# Package 'eplusr'

November 5, 2019

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

**Version** 0.10.4

**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** R6, RSQLite, callr (>= 2.0.4), crayon, cli, data.table (>= 1.9.8), later, lubridate, methods, processx (>= 3.2.0), progress (>= 1.2.0), stringi, units

**Suggests** testthat, covr, pkgdown, knitr, rmarkdown

**Encoding** UTF-8

**LazyData** true

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

**RoxygenNote** 6.1.1

**Collate** 'constants.R' 'assertions.R' 'diagram.R' 'eplusr.R' 'utils.R' 'impl.R' 'parse.R' 'impl-epw.R' 'epw.R' 'err.R' 'format.R' 'impl-idd.R' 'idd.R' 'idd_object.R' 'impl-idf.R' 'idf.R' 'impl-idfobj.R' 'idf_object.R' 'impl-iddobj.R' 'impl-sql.R' 'install.R' 'job.R' 'param.R' 'rdd.R' 'run.R' 'sql.R' 'units.R' 'validate.R' 'zzz.R'

**VignetteBuilder** knitr

**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** 2019-11-05 17:40:02 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.

With eplusr, you can do:

- Read, parse and modify EnergyPlus Weather File (EPW).
- Read and parse EnergyPlus IDF files.
- Query on models, including classes, objects and fields
- Directly add, modify, duplicate, and delete objects of IDF in R.
- Automatically change referred fields when modifying objects.
- Check any possible errors whenever modifications are made.
- Save the changed models into standard formats in the same way as IDFEditor distributed along with EnergyPlus.
- Run your models directly and collect the simulation output of EnergyPlus in R.
- Run parametric analysis in parallel and collect results in one go.

## 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 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:
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:
clean_wd("foo.idf")

## 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,
  autofield = 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. |
| autofield | 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", "la")

## 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

```
if (is_avail_idd(8.8)) empty_idf(8.8)
```

---

EplusJob *Run EnergyPlus Simulation and Collect Outputs*

---

## Description

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

## 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.

## NOTE

When using $run() in Idf class, which internally creates an EplusJob object and calls its $run() method, an object in Output:SQLite with Option Type value of SimpleAndTabular will be automatically created if it does not exists.

However, when creating an EplusJob using eplus_job(), the IDF file is not parsed but directly pass its path to EnergyPlus. Thus, that process of handling Output:SQLite class is not performed. If you want to ensure that the output collection functionality in EplusJob class works successfully, it is recommended to first read that IDF file using read_idf() and then use $run() method in Idf class by doing idf$run().

## Usage

```
job <- eplus_job(idf, epw)
job$version()
job$path(type = c("all", "idf", "epw"))
job$run(wait = TRUE, force = FALSE)
job$kill()
job$status()
job$errors(info = FALSE)
job$output_dir(open = FALSE)
job$locate_output(suffix = ".err", strict = TRUE)
job$list_table()
```

```
job$read_table(name)
job$report_data_dict()
job$report_data(key_value = NULL, name = NULL, year = NULL, tz = "UTC", case = "auto", all = FALSE,
                period = NULL, month = NULL, day = NULL, hour = NULL, minute = NULL,
              interval = NULL, simulation_days = NULL, day_type = NULL, environment_name = NULL)
job$tabular_data(report_name = NULL, report_for = NULL, table_name = NULL, column_name = NULL, row_name
job$print()
```

### Basic info

```
job <- eplus_job(idf, epw)
job$version()
job$path(type = c("all", "idf", "epw"))
```

`$version()` reutrns the version of IDF that current `EplusJob` uses.

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

#### 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.
- `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. Default: `"all"`.

### Run

```
job$run(wait = TRUE, force = FALSE)
job$kill()
job$status()
job$errors(info = FALSE)
```

`$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.

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

`$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.

$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.

#### Arguments

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

### Simulation Output Extraction

```
job$output_dir(open = FALSE)
job$locate_output(suffix = ".err", strict = TRUE)
job$list_table()
job$read_table(table)
job$report_data_dict()
job$report_data(key_value = NULL, name = NULL, year = NULL, tz = "UTC",case = "auto", all = FALSE,
                period = NULL, month = NULL, day = NULL, hour = NULL, minute = NULL,
              interval = NULL, simulation_days = NULL, day_type = NULL, environment_name = NULL)
job$tabular_data(report_name = NULL, report_for = NULL, table_name = NULL, column_name = NULL, row_name
```

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

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

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

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

`$report_data_dict()` returns a 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.

`$report_data()` extracts the report data in a data.table using key values, variable names and other specifications. `$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 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_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 compliance 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.

`$tabular_data()` extracts the tabular data in a data.table using report, table, column and row name specifications. The returned data.table has 8 columns:

- `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**

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

**Arguments**

- `open`: If `TRUE`, the output directory will be opened.
- `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`.
- `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 `$report_data_dict()`. Default: `NULL`.
- `year`: Year of the date time in column `datetime`. If `NULL`, it will calculate a year value that meets the start day of week restriction for each environment. Default: `NULL`.
- `tz`: Time zone of date time in column `datetime`. Default: `"UTC"`.
- `case`: If not `NULL`, a character column will be added indicates the case of this simulation. If `"auto"`, the name of the IDF file without extension is used.
- `all`: If `TRUE`, extra columns are also included in the returned data.table.
- `period`: A Date or POSIXt vector used to specify which time period to return. The year value does not matter and only month, day, hour and minute value will be used when subsetting. If `NULL`, all time period of data is returned. Default: `NULL`.
- `month`, `day`, `hour`, `minute`: Each is an integer vector for month, day, hour, minute subsetting of `datetime` column when querying on the SQL database. If `NULL`, no subsetting is performed on those components. All possible `month`, `day`, `hour` and `minute` can be obtained using `$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. All possible `environment_name` for current simulation output can be obtained using `$read_table("EnvironmentPeriods")`. If `NULL`, all environment data 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.

## Printing

```
job$print()
print(job)
```

`$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.

## Author(s)

Hongyuan Jia

## See Also

ParametricJob class for EnergyPlus parametric simulations.

## 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
    file.copy(c(idf_path, epw_path), tempdir())

    # create an EplusJob from local an IDF and an EPW file
    job <- eplus_job(file.path(tempdir(), idf_name), file.path(tempdir(), epw_name))

    # get paths of idf and epw
    job$path("all")
    job$path("idf")
    job$path("epw")

    # get current job status
    job$status()

    # check if the job has been run before
    job$status()$run_before

    # run the job in waiting mode
    job$run(wait = TRUE)

    # check the job status again
    job$status()$run_before
    job$status()$successful

    # get output directory
    job$output_dir()

    # open the output directory
    job$output_dir(open = TRUE)

    # check simulation errors
```

```
    job$errors()

    # check simulation errors, only including warnings and errors
    job$errors(info = FALSE)

    # get the file path of an output with a given suffix
    job$locate_output(".err")

    # give an error when simulation did not complete successfully or that file
    # does not exist
    job$locate_output(".exe", strict = TRUE)

    # retrieve simulation results will fail if there is no EnergyPlus SQL output.
    job$report_data_dict()

    # instead, using `$run()` method in Idf class, which will add an
    # `Output:SQLite` object automatically
    idf <- read_idf(file.path(tempdir(), idf_name))
    job <- idf$run(file.path(tempdir(), epw_name), dir = NULL)

    # get report data dictionary
    str(job$report_data_dict())

    # extract all report data
    str(job$report_data())

    # extract some report variable
    str(job$report_data(name = "EnergyTransfer:Building", case = NULL))

    # add a "case" column in the returned data.table
    str(job$report_data(name = "EnergyTransfer:Building", case = "Test"))

    # change the format of datetime column in the returned data.table
   str(job$report_data(name = "EnergyTransfer:Building", year = 2016L, tz = Sys.timezone()))

    # get all tabular data
    str(job$tabular_data())
}

## 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_bottom"`. 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`.

