length of field values
mat$print(auto_sep = TRUE)

## End(Not run)
```

IdfSchedule *Create an* IdfScheduleCompact *object.*

Description

schedule_compact() takes a parent Idf object, a name for Schedule:Compact object, and returns a corresponding IdfScheduleCompact object.

IdfScheduleCompact is an abstraction of a single Schedule:Compact object in an Idf. It provides more detailed methods to add, modify and extract schedule values.

Usage

```
schedule_compact(parent, name, new = FALSE)
```

Arguments

parent	An Idf object.
name	A valid name (a string) for a Schedule:Compact object.
new	If TRUE, a new empty IdfScheduleCompact is created. Default: FALSE.

Details

If new is TRUE, an empty IdfScheduleCompact is created, with all field values being empty. The empty IdfScheduleCompact is directly added into the parent Idf object. It is recommended to use $validate() method in IdfScheduleCompact to see what kinds of further modifications are needed for those empty fields and use $set() and $update() method to set field values.

Value

An IdfScheduleCompact object.

Super classes

eplusr::IdfObject -> eplusr::IdfSchedule -> IdfScheduleCompact

Methods

Public methods:

- IdfScheduleCompact$new()
- IdfScheduleCompact$type_limits()
- IdfScheduleCompact$set()
- IdfScheduleCompact$update()
- IdfScheduleCompact$validate()
- IdfScheduleCompact$is_valid()

- IdfScheduleCompact$extract()
- IdfScheduleCompact$clone()

Method `new()`**:** Create an `IdfScheduleCompact` object

Usage:

```
IdfScheduleCompact$new(object, parent, new = FALSE)
```

Arguments:

`object` An integer specifying an object ID.

`parent` An Idf object specifying the parent object.

`new` If `TRUE`, an empty `IdfScheduleCompact` will be created. Default: `FALSE`

Returns: An `IdfScheduleCompact` object.

Examples:

```
\dontrun{
model <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"))

# create an empty 'Schedule:Compact'
schedule_compact(model, "sch", TRUE)

# get an existing 'Schedule:Compact'
sch <- schedule_compact(model, "sch")
}
```

Method `type_limits()`**:** Get or set the schedule type limits

Usage:

```
IdfScheduleCompact$type_limits(name)
```

Arguments:

`name` A string specifying the name of an `ScheduleTypeLimits` object in current Idf. If missing, value of last time will be used.

Returns: A list of 2 elements:

- `name`: The name of the `ScheduleTypeLimits` object
- `range`: The range of current schedule values

Method `set()`**:** Set schedule values

Usage:

```
IdfScheduleCompact$set(..., .check_range = TRUE)
```

Arguments:

`...` Schedule day types and value specifications in lists.

- Group day types inside `c(...)` at the LHS of :=
- Put actual schedule values inside `list(...)` at the RHS of :=
- Each schedule value should be named a time. Time can be given in either `..H` or `"HH:MM"`.

`.check_range` If `TRUE`, schedule values will be checked based on $type_limits(). Default: `TRUE`.

Details: Please note that all schedules will be applied for all days in a year. For detailed modifications, please check $update() method which accepts data.frame input.

Returns: The modified `IdfScheduleCompact` object.

Examples:

```
\dontrun{
sch$set(c("weekday", "summerdesignday") := list(
    ..6 = 0.2, "8:00" = 0.5,
    ..12 = 0.95, "13:30" = 0.6, ..14 = 0.8,
    ..18 = 0.95, ..19 = 0.2, ..24 = 0),
    allotherday = list(..24 = 0)
)
}
```

Method `update()`**:** Update schedule values using data.frame

Usage:

```
IdfScheduleCompact$update(data, check_range = TRUE, compact = FALSE)
```

Arguments:

`data` A data.frame of at least 4 columns:

- `year_day`: Used for the Through: fields. Can be in one of the following formats:
  - `character`: Day specifications in either `mm/dd` or `mm-dd` where `mm` is the month in `1:12` or in character and `dd` is the day in month in `1:31`
  - `Date`: The year component will be ignored and only the month and day component will be used
  - `integer`: Julian day, e.g. `360`, `365` and etc
- `id` (Optional): Integer type. Used to group together different day types with same schedule values. Grouped day types will be compacted in a single For: field, e.g. For: Weekdays SummaryDesignDay. Grouped day types should have the same schedule values. Otherwise an error will be issued.
- `daytype`: Character type. Used for the For: fields. All possible values are listed below. Case-insensitive matching is used. Different day types can be grouped using the `id` column mentioned above or put together directly in a single string separate by comma (,), e.g. `"weekend,holiday"`
  - `"AllDay(s)"`
  - `"Weekday(s)"`, and also `"Monday"`, `"Tuesday"`, `"Wednesday"`, `"Thursday"` and `"Friday"`
  - `"Weekend(s)"`, and also `"Saturday"` and `"Sunday"`
  - `"SummaryDesignDay"` and `"WinterDesignDay"`
  - `"Holiday"`
  - `"CustomDay1"` and `"CustomDay2"`
  - `"AllOtherDay(s)"`
- `time`: Used for the Until: fields. Can be in one of the following formats:
  - `character`: Time specifications in `HH:MM` where `HH` is the hour in `0:24` and `MM` is the minute in `0:60`
  - `integer`: Time differences from `00:00:00` in **minutes**, e.g. `seq(60,60 * 24,by = 60)`
  - `hms`: `hms` objects constructed using `hms::hms()` or equivalents, e.g. `hms::hms(minutes = 30,hours = 1)`, and `hms::as_hms("1:30:00")`

  - difftime: difftime objects constructed using as.difftime() or equivalents, e.g. as.difftime(1:24,units = "hours")
  - ITime: ITime objects constructed using data.table::as.ITime(), e.g. as.ITime("01:30:00")
- value: Numeric type. Used for the actual schedule values

check_range If TRUE, schedule values will be checked based on $type_limits(). Default: TRUE.

compact If TRUE, same schedule values from different day types will be compacted together. Also, time periods that have the same schedule values will also be compacted. Note that only "Holiday", "CustomDay1" and "CustomDay2" will be compacted into "AllOtherDays". For example, if the daytype column contains only "Weekdays", "SummerDesignDay" and "AllOtherDays", "AllOtherDays" will be expanded to "Weekends", "WinterDesignDay" and "AllOtherDays". Default: FALSE.

Returns: The modified IdfScheduleCompact object.

Examples:

```
\dontrun{
sch$update(sch$extract())

val1 <- data.table::data.table(
    year_day = "12/31",
    daytype = "weekday,summerdesignday",
   time = c("6:00", "8:00", "12:00", "13:30", "14:00", "18:00", "19:00", "24:00"),
    value = c(0.2,    0.5,    0.95,    0.6,     0.8,     0.95,    0.2,     0.0)
)
val2 <- data.table::data.table(
    year_day = "12/31", daytype = "allotherday",
    time = "24:00", value = 0.0
)
val <- data.table::rbindlist(list(val1, val2))
sch$update(val)
}
```

Method validate()**:** Check possible object field value errors

Usage:

```
IdfScheduleCompact$validate(level = eplusr_option("validate_level"))
```

Arguments:

level One of "none", "draft", "final" or a list of 10 elements with same format as custom_validate() output.

Details: $validate() checks if there are errors in current IdfScheduleCompact object under specified validation level and returns an IdfValidity object. Schedule value ranges will be checked if current validate level contains range checking (if corresponding ScheduleTypeLimits Numeric Type is Continuous) or choice checking (if corresponding ScheduleTypeLimits Numeric Type is Discrete).
For detailed description on validate checking, see IdfObject$validate() documentation above.

Returns: An IdfValidity object.

Examples:

```
\dontrun{
sch$validate()

# check at predefined validate level
sch$validate("none")
sch$validate("draft")
sch$validate("final")
}
```

Method `is_valid()`**:** Check if there is any error in current object

Usage:

```
IdfScheduleCompact$is_valid(level = eplusr_option("validate_level"))
```

Arguments:

`level` One of `"none"`, `"draft"`, `"final"` or a list of 10 elements with same format as `custom_validate()` output.

Details: $is_valid() returns `TRUE` if there is no error in current `IdfScheduleCompact` object under specified validation level. Schedule value ranges will be checked if current validate level contains range checking (if corresponding `ScheduleTypeLimits` Numeric Type is `Continuous`) or choice checking (if corresponding `ScheduleTypeLimits` Numeric Type is `Discrete`). $is_valid() checks if there are errors in current `IdfObject` object under specified validation level and returns `TRUE` or `FALSE` accordingly. For detailed description on validate checking, see `IdfObject$validate()` documentation above.

Returns: A single logical value of `TRUE` or `FALSE`.

Examples:

```
\dontrun{
sch$is_valid()
}
```

Method `extract()`**:** Extract schedule values

Usage:

```
IdfScheduleCompact$extract(daytype = NULL, timestep = NULL)
```

Arguments:

`daytype` Should be one of:

- `NULL`: Do nothing Grouped day types will be concatenated with a comma, e.g. Weekdays,SummerDesignDay. This is the default behavior.
- `TRUE` or `"expand"`: All compacted day types will be expanded.
- `FALSE` or `"compact"`: Same schedule values from different day types will be compacted together.
- A character vector specifying the grouped day types, e.g. `c("weekday","summerdesignday")`. All other days except specified ones will be classified into day type `AllOtherDays`, if possible. If not possible, those day types will still be extracted separately.

`timestep` The time step of schedule values, e.g. "10 mins" and "1 hour". Valid units include `sec(s)`, `min(s)`, and `hour(s)`. If `NULL`, the original time specifications will be kept. If `"auto"`, the time periods with the same schedule values will be compacted. Default: `NULL`.

Returns: `NULL` if current schedule is empty. Otherwise, a `data.table::data.table()` of 5 columns:

- `year_day`: Character type. Used for the Through: fields. Day specifications in `mm/dd` format
- `id`: Integer type. The group index of day types
- `daytype`: Character type. Used for the For: fields. All possible values are:
  - `"AllDay"`
  - `"Weekday"`, and also `"Monday"`, `"Tuesday"`, `"Wednesday"`, `"Thursday"` and `"Friday"`
  - `"Weekend"`, and also `"Saturday"` and `"Sunday"`
  - `"SummaryDesignDay"` and `"WinterDesignDay"`
  - `"Holiday"`
  - `"CustomDay1"` and `"CustomDay2"`
  - `"AllOtherDay"`
- `time`: `hms` vector. Used for the Until: fields. It is handy for plotting since `hms` object is directly supported by the scale system of ggplot2 package
- `value`: Numeric type. Actual schedule values

Examples:

```
\dontrun{
sch$extract()
sch$extract("expand")
sch$extract(timestep = "30 mins")
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
IdfScheduleCompact$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Hongyuan Jia

See Also

Idf class

Examples

```
## ------------------------------------------------
## Method `IdfScheduleCompact$new`
## ------------------------------------------------

## Not run:
model <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"))
```

```
# create an empty 'Schedule:Compact'
schedule_compact(model, "sch", TRUE)

# get an existing 'Schedule:Compact'
sch <- schedule_compact(model, "sch")

## End(Not run)

## ------------------------------------------------
## Method `IdfScheduleCompact$set`
## ------------------------------------------------

## Not run:
sch$set(c("weekday", "summerdesignday") := list(
    ..6 = 0.2, "8:00" = 0.5,
    ..12 = 0.95, "13:30" = 0.6, ..14 = 0.8,
    ..18 = 0.95, ..19 = 0.2, ..24 = 0),
    allotherday = list(..24 = 0)
)

## End(Not run)

## ------------------------------------------------
## Method `IdfScheduleCompact$update`
## ------------------------------------------------

## Not run:
sch$update(sch$extract())

val1 <- data.table::data.table(
    year_day = "12/31",
    daytype = "weekday,summerdesignday",
    time = c("6:00", "8:00", "12:00", "13:30", "14:00", "18:00", "19:00", "24:00"),
    value = c(0.2,    0.5,    0.95,    0.6,     0.8,     0.95,    0.2,     0.0)
)
val2 <- data.table::data.table(
    year_day = "12/31", daytype = "allotherday",
    time = "24:00", value = 0.0
)
val <- data.table::rbindlist(list(val1, val2))
sch$update(val)

## End(Not run)

## ------------------------------------------------
## Method `IdfScheduleCompact$validate`
## ------------------------------------------------

## Not run:
sch$validate()

# check at predefined validate level
```

```
sch$validate("none")
sch$validate("draft")
sch$validate("final")

## End(Not run)

## ------------------------------------------------
## Method `IdfScheduleCompact$is_valid`
## ------------------------------------------------

## Not run:
sch$is_valid()

## End(Not run)


## ------------------------------------------------
## Method `IdfScheduleCompact$extract`
## ------------------------------------------------

## Not run:
sch$extract()
sch$extract("expand")
sch$extract(timestep = "30 mins")

## End(Not run)
```

`IdfViewer` *Visualize an EnergyPlus Model Geometry and Simulation Results*

Description

`IdfViewer` is a class designed to view geometry of an Idf and map simulation results to the geometries.

Usage

```
idf_viewer(geometry)
```

Arguments

`geometry`	An IdfGeometry object. `geometry` can also be a path to an IDF file or an Idf object. In this case, an `IdfGeometry` is created based on input Idf.

Value

An `IdfViewer` object.

Methods

Public methods:

- IdfViewer$new()
- IdfViewer$parent()
- IdfViewer$geometry()
- IdfViewer$device()
- IdfViewer$background()
- IdfViewer$viewpoint()
- IdfViewer$win_size()
- IdfViewer$mouse_mode()
- IdfViewer$axis()
- IdfViewer$ground()
- IdfViewer$wireframe()
- IdfViewer$x_ray()
- IdfViewer$render_by()
- IdfViewer$show()
- IdfViewer$focus()
- IdfViewer$close()
- IdfViewer$snapshot()
- IdfViewer$print()

Method `new()`**:** Create an `IdfViewer` object

Usage:

```
IdfViewer$new(geometry)
```

Arguments:

`geometry` An IdfGeometry object. `geometry` can also be a path to an IDF file or an Idf object. In this case, an `IdfGeometry` is created based on input Idf.

Returns: An `IdfViewer` object.

Examples:

```
\dontrun{
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# create from an Idf object
idf <- read_idf(path_idf, use_idd(8.8, "auto"))
viewer <- idf_viewer(idf)
viewer <- IdfViewer$new(idf)

# create from an IDF file
viewer <- idf_viewer(path_idf)
viewer <- IdfViewer$new(path_idf)
}
```

Method `parent()`**:** Get parent Idf object

Usage:

```
IdfViewer$parent()
```

Details: $parent() returns the parent Idf object of current `IdfGeometry` object.

Returns: An Idf object.

Examples:

```
\dontrun{
viewer$parent()
}
```

Method `geometry()`**:** Get parent IdfGeometry object

Usage:

```
IdfViewer$geometry()
```

Details: $geometry() returns the parent IdfGeometry object.

Returns: An IdfGeometry object.

Examples:

```
\dontrun{
viewer$geometry()
}
```

Method `device()`**:** Get Rgl device ID

Usage:

```
IdfViewer$device()
```

Details: If Rgl is used, the Rgl device ID is returned. If WebGL is used, the `elementID` is returned. If no viewer has been open, `NULL` is returned.

Returns: A number or `NULL`

Examples:

```
\dontrun{
viewer$device()
}
```

Method `background()`**:** Set the background color of the scene

Usage:

```
IdfViewer$background(color = "white")
```

Arguments:

`color` A single string giving the background color. Default: `white`.

Examples:

```
\dontrun{
viewer$background("blue")
}
```

Method `viewpoint()`**:** Set the viewpoint orientation of the scene

Usage:

```
IdfViewer$viewpoint(
  look_at = "iso",
  theta = NULL,
  phi = NULL,
  fov = NULL,
  zoom = NULL,
  scale = NULL
)
```

Arguments:

`look_at` A single string indicating a standard view. If specified, `theta` and `phi` will be ignored. Should be one of `c("top","bottom","left","right","front","back","iso")`. `look_at` will be ignored if any of `theta` and `phi` is specified. Default: `iso` (i.e. isometric).

`theta` Theta in polar coordinates. If `NULL`, no changes will be made to current scene. Default: `NULL`.

`phi` Phi in polar coordinates. If `NULL`, no changes will be made to current scene. Default: `NULL`.

`fov` Field-of-view angle in degrees. If `0`, a parallel or orthogonal projection is used. If `NULL`, no changes will be made to current scene. Default: `NULL`.

`zoom` Zoom factor. If `NULL`, no changes will be made to current scene. Default: `NULL`.

`scale` A numeric vector of length 3 giving the rescaling to apply to each axis. If `NULL`, no changes will be made to current scene. Default: `NULL`.

Examples:

```
\dontrun{
viewer$viewpoint()
}
```

Method `win_size()`**:** Set the window size

Usage:

```
IdfViewer$win_size(left = 0, top = 0, right = 600, bottom = 600)
```

Arguments:

`left`, `top`, `right`, `bottom` A single number indicating the pixels of the displayed window. Defaults: `0` (`left`), `0` (`top`), `600` (`right`) and `600` (`bottom`).

Examples:

```
\dontrun{
viewer$win_size(0, 0, 400, 500)
}
```

Method `mouse_mode()`**:** Set the handlers of mouse control

Usage:

```
IdfViewer$mouse_mode(
  left = "trackball",
  right = "pan",
  middle = "fov",
  wheel = "push"
)
```

Arguments:

`left`, `right`, `middle` Refer to the buttons on a three button mouse, or simulations of them on other mice. Defaults: `"trackball"` (`left`), `"pan"` (`right`) and `"fov"` (`middle`).

`wheel` Refer to the mouse wheel. Default: `"push"`.

Details: Possible values are:

Mode	Description
"none"	No action
"trackball"	The mouse acts as a virtual trackball. Clicking and dragging rotates the scene
"xAxis", "yAxis", "zAxis"	Like "trackball", but restricted to rotation about one axis
"polar"	The mouse affects rotations by controlling polar coordinates directly
"zoom"	The mouse zooms the display
"fov"	The mouse affects perspective by changing the field of view
"pull"	Rotating the mouse wheel towards the user " pulls the scene closer"
"push"	The same rotation "pushes the scene away"
"pan"	Pan the camera view vertically or horizontally

Examples:

```
\dontrun{
viewer$mouse_mode()
}
```

Method `axis()`**:** Toggle axis in the scene

Usage:

```
IdfViewer$axis(
  add = TRUE,
  expand = 2,
  width = 1.5,
  color = c("red", "green", "blue", "orange"),
  alpha = 1
)
```

Arguments:

`add` If `TRUE`, axis is added to the scene. If `FALSE`, axis is removed in the scene.