## 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.

## Usage

```
epsql <- eplus_sql(sql)
epsql$path()
epsql$path_idf()
epsql$list_table()
epsql$read_table(table)
epsql$report_data_dict()
epsql$report_data(
   key_value = NULL, name = NULL, year = NULL, tz = "UTC", case = "auto", all = FALSE,
    period = NULL, month = NULL, day = NULL, hour = NULL, minute = NULL,
   interval = NULL, simulation_days = NULL, day_type = NULL, environment_name = NULL
)
job$tabular_data(report_name = NULL, report_for = NULL, table_name = NULL, column_name = NULL, row_name
epsql$print()
print(epsql)
```

### Basic Info

```
epsql <- eplus_sql(sql)
epsql$path()
epsql$path_idf()
```

`$path()` returns the path of EnergyPlus SQLite file.

`$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.

**Arguments**:

- `epsql`: An `EplusSQL` object.
- `sql`: A path to an local EnergyPlus SQLite output file.

### Read Tables

```
epsql$list_table()
epsql$read_table(table)
epsql$report_data_dict()
epsql$report_data(key_value = NULL, name = NULL, year = NULL, tz = "UTC",case = "auto", all = FALSE,
                  period = NULL, month = NULL, day = NULL, hour = NULL, minute = NULL,
                interval = NULL, simulation_days = NULL, day_type = NULL, environment_name = NULL)
epsql$tabular_data(report_name = NULL, report_for = NULL, table_name = NULL, column_name = NULL, row_na
```

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

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

`$report_data_dict()` returns a 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.

`$report_data()` extracts the report data in a data.table using key values, variable names and other specifications. `$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 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_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.

$tabular_data() extracts the tabular data in a data.table using report, table, column and row name specifications. The returned data.table has 8 columns:

- 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**

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

**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.
- 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. All possible environment_name for current simulation output can be obtained using $read_table("EnvironmentPeriods"). If NULL, all environment data 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.

## Print

```
epsql$print()
print(epsql)
```

$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.

### Arguments

- epsql: 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())

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

    # get sql file path
    sql$path()

    # get the parent IDF file path
    sql$path_idf()

    # list all tables in the sql file
    sql$list_table()

    # read a specific table
    sql$read_read("Zones")

    # read report data dictionary
    sql$report_data_dict()

    # read report data
    sql$report_data(name = "EnergyTransfer:Building")

    # read tabular data
    sql$tabular_data()
}

## End(Not run)
```

---

eplus_job *Create an EnergyPlus Simulation Job*

---

## Description

`eplus_job()` takes an IDF and EPW as input, and returns an `EplusJob` object for running EnergyPlus simulation and collecting outputs. For more details, please see EplusJob.

## 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. |

## Value

An `EplusJob` object.

## Author(s)

Hongyuan Jia

## See Also

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

## Examples

```
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
    eplus_job(idf_path, epw_path)

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

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())

    # 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 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.

## Usage

```
epw <- read_epw(path)
epw$location(city, state_province, country, data_source, wmo_number, latitude, longitude, time_zone, e
epw$design_condition()
epw$typical_extreme_period()
epw$ground_temperature()
epw$holiday(leapyear, dst, holiday)
epw$comment1(comment)
epw$comment2(comment)
epw$num_period()
epw$interval()
epw$period(period, name, start_day_of_week)
epw$missing_code()
epw$initial_missing_value()
epw$range_exist()
epw$range_valid()
epw$fill_action()
epw$data(period = 1L, start_year = NULL, tz = "UTC", update = FALSE)
epw$abnormal_data(period = 1L, cols = NULL, keep_all = TRUE, type = c("both", "missing", "out_of_range")
epw$redundant_data()
epw$make_na(period = NULL, missing = FALSE, out_of_range = FALSE)
epw$fill_abnormal(period = NULL, missing = FALSE, out_of_range = FALSE, special = FALSE)
epw$add_unit()
epw$drop_unit()
epw$purge()
epw$add(data, realyear = FALSE, name = NULL, start_day_of_week = NULL, after = 0L, warning = TRUE)
epw$set(data, realyear = FALSE, name = NULL, start_day_of_week = NULL, period = 1L, warning = TRUE)
epw$delete(period)
epw$clone(deep = TRUE)
epw$is_unsaved()
epw$save(path, overwrite = FALSE)
epw$print()
print(epw)
```

### Read

```
epw <- read_epw(path)
```

**Arguments**

- path: Path of an EnergyPlus EPW file.

### Query and Modify Header

#### LOCATION Header:

```
epw$location(city, state_province, country, data_source, wmo_number, latitude, longitude, time_zone,
```

$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.

**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).

#### DESIGN CONDITION Header:

```
epw$design_condition()
```

$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
- extreme: A list, usually of length 16, of the extreme design conditions

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

#### TYPICAL/EXTREME Header:

```
epw$typical_extreme_period()
```

$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

**GROUND TEMPERATURE Header:**

```
epw$ground_temperature()
```

$ground_temperature() returns the parsed values of GROUND TEMPERATURE header in a data.table format with 7 columns:

- index: Integer type. The index of ground temperature record
- depth: Numeric type. The depth of the ground temperature is measured
- month: Integer type. The month when 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
- temperature: Numeric type. The measured group temperature

**HOLIDAYS/DAYLIGHT SAVINGS Header:**

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

$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.

**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")).

### COMMENT1 and COMMENT2 Header:

```
epw$comment1(comment)
epw$comment2(comment)
```

$comment1() and $comment2() both takes a single string of new comments and replaces the old comment with input one. If no input is given, current comment is returned.

**Arguments**:

- comment: A string of new comments.

### DATA PERIODS Header:

```
epw$num_period()
epw$interval()
epw$period(period, name, start_day_of_week)
```

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

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

$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.

$period() 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: Integer 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.

**Arguments**:

- index: 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.

### Weather Data Specifications

```
epw$missing_code()
epw$initial_missing_value()
epw$range_exist()
epw$range_valid()
epw$fill_action(type = c("missing", "out_of_range"))
```

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

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

$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.

$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.

$fill_action() returns a list containing actions that EnergyPlus and also $fill_abnormal() in Epw class 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.

**Arguments**:

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

### Query Weather Data

```
epw$data(period = 1L, start_year = NULL, tz = "UTC", update = FALSE)
epw$abnormal_data(period = 1L, cols = NULL, keep_all = TRUE, type = c("both", "missing", "out_of_range")
epw$redundant_data()
```

$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`. Also, 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. An error will also be issued if applying specified time zone specified using `tz` introduces invalid date time.