`expand` A single number giving the factor to expand based on the largest X, Y and Z coordinate values. Default: `2.0`.

`width` A number giving the line width of axis. `width * 2` is used for the true north axis. Default: `1.5`.

`color` A character of length 4 giving the color of X, Y, Z and true north axis. Default: `c("red","green","blue","orang`

`alpha` A number giving the alpha value of axis. Default: `1.0`.

Details: $axis() adds or removes X, Y and Z axis in the scene.

Returns: A single logical value as `add`.

Examples:

```
\dontrun{
viewer$axis()
}
```

Method `ground()`**:** Toggle ground in the scene

Usage:

```
IdfViewer$ground(add = TRUE, expand = 1.02, color = "#EDEDEB", alpha = 1)
```

Arguments:

`add` If `TRUE`, ground is added to the scene. If `FALSE`, ground is removed in the scene.

`expand` A single number giving the factor to expand based on the largest X, Y and Z coordinate values. Default: `1.02`.

`color` A string giving the color of ground. Default: #EDEDEB.

`alpha` A number giving the alpha value of ground. Default: `1.0`.

Details: $ground() adds or removes ground in the scene.

Returns: A single logical value as add.

Examples:

```
\dontrun{
viewer$ground()
}
```

Method `wireframe()`**:** Toggle wireframe

Usage:

```
IdfViewer$wireframe(add = TRUE, width = 1.5, color = "black", alpha = 1)
```

Arguments:

`add` If `TRUE`, wireframe is turned on. If `FALSE`, wireframe is turned off. Default: `TRUE`.

`width` A number giving the line width of axis. Default: `1.5`.

`color` A character of length 3 giving the color of X, Y and Z axis. Default: `c("red","green","blue")`.

`alpha` A number giving the alpha value of axis. Default: `1.0`.

Details: $wireframe() turns on/off wireframes.

Returns: A single logical value as add.

Examples:

```
\dontrun{
viewer$wireframe()
}
```

Method `x_ray()`**:** Toggle X-ray face style

Usage:

```
IdfViewer$x_ray(on = TRUE)
```

Arguments:

`on` If `TRUE`, X-ray is turned on. If `FALSE`, X-ray is turned off. Default: `TRUE`.

Details: $x_ray() turns on/off X-ray face style.

Returns: A single logical value as on.

Examples:

```
\dontrun{
viewer$x_ray()
}
```

Method `render_by()`**:** Set render style

Usage:

```
IdfViewer$render_by(type = "surface_type")
```

Arguments:

`type` A single string giving the render style. Should be one of:

- `"surface_type"`: Default. Render the model by surface type model. Walls, roofs, windows, doors, floors, and shading surfaces will have unique colors.
- `"boundary"`: Render the model by outside boundary condition. Only surfaces that have boundary conditions will be rendered with a color. All other surfaces will be white.
- `"construction"`: Render the model by surface constructions.
- `"zone"`: Render the model by zones assigned.
- `"normal"`: Render the model by surface normal. The outside face of a heat transfer face will be rendered as white and the inside face will be rendered as red.

Details: $render_by() sets the render style of geometries.

Returns: A same value as `style`.

Examples:

```
\dontrun{
viewer$render_by()
}
```

Method `show()`**:** Show Idf geometry

Usage:

```
IdfViewer$show(
  type = "all",
  zone = NULL,
  surface = NULL,
  width = 1.5,
  dayl_color = "red",
  dayl_size = 5
)
```

Arguments:

`type` A character vector of geometry components to show. If `"all"` (default), all geometry components will be shown. If `NULL`, no geometry faces will be shown. Otherwise, should be a subset of following:

- `"floor"`
- `"wall"`
- `"roof"`
- `"window"`
- `"door"`
- `"shading"`
- `"daylighting"`

`zone` A character vector of names or an integer vector of IDs of zones in current Idf to show. If `NULL`, no subsetting is performed.

`surface` A character vector of names or an integer vector of IDs of surfaces in current Idf to show. If `NULL`, no subsetting is performed.

`width` The line width for the geometry components. Default: `1.5`.

`dayl_color, dayl_size` The color and size of daylighting reference points. Defaults: `"red"` (`dayl_color`) and `5` (`dayl_size`).

Returns: The `IdfViewer` itself, invisibly.

Examples:

```
\dontrun{
viewer$show()
}
```

Method `focus()`**:** Bring the scene window to the top

Usage:

```
IdfViewer$focus()
```

Examples:

```
\dontrun{
viewer$top()
}
```

Method `close()`**:** Close the scene window

Usage:

```
IdfViewer$close()
```

Examples:

```
\dontrun{
viewer$close()
}
```

Method `snapshot()`**:** Capture and save current rgl view as an image

Usage:

```
IdfViewer$snapshot(filename)
```

Arguments:

`filename` A single string specifying the file name. Current supported formats are `png`, `pdf`, `svg`, `ps`, `eps`, `tex` and `pgf`.

Details: $snapshot() captures the current rgl view and saves it as an image file to disk.

Returns: A single string of the file path.

Examples:

```
\dontrun{
viewer$show()
viewer$snapshot(tempfile(fileext = ".png"))
}
```

Method `print()`**:** Print an `IdfViewer` object

Usage:

```
IdfViewer$print()
```

Returns: The `IdfViewer` itself, invisibly.

Examples:

```
\dontrun{
viewer$print()
}
```

Author(s)

Hongyuan Jia

See Also

IdfGeometry class

Examples

```
## ------------------------------------------------
## Method `IdfViewer$new`
## ------------------------------------------------

## Not run:
# example model shipped with eplusr from EnergyPlus v8.8
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# create from an Idf object
idf <- read_idf(path_idf, use_idd(8.8, "auto"))
viewer <- idf_viewer(idf)
viewer <- IdfViewer$new(idf)

# create from an IDF file
viewer <- idf_viewer(path_idf)
viewer <- IdfViewer$new(path_idf)

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$parent`
## ------------------------------------------------

## Not run:
viewer$parent()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$geometry`
## ------------------------------------------------
```

```
## Not run:
viewer$geometry()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$device`
## ------------------------------------------------

## Not run:
viewer$device()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$background`
## ------------------------------------------------

## Not run:
viewer$background("blue")

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$viewpoint`
## ------------------------------------------------

## Not run:
viewer$viewpoint()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$win_size`
## ------------------------------------------------

## Not run:
viewer$win_size(0, 0, 400, 500)

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$mouse_mode`
## ------------------------------------------------

## Not run:
viewer$mouse_mode()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$axis`
## ------------------------------------------------
```

```
## Not run:
viewer$axis()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$ground`
## ------------------------------------------------

## Not run:
viewer$ground()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$wireframe`
## ------------------------------------------------

## Not run:
viewer$wireframe()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$x_ray`
## ------------------------------------------------

## Not run:
viewer$x_ray()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$render_by`
## ------------------------------------------------

## Not run:
viewer$render_by()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$show`
## ------------------------------------------------

## Not run:
viewer$show()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$focus`
```

```
## ------------------------------------------------

## Not run:
viewer$top()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$close`
## ------------------------------------------------

## Not run:
viewer$close()

## End(Not run)

## ------------------------------------------------
## Method `IdfViewer$snapshot`
## ------------------------------------------------

## Not run:
viewer$show()
viewer$snapshot(tempfile(fileext = ".png"))

## End(Not run)


## ------------------------------------------------
## Method `IdfViewer$print`
## ------------------------------------------------

## Not run:
viewer$print()

## End(Not run)
```

`idf_object` *Create an* `IdfObject` *object.*

Description

`idf_object()` takes a parent `Idf` object, an object name or class name, and returns a corresponding IdfObject.

Usage

```
idf_object(parent, object = NULL, class = NULL)
```

Arguments

parent	An Idf object.
object	A valid object ID (an integer) or name (a string). If NULL and class is not NULL, an empty IdfObject is created with all fields fill with default values if possible. Default: NULL.
class	A valid class name (a string). If object is not NULL, class is used to further specify what class is the target object belongs to. If object is NULL, an empty IdfObject of class is created.

Details

If object is not given, an empty IdfObject of specified class is created, with all field values filled with defaults, if possible. Note that validation is performed when creating, which means that an error may occur if current validate level does not allow empty required fields.

The empty IdfObject is directly added into the parent Idf object. It is recommended to use $validate() method in IdfObject to see what kinds of further modifications are needed for those empty fields and use $set() method to set field values.

Value

An IdfObject object.

Examples

```
## Not run:
model <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"))

# get an IdfObject using object ID
idf_object(model, 14)

# get an IdfObject using object name (case-insensitive)
idf_object(model, "zone one")

# `class` argument is useful when there are objects with same name in
# different class
idf_object(model, "zone one", "Zone")

# create a new zone
eplusr_option(validate_level = "draft")
zone <- idf_object(model, class = "Zone")
zone
eplusr_option(validate_level = "final")
zone$validate()

## End(Not run)
```

`install_eplus` *Download and Install EnergyPlus*

Description

Download specified version of EnergyPlus for your platform from GitHub and install it.