`$abnormal_data()` returns abnormal data of specific data period. Basically, there are 2 types of abnormal data in `Epw`, i.e. missing values and out-of-range values. Sometimes, it may be useful to extract and inspect those data especially when inserting measured weather data. `$abnormal_data()` does this.

`$redundant_data()` returns weather data in `Epw` object that do not belong to any data period. This data can be further removed using `$pruge()` method described below.

For `$abnormal_data()` and `$redundant_data()`, a new column named `line` is created indicating the line numbers where abnormal data and redundant data occur in the actual EPW file.

**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`.
- `cols`: A character vector identifying what data columns, i.e. all columns except `datetime`, `year`, `month`, `day`, `hour` and `minute`, 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"`.

### Modify Weather Data In-Place

```
epw$make_na(period = NULL, missing = FALSE, out_of_range = FALSE)
epw$fill_abnormal(period = NULL, missing = FALSE, out_of_range = FALSE, special = FALSE)
epw$add_unit()
epw$drop_unit()
epw$purge()
```

**Note** that all these 5 methods modify the weather data in-place, meaning that the returned data from `$data()` and `$abnormal_data()` may be different after calling these methods.

`$make_na()` converts specified abnormal data into `NA`s in specified data period. This makes it easier to find abnormal data directly using `is.na()` instead of using `$missing_code()`.

`$fill_abnormal()` fills specified abnormal data using corresponding actions listed in `$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.

`$add_unit()` assigns units to numeric weather data using `units::set_units()` if applicable.

`$drop_unit()` removes all units of numeric weather data.

Similarly, `$add_unit()` and `$drop_unit()` are reversible, i.e. `$add_unit()` can be used to counteract the effects introduced by `$drop_unit()`, and vise a versa.

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

**Arguments**:

- `period`: A positive integer vector identifying the data period indexes. Data periods information can be obtained using `$period()` described above. If `NULL`, all data periods are included. Default: `NULL`.
- `missing`: If `TRUE`, missing values are included. Default: `FALSE`.
- `out_of_range`: If `TRUE`, out-of-range values are included. Default: `FALSE`.
- `special`: If `TRUE`, abnormal data are filled using corresponding actions listed `$fill_action()`. If `FALSE`, all abnormal data are fill with missing code described in `$missing_code()`.

### Set Weather Data

```
epw$add(data, realyear = FALSE, name = NULL, start_day_of_week = NULL, after = 0L, warning = TRUE)
epw$set(data, realyear = FALSE, name = NULL, start_day_of_week = NULL, period = 1L, warning = TRUE)
epw$delete(period)
```

`$add()` adds a new data period into current `Epw` object at specified position.

`$set()` replaces existing data period using input new weather data.

The validity of input data is checked before adding or setting 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.

`$delete(period)` removes one specified data period.

**Arguments**:

- `data`: A 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`.
- `period`: A single integer identifying the index of data period to set.
- `warning`: If `TRUE`, warnings are given if any missing data, out-of-range data are found. Default: `TRUE`.

### Save

```
epw$is_unsaved()
epw$save(path, overwrite = FALSE, purge = FALSE)
```

`$is_unsaved()` returns `TRUE` if there are any modifications to the `Epw` object since it was saved or since it was created if not saved before.

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

**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`.

### Clone

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

`$clone()` copies and returns the cloned `Epw` object. Because `Epw` uses `R6Class` under the hook which has "modify-in-place" semantics, `epw_2 <-epw_1` does not copy `epw_1` at all but only create a new binding to `epw_1`. Modify `epw_1` will also affect `epw_2` as well, as these two are exactly the same thing underneath. In order to create a complete cloned copy, please use `$clone(deep = TRUE)`.

**Arguments**

- deep: Has to be TRUE if a complete cloned copy is desired. Default: TRUE.

## Print

```
epw$print()
print(epw)
```

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

## Author(s)

Hongyuan Jia

## Examples

```
## 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)) {
    epw_path <- file.path(
        eplus_config(8.8)$dir,
        "WeatherData",
        "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"
    )
    epw <- read_epw(path_epw)
}

# get file path
epw$path()

# get header data
epw$location()
epw$location()
epw$design_condition()
epw$typical_extreme_period()
epw$ground_temperature()
epw$holiday()
epw$comment1()
epw$comment2()
epw$num_period()
epw$interval()
epw$period()

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

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

# modify data period name
epw$period(1, name = "test")

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

# get data specifications
epw$missing_code()
epw$initial_missing_value()
epw$range_exist()
epw$range_valid()
epw$fill_action()

# 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)

# 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)

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

# change the weather data
epw$set(epw$data())
# save the weather file
epw$save(file.path(tempdir(), "weather.epw"))

## End(Not run)
```

---

`format.IddObject` *Format an IddObject*

---

## Description

Format an IddObject into a string of an empty object of current class. It is formatted exactly the same as in IDF Editor.

## Usage

```
## S3 method for class 'IddObject'
format(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 single length character vector.

## Examples

```
## Not run:
cat(format(use_idd(8.8, download = "auto")$Materal, leading = 0))

## 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.

## Usage

```
idd <- use_idd(ver)
idd$version()
idd$build()
idd$group_index(group = NULL)
idd$group_name()
idd$from_group(class)
idd$class_index(class = NULL, by_group = FALSE)
idd$class_name(index = NULL, by_group = FALSE)
idd$required_class_name()
idd$unique_class_name()
idd$extenesible_class_name()
idd$is_valid_group(group)
idd$is_valid_class(class)
idd$object_relation(class, direction = c("all", "ref_by", "ref_to"))
idd$object(class)
idd$objects(class)
idd$objects_in_relation(class, direction = c("ref_to", "ref_by"))
idd$objects_in_group(group)
idd$ClassName
idd[[ClassName]]
idd$to_table(class, all = FALSE)
idd$to_string(class, leading = 4L, sep_at = 29L, sep_each = 0L, all = FALSE)
idd$print()
print(idd)
```

## Arguments

- `ver`: A valid EnergyPlus IDD version, e.g. 8.8, "8.6.0".

- idd: An Idd object.
- group: A character vector of valid group names. For $objects_in_group(), a single string of valid group name.
- index: An integer vector giving indexes of name appearance in the IDD file of specified classes.
- class: A character vector of valid class names. For $object_relation() and $objects_in_relation(), a single string of valid class name.
- ClassName: A single string of valid class name.
- direciton: The relation direction to extract. Should be either "all", "ref_to" or "ref_by". For $objects_in_relation(), only "ref_to" and "ref_by" are acceptable.
- all: If TRUE, all fields in specified classes are returned. If FALSE, only minimum required fields are returned. Default: FALSE.
- 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.
- by_group: If TRUE, a list is returned which separates class indexes or names by the group they belongs to. Default: FALSE.

## Detail

$version() returns the IDD version in numeric_version.

$build() returns the build tag string.

$group_index() returns integer indexes (indexes of name appearance in the IDD file) of specified groups.

$group_name() returns all group names.