Usage

```
install_eplus(ver = "latest", local = FALSE, dir = NULL, force = FALSE, ...)

download_eplus(ver = "latest", dir)
```

Arguments

`ver`	The EnergyPlus version number, e.g., `8.7`. The special value `"latest"`, which is the default, means the latest version.
`local`	Whether to install EnergyPlus only for current user. For Windows and Linux, if `FALSE`, administrative privileges are required to install EnergyPlus to the default system-level location. See details. `local` should be also set to `FALSE` if you do not have the write access to the directory specified via `dir`. Default: `FALSE`. For macOS, administrative privileges are always required no matter you want EnergyPlus to be install at /Applications or ~/Applications.
`dir`	• For `download_eplus()`, where to save EnergyPlus installer file. Default: `"."`. – For `install_eplus()`, the installer will always be saved into `tempdir()`. But you can use `dir` to specify the **parent** directory of EnergyPlus installation, i.e. the **parent** directory of `EnergyPlusVX-Y-0` on Windows and `EnergyPlus-X-Y-0` on Linux. macOS is not supported. If `NULL`, the default installation path will be used. See details for more information. Please note that `dir` does not work on macOS and EnergyPlus will always be installed into the default location. Default: `NULL`.
`force`	Whether to install EnergyPlus even if it has already been installed.
`...`	Other arguments to be passed to the installer. Current only one additional argument exists and is only for Linux: * dir_bin: A path where symbolic links will be created to the software executables. The default is /usr/local/bin if `local` is `FALSE` and ~/.local/bin if `local` is `TRUE`.

Details

`download_eplus()` downloads specified version of EnergyPlus from EnergyPlus GitHub Repository.

`install_eplus()` will try to install EnergyPlus into the default location, e.g. C:\EnergyPlusVX-Y-0 on Windows, /usr/local/EnergyPlus-X-Y-0 on Linux, and /Applications/EnergyPlus-X-Y-0 on macOS.

Note that installing to the default location requires administrative privileges and you have to run R with administrator (or with sudo if you are on Linux) to make it work if you are not in interactive mode.

If you can't run R with administrator, it is possible to install EnergyPlus to your home corresponding directory by setting `local` to `TRUE`.

The user level EnergyPlus installation path is:

- Windows:
  - `dir(Sys.getenv("LOCALAPPDATA"),"EnergyPlusVX-Y-0")` OR
  - C:\Users\<User>\AppData\Local\EnergyPlusVX-Y-0 if environment variable `"LOCALAPPDATA"` is not set
- macOS: /Users/<User>/Applications/EnergyPlus-X-Y-0
- Linux: `"~/.local/EnergyPlus-X-Y-0"`

On Windows and Linux, you can also specify your custom directory using the `dir` argument. Remember to change `local` to `FALSE` in order to ask for administrator privileges if you do not have the write access to that directory.

Please note that on macOS, when `local` is set to `FALSE`, no symbolic links will be created, since this process requires administrative privileges.

Value

An invisible integer `0` if succeed. Moreover, some attributes will also be returned:

- For `install_eplus()`:
  - `path`: the EnergyPlus installation path
  - `installer`: the path of downloaded EnergyPlus installer file
- For `download_eplus()`:
  - `file`: the path of downloaded EnergyPlus installer file

Author(s)

Hongyuan Jia

Examples

```
## Not run:
# download the latest version of EnergyPlus
download_eplus("latest", dir = tempdir())
# install the latest version of EnergyPlus system-wide which is the default
# and requires administrative privileges
install_eplus("latest")

# for a specific version of EnergyPlus
download_eplus(8.8, dir = tempdir())
install_eplus(8.8)

# force to reinstall
```

```
install_eplus(8.8, force = TRUE)

# install EnergyPlus in your home directory
install_eplus(8.8, local = TRUE, force = TRUE)

# custom EnergyPlus install home directory
install_eplus(8.8, dir = "~/MyPrograms", local = TRUE, force = TRUE)

## End(Not run)
```

`is_eplus_ver` *Check for Idd, Idf and Epw objects*

Description

These functions test if input is a valid object of Idd, Idf, Epw and other main classes.

Usage

```
is_eplus_ver(ver, strict = FALSE)

is_idd_ver(ver, strict = FALSE)

is_eplus_path(path)

is_idd(x)

is_idf(x)

is_iddobject(x)

is_idfobject(x)

is_epw(x)
```

Arguments

`ver`	A character or numeric vector with suitable numeric version strings.
`strict`	If `FALSE`, `ver` can be a special string "latest" which represents the latest version.
`path`	A path to test.
`x`	An object to test.

Details

`is_eplus_ver()` returns `TRUE` if input is a valid EnergyPlus version.

`is_idd_ver()` returns `TRUE` if input is a valid EnergyPlus IDD version.

`is_eplus_path()` returns `TRUE` if input path is a valid EnergyPlus path, i.e. a path where there is an `energyplus` executable and an `Energy+.idd` file.

`is_idd()` returns `TRUE` if input is an Idd object.

`is_idf()` returns `TRUE` if input is an Idf object.

`is_iddobject()` returns `TRUE` if input is an IddObject object.

`is_idfobject()` returns `TRUE` if input is an IdfObject object.

`is_epw()` returns `TRUE` if input is an Epw object.

Value

A logical vector.

Examples

```
is_eplus_ver(8.8)
is_eplus_ver(8.0)
is_eplus_ver("latest", strict = FALSE)

is_idd_ver("9.0.1")
is_idd_ver("8.0.1")

is_eplus_path("C:/EnergyPlusV9-0-0")
is_eplus_path("/usr/local/EnergyPlus-9-0-1")

is_idd(use_idd(8.8, download = "auto"))

idf <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"),
    idd = use_idd(8.8, download = "auto"))
is_idf(idf)

is_iddobject(idd_object(8.8, "Version"))

is_idfobject(idf_object(idf, 1))

## Not run:
is_epw(read_epw(download_weather("los angeles.*tmy3", type = "epw", ask = FALSE, max_match = 1)))

## End(Not run)
```

`level_checks` *Show components of validation strictness level*

Description

`level_checks()` takes input of a built in validation level or a custom validation level and returns a list with all validation components that level contains.

Usage

```
level_checks(level = eplusr_option("validate_level"))
```

Arguments

level	Should be one of "none", "draft", "final" or an output of custom_validate().

Value

A named list with 10 elements, e.g. required_object, unique_object, unique_name, extensible, required_field, auto_field, type, choice, range and reference. For the meaning of each validation component, seecustom_validate().

Examples

```
level_checks("draft")
level_checks("final")
level_checks(custom_validate(auto_field = TRUE))
level_checks(eplusr_option("validate_level"))
```

ParametricJob *Create and Run Parametric Analysis, and Collect Results*

Description

ParametricJob class provides a prototype of conducting parametric analysis of EnergyPlus simulations.

param_job() takes an IDF and EPW as input and returns a ParametricJob. For details on ParametricJob, please see ParametricJob class.

Usage

```
param_job(idf, epw)
```

Arguments

idf	A path to EnergyPlus IDF or IMF file or an Idf object.
epw	A path to EnergyPlus EPW file or an Epw object. epw can also be NULL which will force design-day-only simulation when $run() method is called. Note this needs at least one Sizing:DesignDay object exists in the Idf.

Details

Basically, it is a collection of multiple EplusJob objects. However, the model is first parsed and the Idf object is stored internally, instead of storing only the path of Idf like in EplusJob class. Also, an object in Output:SQLite with Option Type value of SimpleAndTabular will be automatically created if it does not exists, like Idf class does.

Value

A ParametricJob object.

Super class

eplusr::EplusGroupJob -> ParametricJob

Methods

Public methods:

- ParametricJob$new()
- ParametricJob$version()
- ParametricJob$seed()
- ParametricJob$weather()
- ParametricJob$models()
- ParametricJob$apply_measure()
- ParametricJob$save()
- ParametricJob$run()
- ParametricJob$print()

Method new(): Create a ParametricJob object

Usage:

```
ParametricJob$new(idf, epw)
```

Arguments:

idf Path to EnergyPlus IDF file or an Idf object.

epw Path to EnergyPlus EPW file or an Epw object. epw can also be NULL which will force design-day-only simulation when $run() method is called. Note this needs at least one Sizing:DesignDay object exists in the Idf.

Returns: A ParametricJob object.

Examples:

```
\dontrun{
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from an IDF and an EPW
    param <- param_job(idf_path, epw_path)
    param <- ParametricJob$new(idf_path, epw_path)

    # create from an Idf and an Epw object
    param_job(read_idf(idf_path), read_epw(epw_path))
}
```

```
}
```

Method `version()`**:** Get the version of seed IDF

Usage:

```
ParametricJob$version()
```

Details: $version() returns the version of input seed Idf object.

Returns: A `base::numeric_version()` object.

Examples:

```
\dontrun{
param$version()
}
```

Method `seed()`**:** Get the seed Idf object

Usage:

```
ParametricJob$seed()
```

Details: $seed() returns the parsed input seed Idf object.

Examples:

```
\dontrun{
param$seed()
}
```

Method `weather()`**:** Get the Epw object

Usage:

```
ParametricJob$weather()
```

Details: $weather() returns the input Epw object. If no Epw is provided when creating the `ParametricJob` object, `NULL` is returned.

Examples:

```
\dontrun{
param$weather()
}
```

Method `models()`**:** Get created parametric Idf objects

Usage:

```
ParametricJob$models()
```

Details: $models() returns a list of parametric models generated using input Idf object and `$apply_measure()` method. Model names are assigned in the same way as the `.names` argument in `$apply_measure()`. If no measure has been applied, `NULL` is returned. Note that it is not recommended to conduct any extra modification on those models directly, after they were created using `$apply_measure()`, as this may lead to an un-reproducible process. A warning message will be issued if any of those models has been modified when running simulations.