$from_group() returns the names of group that specified classes belongs to.

$is_valid_group() return TRUE if the input is a valid group name.

$class_index() returns integer indexes (indexes of name appearance in the IDD file) of specified classes. If by_group is TRUE, a list is returned which separate class indexes by the group they belong to. Otherwise, an integer vector is returned.

$class_name() returns class names of specified class indexes. If by_group is TRUE, a list is returned which separate class names by the group they belong to. Otherwise, a character vector is returned.

$required_class_name() returns the names of all required classes.

$unique_class_name() returns the names of all unique-object classes.

$extensible_class_name() returns the names of all extensible classes.

$is_valid_class() return TRUE if the input is a valid class name.

$object() returns an IddObject of specified class.

$objects() returns a list of IddObjects of specified classes.

`$object_relation()` returns an `IddRelation` object which contains field data that have relation with specified class. For instance, if `idd$object_relation("Class 1","ref_to")` gives results below:

```
-- Refer to Others ---------------------
  Class: <Class 1>
  +- Field: <1: Field 1>
  |  v~~~~~~~~~~~~~~~~~~
  |  \- Class: <Class 2>
  |     \- Field: <2: Field 2>
  |
  \- Field: <2: Field 2>
```

This means that `Field 2` in class `Class 1` does not refer to any other fields. But `Field 1` in class `Class 2` refers to `Field 2` in class named `Class 2`.

`$objects_in_relation()` returns a list of IddObjects that have relations. The first element is always the IddObject of that class. If specified class does not have any relation with other classes, a list that only contains the IddObject of that class is returned. For instance, `idd$objects_in_relation("Construction","ref_` will return a list of an IddObject of class `Construction` and also all IddObjects that class `Construction` refers to; similarly, `idd$objects_in_relation("Construction","ref_by")` will return a list of an IddObject of class `Construction` and also all IddObjects that refer to class `Construction`.

`$objects_in_group()` returns a list of IddObjects of specified group.

eplusr also provides custom S3 method of `$` and `[[` to make it more convenient to get a single IddObject. Basically, `idd$ClassName` and `idd[[ClassName]]`, is equivalent to `idd$object(ClassName)[[1]]`. Here, `ClassName` is a single valid class name where all characters other than letters and numbers are replaced by a underscore `_`.

For details about IddObject, please see IddObject class.

`$to_table()` returns a data.table that contains core data of specified class. It has 3 columns:

- `class`: Character type. Class names.
- `index`: Integer type. Field indexes.
- `field`: Character type. Field names.

`$to_string()` returns empty objects of specified class in a character vector format. It is formatted exactly the same as in IDF Editor.

`$print()` prints basic info ob current Idd object, including version, build tag and total class number.

## Author(s)

Hongyuan Jia

## References

IDFEditor, OpenStudio utilities library

## See Also

IddObject class which provides detailed information of curtain class

## Examples

```
## Not run:
# get the Idd object of EnergyPlus v8.8
idd <- use_idd(8.8, download = "auto")

# version
idd$version()

# build
idd$build()

# all group names
str(idd$group_name())

# all class names
str(idd$class_name())

# all required class names
str(idd$required_class_name())

# all unique class names
str(idd$unique_class_name())

# IddObject of SimulationControl class
idd$object("SimulationControl")
# OR
idd$SimulationControl
# OR
idd[["SimulationControl"]]

# IddObject of Construction and Material class
idd$objects(c("Construction", "Material"))

# show all classes that refer to Material class
length(idd$object_relation("Material", "ref_by"))

# IddObjects that refer to class Construction
length(idd$objects_in_relation("Construction", "ref_by"))

# IddObjects that class Construction refers to
length(idd$objects_in_relation("Construction", "ref_to"))

# All IddObjects in group Schedules
length(idd$objects_in_group("Schedules"))

# Extract core data of class Material and Construction
idd$to_table(c("Material", "Construction"))

# Get empty Material object and Construction object in a character vector
idd$to_string(c("Material", "Construction"))

## 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.

## Usage

```
iddobj <- idd$object(class)
iddobj <- idd_object(idd, class)
iddobj$version()
iddobj$group_name()
iddobj$group_index()
iddobj$class_name()
iddobj$class_index()
iddobj$class_format()
iddobj$min_fields()
iddobj$num_fields()
iddobj$memo()
iddobj$has_name()
iddobj$is_required()
iddobj$is_unique()
iddobj$is_extensible()
iddobj$num_extensible()
iddobj$first_extensible_index()
iddobj$extensible_group_num()
iddobj$add_extensible_group(num = 1L)
iddobj$del_extensible_group(num = 1L)
iddobj$field_name(index = NULL, unit = FALSE, in_ip = eplusr_option("view_in_ip"))
iddobj$field_index(name = NULL)
iddobj$field_type(which = NULL)
iddobj$field_note(which = NULL)
iddobj$field_unit(which = NULL, in_ip = eplusr_option("view_in_ip"))
iddobj$field_default(which = NULL, in_ip = eplusr_option("view_in_ip"))
iddobj$field_choice(which = NULL)
iddobj$field_range(which = NULL)
iddobj$field_relation(which = NULL, type = c("all", "ref_by", "ref_to"))
iddobj$field_possible(which = NULL)
```

```
iddobj$is_valid_field_num(num)
iddobj$is_extensible_index(index)
iddobj$is_valid_field_name(name)
iddobj$is_valid_field_index(which)
iddobj$is_autosizable_field(which = NULL)
iddobj$is_autocalculatable_field(which = NULL)
iddobj$is_numeric_field(which = NULL)
iddobj$is_integer_field(which = NULL)
iddobj$is_real_field(which = NULL)
iddobj$is_required_field(which = NULL)
iddobj$has_ref(which = NULL)
iddobj$has_ref_to(which = NULL)
iddobj$has_ref_by(which = NULL)
iddobj$to_table(all = FALSE)
iddobj$to_string(comment = NULL, leading = 4L, sep_at = 29L, all = FALSE)
iddobj$print(brief = FALSE)
print(iddobj)
```

### Basic

```
iddobj <- idd$object(class)
iddobj <- idd_object(idd, class)
iddobj$version()
```

An `IddObject` can be created from the parent Idd object, using `$object()`, idd_object and other equivalent.

`$version()` returns the version of parent IDD current object belongs to.

#### Arguments

- `idd`: An Idd object.
- `class`: A valid class name (a string).
- `iddobj`: An IddObject object.

### Class Property

```
iddobj$group_name()
iddobj$group_index()
iddobj$class_name()
iddobj$class_index()
iddobj$class_format()
iddobj$min_fields()
iddobj$num_fields()
iddobj$memo()
iddobj$has_name()
iddobj$is_required()
iddobj$is_unique()
```

`$group_index()` returns the index of IDD group it belongs to.

`$group_name()` returns the name of IDD group it belongs to.

`$class_index()` returns the index of this IDD class.

`$class_name()` returns the name of this IDD class.

`$class_format()` returns the format of this IDD class. This format indicator is currently not used by eplusr. **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.

`$min_fields()` returns the minimum fields required for this class. If no required, `0` is returned.

`$num_fields()` returns current total number of fields in this class. This number may change if the class is extensible and after `$add_extensible_group()` or `$del_extensible_group()`.

`$memo()` returns memo of this class. Usually a brief description of this class.

`$has_name()` return `TRUE` if this class has name attribute.

`$is_unique()` return `TRUE` if this class is unique.

`$is_required()` returns `TRUE` if this class is required.

### Extensible Group