Examples:

```
\dontrun{
param$models()
}
```

Method `apply_measure()`**:** Create parametric models

Usage:

```
ParametricJob$apply_measure(measure, ..., .names = NULL)
```

Arguments:

`measure` A function that takes an `Idf` and other arguments as input and returns an Idf object as output.

`...` Arguments **except first** `Idf` **argument** that are passed to that `measure`.

`.names` A character vector of the names of parametric `Idfs`. If `NULL`, the new `Idfs` will be named in format `measure_name + number`.

Details: $apply_measure() allows to apply a measure to an Idf and creates parametric models for analysis. Basically, a measure is just a function that takes an Idf object and other arguements as input, and returns a modified Idf object as output. Use `...` to supply different arguments, **except for the first** `Idf` **argument**, to that measure. Under the hook, `base::mapply()` is used to create multiple Idfs according to the input values.

Returns: The modified `ParametricJob` object itself, invisibly.

Examples:

```
\dontrun{
# create a measure to change the orientation of the building
rotate_building <- function (idf, degree = 0L) {
    if (!idf$is_valid_class("Building")) {
       stop("Input model does not have a Building object")
    }

    if (degree > 360 || degree < -360 ) {
        stop("Input degree should in range [-360, 360]")
    }

    cur <- idf$Building$North_Axis

    new <- cur + degree

    if (new > 360) {
        new <- new %% 360
        warning("Calculated new north axis is greater than 360. ",
            "Final north axis will be ", new
        )
    } else if (new < -360) {
        new <- new %% -360
        warning("Calculated new north axis is smaller than -360. ",
            "Final north axis will be ", new
```

```
        )
    }

    idf$Building$North_Axis <- new

    idf
}

# apply measure
# this will create 12 models
param$apply_measure(rotate_building, degree = seq(30, 360, 30))

# apply measure with new names specified
param$apply_measure(rotate_building, degree = seq(30, 360, 30),
    .names = paste0("rotate_", seq(30, 360, 30))
)
}
```

Method `save()`**:** Save parametric models

Usage:

```
ParametricJob$save(dir = NULL, separate = TRUE, copy_external = FALSE)
```

Arguments:

`dir` The parent output directory for models to be saved. If `NULL`, the directory of the seed model will be used. Default: `NULL`.

`separate` If `TRUE`, all models are saved in a separate folder with each model's name under specified directory. If `FALSE`, all models are saved in the specified directory. Default: `TRUE`.

`copy_external` Only applicable when `separate` is `TRUE`. If `TRUE`, the external files that every `Idf` object depends on will also be copied into the saving directory. The values of file paths in the Idf will be changed automatically.

Details: $save() saves all parametric models in specified folder. An error will be issued if no measure has been applied.

Returns: A `data.table::data.table()` with two columns:

- model: The path of saved parametric model files.
- weather: The path of saved weather files.

Examples:

```
\dontrun{
# save all parametric models with each model in a separate folder
param$save(tempdir())

# save all parametric models with all models in the same folder
param$save(tempdir(), separate = FALSE)
}
```

Method `run()`**:** Run parametric simulations

Usage:

```
ParametricJob$run(
  dir = NULL,
  wait = TRUE,
  force = FALSE,
  copy_external = FALSE,
  echo = wait,
  separate = TRUE
)
```

Arguments:

`dir` The parent output directory for specified simulations. Outputs of each simulation are placed in a separate folder under the parent directory.

`wait` If `TRUE`, R will hang on and wait all EnergyPlus simulations finish. If `FALSE`, all EnergyPlus simulations are run in the background. Default: `TRUE`.

`force` Only applicable when the last simulation runs with `wait` equals to `FALSE` and is still running. If `TRUE`, current running job is forced to stop and a new one will start. Default: `FALSE`.

`copy_external` If `TRUE`, the external files that current `Idf` object depends on will also be copied into the simulation output directory. The values of file paths in the Idf will be changed automatically. Currently, only `Schedule:File` class is supported. This ensures that the output directory will have all files needed for the model to run. Default is `FALSE`.

`echo` Only applicable when `wait` is `TRUE`. Whether to simulation status. Default: same as `wait`.

`separate` If `TRUE`, all models are saved in a separate folder with each model's name under `dir` when simulation. If `FALSE`, all models are saved in `dir` when simulation. Default: `TRUE`.

Details: $run() runs all parametric simulations in parallel. The number of parallel EnergyPlus process can be controlled by `eplusr_option("num_parallel")`. If `wait` is FALSE, then the job will be run in the background. You can get updated job status by just printing the `ParametricJob` object.

Returns: The `ParametricJob` object itself, invisibly.

Examples:

```
\dontrun{
# run parametric simulations
param$run(wait = TRUE, echo = FALSE)

# run in background
param$run(wait = FALSE)
# get detailed job status by printing
print(param)
}
```

Method `print()`**:** Print `ParametricJob` object

Usage:

```
ParametricJob$print()
```

Details: $print() shows the core information of this `ParametricJob`, including the path of IDFs and EPWs and also the simulation job status.
$print() is quite useful to get the simulation status, especially when `wait` is `FALSE` in $run(). The job status will be updated and printed whenever $print() is called.

Returns: The `ParametricJob` object itself, invisibly.

Examples:

```
\dontrun{
param$print()

Sys.sleep(10)
param$print()
}
```

Author(s)

Hongyuan Jia

See Also

`eplus_job()` for creating an EnergyPlus single simulation job.

Examples

```
## ------------------------------------------------
## Method `ParametricJob$new`
## ------------------------------------------------

## Not run:
if (is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from an IDF and an EPW
    param <- param_job(idf_path, epw_path)
    param <- ParametricJob$new(idf_path, epw_path)

    # create from an Idf and an Epw object
    param_job(read_idf(idf_path), read_epw(epw_path))
}

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$version`
```

```
## ------------------------------------------------

## Not run:
param$version()

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$seed`
## ------------------------------------------------

## Not run:
param$seed()

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$weather`
## ------------------------------------------------

## Not run:
param$weather()

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$models`
## ------------------------------------------------

## Not run:
param$models()

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$apply_measure`
## ------------------------------------------------

## Not run:
# create a measure to change the orientation of the building
rotate_building <- function (idf, degree = 0L) {
    if (!idf$is_valid_class("Building")) {
       stop("Input model does not have a Building object")
    }

    if (degree > 360 || degree < -360 ) {
        stop("Input degree should in range [-360, 360]")
    }
```

```
    cur <- idf$Building$North_Axis

    new <- cur + degree

    if (new > 360) {
        new <- new %% 360
        warning("Calculated new north axis is greater than 360. ",
            "Final north axis will be ", new
        )
    } else if (new < -360) {
        new <- new %% -360
        warning("Calculated new north axis is smaller than -360. ",
            "Final north axis will be ", new
        )
    }

    idf$Building$North_Axis <- new

    idf
}

# apply measure
# this will create 12 models
param$apply_measure(rotate_building, degree = seq(30, 360, 30))

# apply measure with new names specified
param$apply_measure(rotate_building, degree = seq(30, 360, 30),
    .names = paste0("rotate_", seq(30, 360, 30))
)

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$save`
## ------------------------------------------------

## Not run:
# save all parametric models with each model in a separate folder
param$save(tempdir())

# save all parametric models with all models in the same folder
param$save(tempdir(), separate = FALSE)

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$run`
## ------------------------------------------------

## Not run:
# run parametric simulations
```

```
param$run(wait = TRUE, echo = FALSE)

# run in background
param$run(wait = FALSE)
# get detailed job status by printing
print(param)

## End(Not run)


## ------------------------------------------------
## Method `ParametricJob$print`
## ------------------------------------------------

## Not run:
param$print()

Sys.sleep(10)
param$print()

## End(Not run)
```

`parse_dots_value` *Parse object field values given in list format*

Description

Parse object field values given in list format

Usage

```
parse_dots_value(
  ...,
  .scalar = TRUE,
  .pair = FALSE,
  .ref_assign = TRUE,
  .unique = FALSE,
  .empty = FALSE,
  .env = parent.frame()
)
```

Arguments

`...`	Lists of object definitions. Each list should be named with a valid class/object id/name. ID should be denoted in style `..ID`. There is a special element `.comment` in each list, which will be used as new comments of the object. If `.ref_assign` is `TRUE`, := can be used to group ids/names:

- When `.type` equals `"class"`, LHS multiple class indices/names should be wrapped by `.()`, `c()`.
- When `.type` equals `"object"`, LHS multiple object ids/names should be wrapped by `.()` or `c()`. LHS **SINGLE** class name should be wrapped by `..()`.

`.scalar`	If `TRUE`, make sure the value of each field in the object is a scalar value. If `FALSE`, `value_chr` and `value_num` column will be list type. Default: `TRUE`.
`.pair`	Only works when `.scalar` is `FALSE`. If `.pair` is `TRUE`, vector field values will be paired to each id/name on the LHS. In this case, `value_chr` and `value_num` will be character type and double type, respectively. When there is only one id/name on the LHS, it will be replicated to match the length of the value vector. Default: `FALSE`.
`.ref_assign`	If `TRUE`, allow using := to gather multiple classes/objects on the LHS when defining the objects. Default: `TRUE`.
`.unique`	If `TRUE`, make sure there are no duplicated classes/objects in the input. Default: `FALSE`.
`.empty`	If `TRUE`, allow using an empty list, i.e. `list()` to define an object with all default values. Default: `TRUE`.
`.env`	An environment specifying the environment to evaluate the `....` Default: `parent.frame()`.