```
iddobj$is_extensible()
iddobj$num_extensible()
iddobj$first_extensible_index()
iddobj$extensible_group_num()
iddobj$add_extensible_group(num = 1L)
iddobj$del_extensible_group(num = 1L)
```

`$is_extensible()` returns `TRUE` if this class is extensible.

`$num_extensible()` returns the number of extensible fields in this class. If not zero, it means that objects in this class is dynamically extensible.

`$first_extensible_index()` returns the field index of the first extensible field in this class. If this class is not extensible, `0` is return.

`$extensible_group_num()` returns the number of extensible groups in this class.

`$add_extensible_groups()` adds extensible groups in this class.

`$del_extensible_groups()` deletes extensible groups in this class.

#### Arguments

- `num`: A positive integer of how many extensible groups to add or delete. Default: `1`.

### Field Property

```
iddobj$field_name(index = NULL, unit = FALSE, in_ip = eplusr_option("view_in_ip"))
iddobj$field_index(name = NULL)
iddobj$field_type(which = NULL)
iddobj$field_note(which = NULL)
iddobj$field_unit(which = NULL, in_ip = eplusr_option("view_in_ip"))
iddobj$field_default(which = NULL, in_ip = eplusr_option("view_in_ip"))
iddobj$field_choice(which = NULL)
```

```
iddobj$field_range(which = NULL)
iddobj$field_relation(which = NULL, type = c("all", "ref_by", "ref_to"))
iddobj$field_possible(which = NULL)
iddobj$is_valid_field_num(num)
iddobj$is_extensible_index(index)
iddobj$is_valid_field_name(name)
iddobj$is_valid_field_index(which)
iddobj$is_autosizable_field(which = NULL)
iddobj$is_autocalculatable_field(which = NULL)
iddobj$is_numeric_field(which = NULL)
iddobj$is_integer_field(which = NULL)
iddobj$is_real_field(which = NULL)
iddobj$is_required_field(which = NULL)
iddobj$has_ref(which = NULL)
iddobj$has_ref_to(which = NULL)
iddobj$has_ref_by(which = NULL)
```

`$field_name()` returns names of fields specified by field indexes. If `index` is `NULL`, names of all fields in this class are returned. If `lower` is `TRUE`, "lower-style" names are returned, i.e. all spaces and dashes is replaced by underscores. "lower-style" names are useful when use them as filed names in `$set_value()` in `IdfObject` class and `$set_object()` in `Idf` class. If `unit` is `TRUE`, the units of those fields are also returned. If `in_ip`, corresponding imperial units are returned. It only has effect when `unit` is `TRUE`.

`$field_index()` returns indexes of fields specified by field names. If `name` is `NULL`, indexes of all fields in this class are returned.

All other `$field_*()` returns specific field properties. If `which` is `NULL`, properties of all fields in this class are returned.

`$field_type()`: returns field types. 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) and `"node"` (name used in connecting HVAC components).

`$field_unit()`: returns a character vector of field units. If `in_ip` is `TRUE`, IP unites are returned.

`$field_default()`: returns a list of default values of those fields. If no defaults found, `NA`s are returned.

`$field_choice()`: returns a list of all valid choices for those fields. If no choices found, `NA`s are returned.

`$field_range()`: returns a list of ranges for those fields. 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.

`$field_relation()`: returns a list of references for those fields that have the `object-list` and/or `reference` and `reference-class-name` attribute. Basically, it is a list of two elements `ref_to` and `ref_by`. Underneath, `ref_to` and `ref_by` are data.tables which contain source field data and reference field data with custom printing method. For instance, if `iddobj$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`.

`$field_possible()`: returns all possible values for specified fields, including auto-value (`Autosize`, `Autocalculate`, and `NA` if not applicable), and results from `$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"`.

`$is_valid_field_num()` returns `TRUE` if `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.

`$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"`.

`$is_valid_field_index()` returns `TRUE` if `index` is a valid field index.

`$is_autosizable_field()` returns `TRUE` if the field can be assigned to `autosize`.

`$is_autocalculatable_field()` returns `TRUE` if the field can be assigned to `autocalculate`.

`$is_numeric_field()` returns `TRUE` if the field value should be numeric ( an integer or a real number).

`$is_integer_field()` returns `TRUE` if the field value should be an integer.

`$is_real_field()` returns `TRUE` if the field value should be a real number but not an integer.

`$is_required_field()` returns `TRUE` if the field is required.

`$has_ref()` returns `TRUE` if the field refers to or can be referred by other fields.

`$has_ref_to()` returns `TRUE` if the field refers to other fields.

`$has_ref_by()` returns `TRUE` if the field refers can be referred by other fields.

**Arguments**

- `index`: An integer vector of field indexes.
- `name`: A character vector or field names. Can be given in underscore style, e.g. `"Thermal Resistance"` can be given in format `"thermal_resistance"`.
- `which`: An integer vector of field indexes or a character vector of field names. Field names can be given in underscore style.
- `unit`: If `TRUE`, field units will be pasted after field names, just like the way IDF Editor does. Default: `FALSE`.
- `in_ip`: If `TRUE`, field names or values will be returned in IP units. Default: the value of `eplusr_option("view_in_ip")`.
- `type`: The direction of relation to search. Should be one of `"all"`, `"ref_by"` and `"ref_to"`. If `"ref_by"`, the relation data of specified fields and fields that refer to specified fields is returned. If `"ref_to"`, the relation data of specified fields and fields that are referred by specified fields is returned. If `"all"`, both are returned.

## Data Extraction

```
iddobj$to_table(all = FALSE)
iddobj$to_string(comment = NULL, leading = 4L, sep_at = 29L, all = FALSE)
```

`$to_table()` returns a data.table that contains core data of current class. It has 3 columns:

- `class`: Character type. Current class name.
- `index`: Integer type. Field indexes.
- `field`: Character type. Field names.

`$to_string()` returns an empty object of current class in a character vector format. It is formatted exactly the same as in IDF Editor.

**Arguments**

- `all`: If `TRUE`, all fields in current class are returned, otherwise only minimum fields are returned.
- `unit`: If `TRUE`, units are also returned. Default: `FALSE`.
- `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: `4L`.
- `sep_at`: The character width to separate value string and field string. Default: `29L` which is the same as IDF Editor.

### Print

`$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 bullet marks. 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.

**Argument**:

- `brief`: If `TRUE`, only class name part is printed. Default: `FALSE`.

### Author(s)

Hongyuan Jia

### See Also

Idd Class

### Examples

```
## Not run:
# ===== CREATE =====
# get a parent Idd object
idd <- use_idd(8.8, download = "auto")