Value

A named list of 2 element `object` and `value` which is a `data.table::data.table()` with object data and value data respectively.

`print.ErrFile` *Print EnergyPlus Error File*

Description

`ErrFile` is mainly used to extract and print data in an EnergyPlus Error File (`.err`).

Usage

```
## S3 method for class 'ErrFile'
print(x, brief = FALSE, info = TRUE, ...)
```

Arguments

`x`	An `ErrFile` created using `read_err()`.
`brief`	If `TRUE`, only summary data is printed. Default: `FALSE`.
`info`	If `FALSE`, informative messages are excluded. Only warnings and errors are printed. Default: `TRUE`.
`...`	Further arguments passed to or from other methods.

Value

An ErrFile object, invisibly.

---

rdd_to_load *Format RddFile Object to Standard Input for* Idf$load() *Method*

Description

rdd_to_load() and mdd_to_load() takes a RddFile and MddFile object respectively and format it into a data.table in acceptable format for $load() method in Idf class.

Usage

```
rdd_to_load(rdd, key_value, reporting_frequency)

mdd_to_load(
  mdd,
  reporting_frequency,
 class = c("Output:Meter", "Output:Meter:MeterFileOnly", "Output:Meter:Cumulative",
    "Output:Meter:Cumulative:MeterFileOnly")
)
```

Arguments

rdd, mdd	A RddFile object created using read_rdd() and a MddFile object created using read_mdd(), respectively.
key_value	Key value name for **all** variables. If not specified and the key_value column in the input RddFile object will be used. If key_value column does not exist, "*" are used for all variables.
reporting_frequency	Variable value reporting frequency for **all** variables. If not specified and the reporting_freqency column in the input RddFile object will be used. If reporting_freqency column does not exist, "Timestep" are used for all variables. All possible values: "Detailed", "Timestep", "Hourly", "Daily", "Monthly", "RunPeriod", "Environment", and "Annual".
class	Class name for meter output. All possible values: "Output:Meter", "Output:Meter:MeterFileOnly", "Output:Meter:Cumulative", and "Output:Meter:Cumulative:MeterFileOnly". Default: "Output:Meter".

Value

A data.table with 5 columns with an additional attribute named eplus_version extracted from the original RddFile and MddFile:

- id: Integer type. Used to distinguish each object definition.
- class: Character type. Class names, e.g. Output:Variable and Output:Meter.

- index: Integer type. Field indices.
- field: Character type. Field names.
- value: Character type. The value of each field to be added.

Examples

```
## Not run:
# read an example distributed with eplusr
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")
idf <- read_idf(path_idf)

# run Design-Day-Only simulation silently
job <- idf$run(NULL, tempdir(), echo = FALSE)

# read RDD and MDD
rdd <- job$read_rdd()
mdd <- job$read_mdd()

# perform subsetting on the variables
# e.g.:
rdd_sub <- rdd[grepl("Site", variable)]
mdd_sub <- mdd[grepl("Electricity", variable)]

# use newly added helper `rdd_to_load()` and `mdd_to_load()` and `$load()` to
# add `Output:Variable` and `Output:Meter*`
idf$load(rdd_to_load(rdd_sub))
idf$load(mdd_to_load(mdd_sub))

# default `Key Value` is `"*"` and `Reporting Frequency` is `Timestep`
# can overwrite using `key_value` and `reporting_freqency` arg
rdd_to_load(rdd_sub, key_value = "Environment", reporting_frequency = "hourly")

# if input has column `key_value`, default is to use it, unless `key_value` is
# explicitly specified
rdd_to_load(rdd_sub[, key_value := "Environment"])
rdd_to_load(rdd_sub[, key_value := "Environment"], key_value = "*")

# `reporting_frequency` arg works in the same way as `key_value` arg, i.e.:
# if input has column `reporting_frequency`, use it, unless
# `reporting_frequency` is explicitly specified
rdd_to_load(rdd_sub[,  reporting_frequency := "monthly"])
rdd_to_load(rdd_sub[, reporting_frequency := "monthly"], reporting_frequency = "detailed")

# if input has column `key_value`, default is to use it, unless `key_value` is
# explicitly specified
rdd_to_load(rdd_sub[, key_value := "Environment"])
rdd_to_load(rdd_sub[, key_value := "Environment"], key_value = "*")

# meter class can be further specified using `class` arg
mdd_to_load(mdd_sub, class = "Output:Meter:MeterFileOnly")

## End(Not run)
```

read_epw *Read and Parse EnergyPlus Weather File (EPW)*

Description

read_epw() parses an EPW file and returns an Epw object. The parsing process is extremely inspired by [EnergyPlus/WeatherManager.cc] with some simplifications. For more details on Epw, please see Epw class.

Usage

```
read_epw(path)
```

Arguments

path	A path of an EnergyPlus EPW file.

Value

An Epw object.

Author(s)

Hongyuan Jia

See Also

Epw class

Examples

```
## Not run:
# read an EPW file from EnergyPlus v8.8 installation folder
if (is_avail_eplus(8.8)) {
    path_epw <- file.path(
        eplus_config(8.8)$dir,
        "WeatherData",
        "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
   )
   read_epw(path_epw)
}

# read an EPW file from EnergyPlus website
path_base <- "https://energyplus.net/weather-download"
path_region <- "north_and_central_america_wmo_region_4/USA/CA"
path_file <- "USA_CA_San.Francisco.Intl.AP.724940_TMY3/USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
path_epw <- file.path(path_base, path_region, path_file)
read_epw(path_epw)
```

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

`read_err` *Read an EnergyPlus Simulation Error File*

Description

`read_err()` takes a file path of EnergyPlus simulation error file, usually with an extension `.err`, parses it and returns an `ErrFile` object.

Usage

```
read_err(path)
```

Arguments

`path`	a file path of EnergyPlus simulation error file, usually with an extension `.err`.

Details

Basically, an `ErrFile` object is a data.table with 6 columns and 6 additional attributes:

6 Columns:

- `index`: Integer. Index of messages.
- `envir_index`: Integer. Index of simulation environments.
- `envir`: Character. Names of simulation environments.
- `level_index`: Integer. Index for each severe level.
- `level`: Character. Name of severe levels. Possible values: `Info`, `Warning`, `Severe`, and etc.
- `message`: Character. Error messages.

6 Attributes:

- `path`: A single string. The path of input file.
- `eplus_version`: A numeric_version object. The version of EnergyPlus used during the simulation.
- `eplus_build`: A single string. The build tag of EnergyPlus used during the simulation.
- `datetime`: A DateTime (POSIXct). The time when the simulation started.
- `idd_version`: A numeric_version. The version of IDD used during the simulation.
- `successful`: `TRUE` when the simulation ended successfully, and `FALSE` otherwise.
- `terminated`: `TRUE` when the simulation was terminated, and `FALSE` otherwise.

Value

An `ErrFile` object.

Examples

```
## Not run:
# run simulation and get the err file
idf_name <- "1ZoneUncontrolled.idf"
epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)
job <- eplus_job(idf_path, epw_path)
job$run(dir = tempdir())

# read the err file
read_err(job$locate_output(".err"))

## End(Not run)
```

`read_idf` *Read an EnergyPlus Input Data File (IDF)*

Description

`read_idf` takes an EnergyPlus Input Data File (IDF) as input and returns an `Idf` object. For more details on `Idf` object, please see Idf class.

Usage

```
read_idf(path, idd = NULL)
```

Arguments

`path`	Either a path, a connection, or literal data (either a single string or a raw vector) to an EnergyPlus Input Data File (IDF). If a file path, that file usually has a extension `.idf`.
`idd`	Any acceptable input of `use_idd()`. If `NULL`, which is the default, the version of IDF will be passed to `use_idd()`. If the input is an `.ddy` file which does not have a version field, the latest version of Idf cached will be used.

Details

Currently, Imf file is not fully supported. All EpMacro lines will be treated as normal comments of the nearest downwards object. If input is an Imf file, a warning will be given during parsing. It is recommended to convert the Imf file to an Idf file and use ParametricJob class to conduct parametric analysis.

Value

An Idf object.

Author(s)

Hongyuan Jia

See Also

Idf class for modifying EnergyPlus model. use_idd() and download_idd() for downloading and parsing EnergyPlus IDD file. use_eplus() for configuring which version of EnergyPlus to use.

Examples

```
## Not run:
# example model shipped with eplusr from EnergyPlus v8.8
idf_path <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr") # v8.8

# if neither EnergyPlus v8.8 nor Idd v8.8 was found, error will occur
# if EnergyPlus v8.8 is found but Idd v8.8 was not, `Energy+.idd` in EnergyPlus
# installation folder will be used for pasing
# if Idd v8.8 is found, it will be used automatically
is_avail_eplus(8.8)
is_avail_idd(8.8)

read_idf(idf_path)

# argument `idd` can be specified explicitly using `use_idd()`
read_idf(idf_path, idd = use_idd(8.8))

# you can set `download` arugment to "auto" in `use_idd()` if you want to
# automatically download corresponding IDD file when necessary
read_idf(idf_path, use_idd(8.8, download = "auto"))

# Besides use a path to an IDF file, you can also provide IDF in literal
# string format
idf_string <-
    "
    Version, 8.8;
    Building,
        Building;                !- Name
    "

read_idf(idf_string, use_idd(8.8, download = "auto"))

## End(Not run)
```

---

read_rdd *Read an EnergyPlus Report Data Dictionary File*

Description

read_rdd() takes a file path of EnergyPlus Report Data Dictionary (RDD) file, parses it and returns a RddFile object. read_mdd() takes a file path of EnergyPlus Meter Data Dictionary (MDD) file, parses it and returns a MddFile object.

Usage

```
read_rdd(path)

read_mdd(path)
```

Arguments

path	For `read_rdd()`, a file path of EnergyPlus EnergyPlus Report Data Dictionary file with an extension `.rdd`. For `read_mdd()`, a file path of EnergyPlus EnergyPlus Meter Data Dictionary file with an extension `.mdd`

Details

Basically, a `RddFile` and `MddFile` object is a data.table with 5 columns and 3 additional attributes:

5 Columns:

*`index`: Integer. Index of each variable.

- `reported_time_step`: Character. Reported time step for the variables. Possible value: `Zone` and `HVAC`.
- `report_type`: Character. Report types. Possible value: `Average`, `Sum` and `Meter`. Note that `Meter` is only for MDD file. All variables will have `report_type` being `Meter`.
- `variable`: Character. Report variable names.
- `units`: Character. Units of reported values. `NA` if report values do not have units.

3 Attributes:

- `eplus_version`: A numeric_version object. The version of EnergyPlus used during the simulation.
- `eplus_build`: A single string. The build tag of EnergyPlus used during the simulation.
- `datetime`: A DateTime (POSIXct). The time when the simulation started.

Value

For `read_rdd()`, an `RddFile` object. For `read_mdd()`, a `MddFile` object.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
# run simulation and get the err file
idf_name <- "1ZoneUncontrolled.idf"
epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
idf_path <- file.path(eplus_config(8.8)$dir, "ExampleFiles", idf_name)
epw_path <- file.path(eplus_config(8.8)$dir, "WeatherData", epw_name)
job <- eplus_job(idf_path, epw_path)
job$run(dir = tempdir())
```

```
# read the err file
read_rdd(job$locate_output(".rdd"))
read_mdd(job$locate_output(".mdd"))

## End(Not run)
```

`reload` *Reload Idf data*

Description

Reload Idf data

Usage

```
reload(x, ...)
```

Arguments

`x`	An object of class Idd, IddObject, Idf, IdfObject, Epw, EplusJob, EplusGroupJob or ParametricJob object. Any object of other class will be directly returned without any modifications.
`...`	further arguments passed to or from other methods. Currently not used.

Details

eplusr relies heavily on the data.table package. The core data of all main classes in eplusr are saved as `data.table::data.table()`s. This introduces a problem when loading saved Idf objects or other class objects via an *.RDS and *.RData file on disk: the stored `data.table::data.table()`s lose their column over-allocation. `reload()` is a helper function that calls `data.table::setDT()` on all internal `data.table::data.table()`s to make sure they are initialized properly.

It is recommended to call `reload()` on each Idd, Idf and other class object in eplusr loaded with `readRDS()` or `load()`, to make sure all eplusr's functionaries works properly.

Value

The input object with its internal `data.table::data.table()`s properly initialized.

run_idf *Run simulations of EnergyPlus models.*

Description

Run simulations of EnergyPlus models.

Usage

```
run_idf(
  model,
  weather,
  output_dir,
  design_day = FALSE,
  annual = FALSE,
  expand_obj = TRUE,
  wait = TRUE,
  echo = TRUE,
  eplus = NULL
)

run_multi(
  model,
  weather,
  output_dir,
  design_day = FALSE,
  annual = FALSE,
  expand_obj = TRUE,
  wait = TRUE,
  echo = TRUE,
  eplus = NULL
)
```

Arguments

model	A path (for run_idf()) or a vector of paths (for run_multi()) of EnergyPlus IDF or IMF files.
weather	A path (for run_idf()) or a vector of paths (for run_multi()) of EnergyPlus EPW weather files. For run_multi(), weather can also be a single EPW file path. In this case, that weather will be used for all simulations; otherwise, model and weather should have the same length.
output_dir	Output directory path (for rum_idf()) or paths (for run_mult()). If NULL, the directory of input model is used. For run_multi(), output_dir, if not NULL, should have the same length as model. Any duplicated combination of model and output_dir is prohibited.

design_day	Force design-day-only simulation. For rum_multi(), design_day can also be a logical vector which has the same length as model. Default: FALSE.
annual	Force design-day-only simulation. For rum_multi(), annual can also be a logical vector which has the same length as model. Note that design_day and annual cannot be all TRUE at the same time. Default: FALSE.
expand_obj	Whether to run ExpandObject preprocessor before simulation. Default: TRUE.
wait	If TRUE, R will hang on and wait all EnergyPlus simulations finish. If FALSE, all EnergyPlus simulations are run in the background, and a process object is returned.
echo	Only applicable when wait is TRUE. Whether to show standard output and error from EnergyPlus command line interface for run_idf() and simulation status for run_multi(). Default: TRUE.
eplus	An acceptable input (for run_idf()) or inputs (for run_multi()) of use_eplus() and eplus_config(). If NULL, which is the default, the version of EnergyPlus to use is determined by the version of input model. For run_multi(), eplus, if not NULL, should have the same length as model.

Details

run_idf() is a wrapper of EnergyPlus command line interface which enables to run EnergyPlus model with different options.

run_multi() provides the functionality of running multiple models in parallel.

run_idf() and run_multi() currently only support EnergyPlus v8.3 and above. This is because eplusr uses EnergyPlus command line interface which is a new feature as of EnergyPlus v8.3.

It is suggested to run simulations using EplusJob class and EplusGroupJob class, which provide much more detailed controls on the simulation and also methods to extract simulation outputs.

Value

- For run_idf(), a named list of 11 elements:

No.	Column	Type	Description
1	idf	character(1)	Full path of input IDF file
2	epw	character(1) or NULL	Full path of input EPW file
3	version	character(1)	Version of called EnergyPlus
4	exit_status	integer(1) or NULL	Exit status of EnergyPlus. NULL if terminated or wait is FALSE
5	start_time	POSIXct(1)	Start of time of simulation
6	end_time	POSIXct(1) or NULL	End of time of simulation. NULL if wait is FALSE
7	output_dir	character(1)	Full path of simulation output directory
8	energyplus	character(1)	Full path of called EnergyPlus executable
9	stdout	character(1) or NULL	Standard output of EnergyPlus during simulation
10	stderr	character(1) or NULL	Standard error of EnergyPlus during simulation
11	process	process	A process object which called EnergyPlus and ran the simulation

- For rum_multi(), if wait is TRUE, a data.table of 12 columns:

No.	Column	Type	Description
1	index	integer	Index of simulation
2	status	character	Simulation status
3	idf	character	Full path of input IDF file
4	epw	character	Full path of input EPW file. NA for design-day-only simulation
5	version	character	Version of EnergyPlus
6	exit_status	integer	Exit status of EnergyPlus. NA if terminated
7	start_time	POSIXct	Start of time of simulation
8	end_time	POSIXct	End of time of simulation.
9	output_dir	character	Full path of simulation output directory
10	energyplus	character	Full path of called EnergyPlus executable
11	stdout	list	Standard output of EnergyPlus during simulation
12	stderr	list	Standard error of EnergyPlus during simulation

For column status, there are 4 possible values:

- "completed": the simulation job is completed successfully
- "failed": the simulation job ended with error
- "terminated": the simulation job started but was terminated
- "cancelled": the simulation job was cancelled, i.e. did not start at all

- For run_multi(), if wait is FALSE, a r_process object of background R process which handles all simulation jobs is returned. You can check if the jobs are completed using $is_alive() and get the final data.table using $get_result() method.

Author(s)

Hongyuan Jia

References

Running EnergyPlus from Command Line (EnergyPlus GitHub Repository)

See Also

EplusJob class and ParametricJob class which provide a more friendly interface to run EnergyPlus simulations and collect outputs.

Examples

```
## Not run:
idf_path <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")

if (is_avail_eplus(8.8)) {
    # run a single model
    epw_path <- file.path(
        eplus_config(8.8)$dir,
        "WeatherData",
        "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
    )
```

```
    run_idf(idf_path, epw_path, output_dir = tempdir())

    # run multiple model in parallel
    idf_paths <- file.path(eplus_config(8.8)$dir, "ExampleFiles",
        c("1ZoneUncontrolled.idf", "1ZoneUncontrolledFourAlgorithms.idf")
    )
    epw_paths <- rep(epw_path, times = 2L)
    output_dirs <- file.path(tempdir(), tools::file_path_sans_ext(basename(idf_paths)))
    run_multi(idf_paths, epw_paths, output_dir = output_dirs)
}

## End(Not run)
```

`transition` *Perform version transition of EnergyPlus model*

Description

`transition()` takes an Idf object or a path of IDF file and a target version, performs version transitions and returns an Idf object of specified version.

Usage