# get an IddObject of class "Material"
mat <- idd$Material
# OR
mat <- idd_object(idd, "Material")

# ===== BASIC INFO =====
# get the version of parent IDD
mat$version()

# ===== CLASS PROPERTY =====
# get name of IDD group it belongs to
mat$group_name()

# get index of IDD group it belongs to
mat$group_index()

# get name of current IDD class
mat$class_name()
```

```
# get index of current IDD class
mat$class_index()

# get the format of current IDD class
mat$class_format()

# get minimum field number
mat$min_fields()

# get total field number
mat$num_fields()

# get memo of current class
mat$memo()

# check if current class has name attribute or not
mat$has_name()

# check if current class is required
mat$is_required()

# check if current class is unique
mat$is_unique()

# ===== EXTENSIBLE GROUP =====
# get an IddObject of extensible class "Branch"
bran <- idd$Branch

# check if the class is extensible
bran$is_extensible()

# get number of extensible fields, index of first extensible field and number of
# current extensible groups in "Branch" class
bran$num_extensible()

bran$first_extensible_index()

bran$extensible_group_num()

# get current number of fields
bran$num_fields()

# add ten extensible groups
bran$add_extensible_group(10)
# the number of fields has been increased by 10 * 4 (= 40)
bran$num_fields()

# delete eight extensible groups
bran$del_extensible_group(8)
# the number of fields has been decreased by 8 * 4 (= 32)
bran$num_fields()

# ===== FIELD PROPERTY =====
```

```
# list all field names without units
mat$field_name()

# get field indexes
mat$field_index(c("thickness", "roughness", "name"))

# get field types
mat$field_type(c("solar_absorptance", "Density", "Name"))

# get field notes
bran$field_note(c(2, 4))

# get field SI units
mat$field_unit(c(1,3,5), in_ip = FALSE)

# get field IP units
mat$field_unit(c(1,3,5), in_ip = TRUE)

# get field default values in SI units
str(mat$field_default(in_ip = FALSE))

# get field choices
str(mat$field_choice(1:3))

# get field ranges
mat$field_range(c("roughness", "thickness", "conductivity", "solar_absorptance"))

# get field relation with other fields
mat$field_relation(type = "all")

# get all possible values of fields
mat$field_possible()

# check if input is a valid field number for current class
## get required minimum field number
mat$min_fields()

# (1) if less than required minimum field number
mat$is_valid_field_num(3)

# (2) if larger than required minimum field number but less than total field
# number
mat$is_valid_field_num(7)

# (3) if larger than total field number
mat$is_valid_field_num(10)
# [1] FALSE

# for extensible class
bran$num_fields()
bran$num_extensible()
# if larger than required minimum field number
# (1) but cannot give whole extensible groups
```

```
bran$is_valid_field_num(c(55, 57, 60))

# (2) and can give whole extensible groups
bran$is_valid_field_num(c(58, 62, 70))

# check if input field index is an extensible field index
bran$is_extensible_index(1:4)

# check if input is valid field name
# NOTE: lower-style names are treated as valid
mat$is_valid_field_name(c("nAmE", "specific heat", "Specific Heat", "specific_heat"))

# check if input is valid field index
bran$is_valid_field_index(c(1, 4, 54, 57))

# check if fields are autosizable, i.e. can be set to "Autosize"
mat$is_autosizable_field(1:4)

# check if fields are autocalculatable, i.e. can be set to "Autocalculate"
mat$is_autocalculatable_field(1:4)

# check if fields are numeric fields, i.e. field values should be either
# integers or float numbers
mat$is_numeric_field(c("roughness", "thickness", "density"))

# check if fields are integer fields, i.e. field values should be integers
mat$is_integer_field(c("name", "specific_heat"))

# check if fields are real fields, i.e. field values should be real numbers
# but not integers
mat$is_real_field(c("name", "specific_heat"))

# check if fields are required, i.e. field values should not be empty
mat$is_required_field(c("name", "roughness", "solar_absorptance"))

# check if fields refer to or can be referred by other fields
mat$has_ref()

# check if fields refer to other fields
mat$has_ref_to()

# check if fields can be referred by other fields
mat$has_ref_by(which = NULL)

# ===== DATA EXTRACTION =====
# get core data of current class
mat$to_table()

# get an empty string-foramt object of current class
mat$to_string()

# ===== PRINT =====
mat$print()
```

```
## 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.

## Overview

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. Below is the detailed documentation on each method.

## Usage

```
model <- read_idf(path)
model$version()
model$path()
model$group_name(all = FALSE, sorted = TRUE)
model$class_name(all = FALSE, sorted = TRUE, by_group = FALSE)
model$is_valid_group(group, all = FALSE)
model$is_valid_class(class, all = FALSE)
model$definition(class)
model$object_id(class = NULL, simplify = FALSE)
model$object_name(class = NULL, simplify = FALSE)
model$is_valid_id(id)
model$is_valid_name(name)
model$object_num(class = NULL)
model$object(which)
model$object_unique(class)
model$objects(which)
model$objects_in_class(class)
model$objects_in_group(group)
model$object_relation(which, direction = c("all", "ref_to", "ref_by", "node"), recursive = FALSE, depth
model$objects_in_relation(which, direction = c("ref_to", "ref_by", "node"), class = NULL, recursive = F
model$search_object(pattern, class = NULL, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBytes =
model$ClassName
model[[ClassName]]
model$dup(...)
model$add(..., .default = TRUE, .all = FALSE)
model$set(..., .default = TRUE)
model$del(..., .ref_by = FALSE, .ref_to = FALSE, .recursive = FALSE, .force = FALSE)
model$insert(..., .unique = TRUE)
model$load(..., .unique = TRUE, .default = TRUE)
model$update(..., .default = TRUE)
model$rename(...)
model$paste(in_ip = FALSE, ver = NULL, unique = TRUE)
model$search_value(pattern, class = NULL, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBytes = F
```

```
model$replace_value(pattern, class = NULL, replacement, ignore.case = FALSE, perl = FALSE, fixed = FALSE
model$is_unsaved()
model$validate(level = eplusr_option("validate_level"))
model$is_valid(level = eplusr_option("validate_level"))
model$to_table(which = NULL, class = NULL, string_value = TRUE, unit = FALSE, wide = FALSE)
model$to_string(which = NULL, class = NULL, comment = TRUE, header = TRUE, format = eplusr_option("save_
model$save(path = NULL, format = eplusr_option("save_format"), overwrite = FALSE, copy_external = TRUE)
model$run(weather = NULL, dir = NULL, wait = TRUE, force = FALSE, copy_external = FALSE)
model$clone(deep = TRUE)
model$print(zoom = c("object", "class", "group", "field"), order = TRUE)
print(model)
```

### Basic Info

```
model <- read_idf(path)
model$version()
model$path()
model$group_name(all = FALSE, sorted = TRUE)
model$class_name(all = FALSE, sorted = TRUE, by_group = FALSE)
model$is_valid_group(group, all = FALSE)
model$is_valid_class(class, all = FALSE)
```

`$version()` returns the version of current model in a numeric_version format. This makes it easy to direction compare versions of different model, e.g. `model1$version() > 8.6` or `model1$version() > model2$version()`.

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

`$group_name()` returns all groups the model contains when `all` is `FALSE` or all groups the underlying Idd object contains when `all` is `TRUE`.