```
transition(idf, ver, keep_all = FALSE, save = FALSE, dir = NULL)
```

Arguments

`idf`	An Idf object or a path of IDF file.
`ver`	A valid EnergyPlus IDD version, e.g. `9`, `8.8`, or `"8.8.0"`.
`keep_all`	If `TRUE`, a list will be return which contains all Idf objects of intermediate versions. The list will be named using first two number of that version, e.g. `8.1`, `8.2`. If `FALSE`, only the Idf object of the version specified by `ver` will be returned. Default: `FALSE`.
`save`	If `TRUE`, the models will be saved into specified directory. Default: `FALSE`.
`dir`	Only applicable when `save` is `TRUE`. The directory to save the new IDF files. If the directory does not exist, it will be created before save. If `NULL`, the directory of input Idf object or IDF file will be used. Default: `NULL`.

Value

An Idf object if `keep_all` is `FALSE` or a list of Idf objects if `keep_all` is `TRUE`.

Author(s)

Hongyuan Jia

See Also

See `version_updater()` which directly call EnergyPlus preprocessor `IDFVersionUpdater` to perform the version transitions.

Examples

```
## Not run:
if (any(avail_eplus()) > 7.2) {
    # create an empty IDF
    idf <- empty_idf(7.2)

    # convert it from v7.2 to the latest EnergyPlus installed
    transition(idf, max(avail_eplus()))

    # convert it from v7.2 to the latest EnergyPlus installed and keep all
    # intermediate versions
    transition(idf, max(avail_eplus()), keep_all = TRUE)

    # convert it from v7.2 to the latest EnergyPlus installed and keep all
    # intermediate versions and save all them
    idf$save(tempfile(fileext = ".idf"))
    transition(idf, max(avail_eplus()), keep_all = TRUE,
        save = TRUE, dir = tempdir()
    )
}

## End(Not run)
```

---

use_eplus *Configure which version of EnergyPlus to use*

Description

Configure which version of EnergyPlus to use

Usage

```
use_eplus(eplus)

eplus_config(ver)

avail_eplus()

is_avail_eplus(ver)
```

Arguments

`eplus`	An acceptable EnergyPlus version or an EnergyPlus installation path.
`ver`	An acceptable EnergyPlus version.

Details

use_eplus() adds an EnergyPlus version into the EnergyPlus version cache in eplusr. That cache will be used to get corresponding Idd object when parsing IDF files and call corresponding EnergyPlus to run models.

eplus_config() returns the a list of configure data of specified version of EnergyPlus. If no data found, an empty list will be returned.

avail_eplus() returns all versions of available EnergyPlus.

is_avail_eplus() checks if the specified version of EnergyPlus is available or not.

Value

- For use_eplus() and eplus_config(), an (invisible for use_eplus()) list of three contains EnergyPlus version, directory and EnergyPlus executable. version of EnergyPlus;
- For avail_eplus(), a numeric_version vector or NULL if no available EnergyPlus is found;
- For is_avis_avail_eplus(), a scalar logical vector.

See Also

download_eplus() and install_eplus() for downloading and installing EnergyPlus

Examples

```
## Not run:
# add specific version of EnergyPlus
use_eplus(8.9)
use_eplus("8.8.0")

# get configure data of specific EnergyPlus version if avaiable
eplus_config(8.6)

## End(Not run)

# get all versions of avaiable EnergyPlus
avail_eplus()

# check if specific version of EnergyPlus is available
is_avail_eplus(8.5)
is_avail_eplus(8.8)
```

---

use_idd *Use a specific EnergyPlus Input Data Dictionary (IDD) file*

Description

Use a specific EnergyPlus Input Data Dictionary (IDD) file

Usage

```
use_idd(idd, download = FALSE)

download_idd(ver = "latest", dir = ".")

avail_idd()

is_avail_idd(ver)
```

Arguments

idd	Either a path, a connection, or literal data (either a single string or a raw vector) to an EnergyPlus Input Data Dictionary (IDD) file, usually named as `Energy+.idd`, or a valid version of IDD, e.g. `8.9`, `"8.9.0"`.
download	If `TRUE` and argument `idd`, the IDD file will be downloaded from EnergyPlus GitHub Repository, and saved to `tempdir()`. It will be parsed after it is downloaded successfully. A special value of `"auto"` can be given, which will automatically download corresponding IDD file if the Idd object is currently not available. It is useful in case when you only want to edit an EnergyPlus Input Data File (IDF) directly but do not want to install whole EnergyPlus software. Default is `FALSE`.
ver	A valid EnergyPlus version, e.g. `8`, `8.7`, `"8.7"` or `"8.7.0"`. For `download_idd()`, the special value `"latest"`, which is default, means the latest version.
dir	A directory to indicate where to save the IDD file. Default: current working directory.

Details

`use_idd()` takes a valid version or a path of an EnergyPlus Input Data Dictionary (IDD) file, usually named "Energy+.idd" and return an `Idd` object. For details on `Idd` class, please see Idd.

`download_idd()` downloads specified version of EnergyPlus IDD file from EnergyPlus GitHub Repository. It is useful in case where you only want to edit an EnergyPlus Input Data File (IDF) directly but do not want to install whole EnergyPlus software.

`avail_idd()` returns versions of all cached `Idd` object.

`is_avail_idd()` returns `TRUE` if input version of IDD file has been parsed and cached.

eplusr tries to detect all installed EnergyPlus in default installation locations when loading. If argument `idd` is a version, eplusr will try the follow ways sequentially to find corresponding IDD:

- The cached `Idd` object of that version
- `"Energy+.idd"` file distributed with EnergyPlus of that version (see `avail_eplus()`).
- The `"VX-Y-Z-Energy+.idd"` file distributed along with IDFVersionUpdater from the latest EnergyPlus detected.

Value

- `use_idd()` returns an `Idd` object
- `download_idd()` returns an invisible integer `0` if succeed. Also an attribute named `file` which is the full path of the downloaded IDD file;
- `avail_idd()` returns a numeric_version vector or `NULL` if no available Idd object found.
- `is_avail_idd()` returns a single logical vector.

Author(s)

Hongyuan Jia

See Also

Idd Class for parsing, querying and making modifications to EnergyPlus IDD file

Examples

```
## Not run:
# get all available Idd version
avail_idd()

# check if specific version of Idd is available
is_avail_idd(8.5)

# download latest IDD file from EnergyPlus GitHub repo
str(download_idd("latest", tempdir()))

# use specific version of Idd
# only works if EnergyPlus v8.8 has been found or Idd v8.8 exists
use_idd(8.8)

# If Idd object is currently not avail_idd, automatically download IDD file
# from EnergyPlus GitHub repo and parse it
use_idd(8.8, download = "auto")

# now Idd v8.8 should be available
is_avail_idd(8.8)

# get specific version of parsed Idd object
use_idd(8.8)

avail_idd() # should contain "8.8.0"

## End(Not run)
```

`version_updater` *Run IDFVersionUpdater to Update Model Versions*

Description

`version_updater()` is a wrapper of IDFVersionUpdater preprocessor distributed with EnergyPlus. It takes a path of IDF file or an Idf object, a target version to update to and a directory to save the new models.

Usage

```
version_updater(idf, ver, dir = NULL, keep_all = FALSE)
```

Arguments

`idf`	An Idf object or a path of IDF file.
`ver`	A valid EnergyPlus IDD version, e.g. `9`, `8.8`, or `"8.8.0"`.
`dir`	The directory to save the new IDF files. If the directory does not exist, it will be created before save. If `NULL`, the directory of input Idf object or IDF file will be used. Default: `NULL`.
`keep_all`	If `TRUE`, a list will be return which contains all Idf objects of intermediate versions. The list will be named using first two number of that version, e.g. `8.1`, `8.2`. If `FALSE`, only the Idf object of the version specified by `ver` will be returned. Default: `FALSE`.

Details

An attribute named `errors` is attached which is a list of ErrFiles that contain all error messages from transition error (.VCpErr) files.

Value

An Idf object if `keep_all` is `FALSE` or a list of Idf objects if `keep_all` is `TRUE`. An attribute named `errors` is attached which contains all error messages from transition error (.VCpErr) files.

Author(s)

Hongyuan Jia

Examples

```
## Not run:
if (any(avail_eplus()) > 7.2) {
    # create an empty IDF
    idf <- empty_idf(7.2)
    idf$save(tempfile(fileext = ".idf"))
```

```
    # convert it from v7.2 to the latest EnergyPlus installed
    updated <- version_updater(idf, max(avail_eplus()))

    # convert it from v7.2 to the latest EnergyPlus installed and keep all
    # intermediate versions
    updated <- version_updater(idf, max(avail_eplus()), keep_all = TRUE)

    # see transition error messages
    attr(updated, "errors")
}

## End(Not run)
```

`with_option` *Evaluate an expression with temporary eplusr options*

Description

These functions evaluate an expression with temporary eplusr options

Usage

```
with_option(opts, expr)

with_silent(expr)

with_verbose(expr)

with_speed(expr)

without_checking(expr)
```

Arguments

`opts`	A list of valid input for `eplusr::eplusr_option()`.
`expr`	An expression to be evaluated.

Details

`with_option` evaluates an expression with specified eplusr options.

`with_silent` evaluates an expression with no verbose messages.

`with_verbose` evaluates an expression with verbose messages.

`without_checking` evaluates an expression with no checkings.

`with_speed` evaluates an expression with no checkings and autocompletion functionality.

Examples

```
## Not run:
path_idf <- system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr")

# temporarily disable verbose messages
idf <- with_silent(read_idf(path_idf, use_idd(8.8, download = "auto")))

# temporarily disable checkings
without_checking(idf$'BuildingSurface:Detailed' <- NULL)
# OR
with_option(list(validate_level = "none"), idf$'BuildingSurface:Detailed' <- NULL)

## End(Not run)
```

Index