`$class_name()` returns all classes the model contains when `all` is `FALSE` or all classes the underlying Idd object contains when `all` is `TRUE`.

`$is_valid_group()` returns `TRUE`s if given group names are valid for current model (when `all` is `FALSE`) or current underlying Idd object (when `all` is `TRUE`).

`$is_valid_class()` returns `TRUE`s if given class names are valid for current model (when `all` is `FALSE`) or underlying Idd object (when `all` is `TRUE`).

**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).
- `all`: If `FALSE`, only values in current `Idf` object will be returned. If `TRUE`, all values in the underlying Idd will be returned. For `$is_valid_group()` and `$is_valid_class()`, `all` equals to `TRUE` means that input group or class names are checked in all existing ones in the underlying Idd object. 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 object. Default: `TRUE`.

- by_group: Only applicable when all or sorted is TRUE. If TRUE, a list is returned which separate class names by the group they belong to.
- group: A character vector of valid group names.
- class: A character vector of valid class names.

### Definition

```
model$definition(class)
```

$definition() returns an IddObject of given class. IddObject contains all data used for parsing and creating an IdfObject. For details, please see IddObject class.

**Arguments**:

- class: A **single** string of valid class name in current IDD.

### Object Info

```
model$object_id(class = NULL, simplify = FALSE)
model$is_valid_id(id)
model$object_name(class = NULL, simplify = FALSE)
model$is_valid_name(name)
model$object_num(class = NULL)
```

$object_id() returns an integer vector (when simplify is TRUE) or a named list (when simplify is FALSE) of all object IDs in specified classes. The returned list is named using specified class names.

$object_name() returns a character vector (when simplify is TRUE) or a named list (when simplify is FALSE) of all object names in specified classes. The returned list is named using specified class names.

$is_valid_id() and $is_valid_name() returns a logical vector whether the given integer vector or character vector contains valid object IDs or names respectively. Note that for $is_valid_name(), object name matching is **case-insensitive**.

$object_num() returns an integer vector of object numbers in specified classes.

**Arguments**:

- id: An integer vector to check.
- name: A character vector to check.
- class: A character vector that contains valid class names. If NULL, all classes in current Idf object are used. Default: NULL.
- simplify: If TRUE, an integer vector (for $object_id()) or a character vector (for $object_name()) is returned. If FALSE, a list with each element being the data per class is returned. If class is NULL, the order of classes returned is the same as that in the underlying Idd object. Default: FALSE.

### Object Relation

```
model$object_relation(which, direction = c("all", "ref_to", "ref_by", "node"), recursive = TRUE, depth
```

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.

`$object_relation()` provides a simple interface to get this kind of relation. It takes a single object ID or name and also a relation direction, and returns an `IdfRelation` object which contains data presenting such relation above. For instance, if `model$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.

**Arguments**:

- `which`: Either a single integer of object ID or a string of object name.
- `direciton`: The relation direction to extract. Should be either `"all"`, `"ref_to"` or "ref_by".
- `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`.
- `depth`: Only applicable when `recursive` is `TRUE`. This is a depth to when searching value relations recursively. If `NULL`, all recursive relations are returned. Default: `1`.

### Object Query

```
model$object(which)
model$objects(which)
model$object_unique(class)
model$objects_in_class(class)
model$objects_in_group(group)
model$objects_in_relation(which, direction = c("ref_to", "ref_by", "node"), class = NULL, recursive = F
model$search_object(pattern, class = NULL, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBytes =
model$ClassName
model[[ClassName]]
```

`$object()` returns an IdfObject specified by an object ID or name. Note that unlike object ID, which is always unique across the whole `Idf` object, sometimes 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 an object name(s) as input.

`$object_unique()` returns the IdfObject in unique-object class, e.g. `SimulationControl` class. This makes it easy to directly extract and modify those unique objects, e.g. `model$object_unique("SimulationContrl")$`
Note that if there are multiple objects in that unique-object class, an error is issued. This makes sure that `$object_unique()` always returns a single IdfObject.

`$objects()` returns a named **list** of IdfObjects specified by object IDs or names.

`$objects_in_class()` returns a named **list** of all IdfObjects in specified class.

`$objects_in_group()` returns a named **list** of all IdfObjects in specified group.

`$objects_in_relation()` returns a named **list** of IdfObjects that have specified relations with given object. The first element of returned list is always the IdfObject of specified object. If that object does not have specified relation with other objects in specified `class`, a list that only contains that IdfObject is returned. For instance, assume that `const` is a valid object name in `Construction` class, `model$objects_in_relation("const","ref_by","BuildingSurface:Detailed")` will return a named list of an IdfObject named `const` and also all other IdfObjects in `BuildingSurface:Detailed` that refer to field values in `const`. Similarly, `model$objects_in_relation("const","ref_to","Material")` will return a named list of an IdfObject named `const` and also all other IdfObjects in `Material` class that `const` refers to. This makes it easy to directly extract groups of related objects and then use `$insert()` method described below to insert them. For example, copy a construction named `const` from an `Idf` object `model1` to another `Idf` object `model2` is simply to do `model2$insert(model1$objects_in_relation("c`

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`.

`$search_object()` returns a named **list** of IdfObjects whose names meet the given pattern in specified classes.

eplusr also provides custom S3 method of `$` and `[[` to make it more convenient to get IdfObjects in specified class. Basically, `model$ClassName` and `model[[ClassName]]`, where `ClassName` is a single valid class name, is equivalent to `model$objects_in_class(ClassName)` if `ClassName` is not an unique-object class and `model$object_unique(ClassName)` if `ClassName` is an unique-object class. For convenience, underscore-style names are allowed, e.g. `BuildingSurface_Detailed` is equivalent to `BuildingSurface:Detailed`. For instance, `model$BuildingSurface_Detailed` and also `model[["BuildingSurface:Detailed"]]` will return all IdfObjects in `BuildingSurface:Detailed` class; `model$Building` and also `model[["Building"]]` will return the IdfObject in `Building` class which is an unique-object class.

**Note**: The returned list from `$objects()`, `$objects_in_class()` and other methods is named using the names of returned IdfObjects in that list. This will makes it easy to using object name to do further subsetting, e.g. `model$objects_in_class("Material")$mat` will return an IdfObject named `mat` in `Material` class, and `model$objects_in_class("Material")[[1]]` will return the first material in `model`. If returned IdfObjects belongs to a class that does not have name attribute, such like `Version`, `SimulationControl` and etc., `NA` is assigned as the name.

IdfObject class provides more detailed methods to modify a single object in an `Idf`. For detailed explanations, please see IdfObject class.

**Arguments**:

- `which`: A single object ID or name for `$object()` and `$objects_in_relation()`; an integer vector of object IDs or a character vector of object names for `$objects()`.
- `class`: A single string of class name for `$object_unique()` and `$objects_in_class()`; a character vector of class names for `$objects_in_relation()` and `$search_object()`.
- `group`: A single string of group name.
- `pattern`, `ignore.case`, `perl`, `fixed` and `useBytes`: All of them are directly passed to base::grepl.
- `ClassName`: A single string of class name. For `[[`, `ClassName` can be an underscore-style class name, where all characters other than letters and numbers are replaced by underscores `_`.
- `direciton`: The relation direction to extract. Should be either `"ref_to"` or "ref_by".
- `recursive`: If `TRUE`, the relation is searched recursively, e.g. one material named `mat` is referred by a construction named `const`, and `const` is also referred by a surface named `surf`, all `mat`, `const` and `surf` are returned.
- `depth`: Only applicable when `recursive` is `TRUE`. This is a depth to when searching value relations recursively. If `NULL`, all recursive relations are returned. Default: `1`.
- `ignore.case`, `perl`, `fixed` and `useBytes`: All are directly passed to base::grepl.

## Object Modification

### Duplicate Objects:

```
model$dup(...)
```

`$dup()` takes integer vectors of object IDs and character vectors of object names, duplicates objects specified, and returns a list of newly created IdfObjects. 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 does not have name attribute.

**Note**:

- Assign 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()`.

**Argument**:

- `...`: Integer vectors of object IDs and character vectors of valid object names. If input has names, they will be used as the names of newly created objects.

**Usage**:

- Without new names: `model$dup(c("name1","name2"),6:10)`.
- With new names: `model$dup(c(new_name1 = "name1",new_name2 = "name2"),new_name3 = 6)`.
- Variable inputs: `a <-c("name1",new_name2 = "name2"); b <-10:20; c <-c(new_name3 = 10); model$dup(a,b,c)`.

### Add Objects:

```
model$add(..., .default = TRUE, .all = FALSE)
```

`$add()` takes object definitions in list format, adds corresponding objects in specified classes, returns a list of newly added IdfObjects. Every list should be named with a valid class name. Underscore-style class name is allowed. Names in 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.

**Note**:

- Empty objects can be added using an empty list, e.g. `model$add(building = list())`. All empty fields will be filled with corresponding default value if `.default` is `TRUE`, leaving other fields as blank. However, adding blank objects may not be successful if required fields are not valued and current validate level includes missing-required-field checking. For what kind of validation components to be performed during modifications, please see `level_checks()`.
- Field name matching is **case-insensitive**. For convenience, underscore-style field names are also allowed, e.g. `eNd_MoNtH` is equivalent to `End Month`.
- There is no need to give all field values if only specific fields are interested, unless other fields are not required. For example, to define a new object in `RunPeriod` class, the following is enough: `model$add(RunPeriod = list(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 `model$add(Construction = list("out_layer",name = "name"))`, `"out_layer"` will be treated as the value of field `Outside Layer` in `Construction` class, as value of field `Name` has been given as `"name"`.

**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`, each required field in input object must have one value. Otherwise, an error will be issued during validation. Default: `TRUE`.
- `.all`: If `TRUE`, all fields are added, otherwise only minimum fields are added. Default: `FALSE`.

**Usage**:

- Empty object with default values: `model$add(Building = list(),.default = TRUE)`.
- Empty object with comments: `model$add(Building = list(.comment = c("This","is","a","comment")))`.
- Empty object with all fields: `model$add(Building = list(),.all = TRUE)`.
- New objects: `model$add(RunPeriod = list("rp",1,1,end_month = 2,1,"Monday"),list(Construction = list("const","mat"),Material = list("mat")))`.
- New objects with comments: `model$add(RunPeriod = list("rp",1,1,2,1,.comment = "comment1"))`.
- Variable inputs: `x <-list(Construction = list("const"),Building = list()); model$add(x)`.

### Set Values of Existing Objects:

```
model$set(..., .default = TRUE)
```

`$set()` takes new field value definitions in list format, sets new values for fields in objects specified, and returns a list of modified IdfObjects. Every list in `$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.

**Note**:

- You can delete a field by assigning `NULL` to it, e.g. `list(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.
- 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"`.

**Arguments**:

- `...`: Lists of object definitions. Each list should be named with a valid object name or a valid object ID denoted in style `..1`, `..2` and etc. There is a special element `.comment` in each list, which will be used as new comments of modified object, overwriting existing ones.
- `.default`: If `TRUE`, default values are used for those blank fields if possible. Default: `TRUE`.

**Usage**:

- Specify object with name: `model$set(Object_Name = list(val1,val2,val3))`.
- Specify object with ID: `model$set(..8 = list(val1))`.
- Overwrite existing object comments: `model$set(..8 = list(.comment = c("new","comment")))`.
- Delete field value: `model$set(Object_Name = list(Field_1 = NULL),.default = FALSE)`.
- Assign default field value: `model$set(Object_Name = list(Field_1 = NULL),.default = TRUE)`.
- Variable input: `a <-list(Object_Name = list(Field_1 = val1)); model$set(a,.default = TRUE)`.
- Set all values of field `fld` in a class `cls`:

```
ids <- model$object_id("cls", simplify = TRUE)
val <- rep(list(list(fld = val)), times = length(ids))
names(val) <- paste0("..", ids)
model$set(val)
```

**Deleting Existing Objects:**

```
model$del(..., .ref_by = FALSE, .ref_to = FALSE, .recursive = FALSE, .force = FALSE)
```

`$del()` takes integer vectors of object IDs and character vectors of object names, and deletes objects specified. 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.

**Note**:

- 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 are only deleted when they only have relation with input 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`.

**Arguments**:

- `.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.

**Usage**:

- Specify object with name: `model$del("Object_Name1","Object_Name2")`.
- Specify object with ID: `model$del(1,2,10)`.
- Delete objects even they are referred by other objects: `model$del(1:5,.force = TRUE)`
- Delete objects and also other objects that refer to them: `model$del(2,"Object_Name1",.ref_by = TRUE)`
- Delete objects and also other objects that refer to them recursively: `model$del(1:5,.ref_by = TRUE,.recursive = TRUE)`
- Delete objects and also other objects that input objects refer to: `model$del(1:5,.ref_to = TRUE)`
- Variable input: `x <-c("Object_Name1","Object_Name2"); y <-c(1:5); model$del(x,y)`.

### Rename Objects:

`model$rename(...)`

`$rename()` takes named character vectors of object names and named integer vectors of object IDs, renames specified object to names of input vectors and returns a list of renamed IdfObjects. An error will be issued if trying to "rename" an object which does not have name attribute.

**Argument**:

- `...`: Integer vectors of valid object IDs and character vectors of valid object names. Each element should be named. That name is used at the new object name.

**Usage**:

- Rename objects: `model$rename(c(new_name1 = "name1",new_name2 = "name2"),new_name3 = 6)`.

- Variable inputs: `a <-c(new_name1 = "name1",new_name2 = "name2"); b <-c(new_name3 = 10); model$rename(a,b)`.

**Insert Objects:**

```
model$insert(..., .unique = TRUE)
```

`$insert()` takes IdfObjects or lists of IdfObjects as input, inserts them into current Idf, and returns a list of inserted IdfObjects.

**Note**:

- You cannot insert an IdfObject which comes from a different version than 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 may be issued if current validation level includes conflicted-name checking. For what kind of validation components to be performed during modifications, please see `level_checks()`.

**Argument**:

- `...`: IdfObjects or lists of IdfObjects.
- `.unique`: If there are duplications in input IdfObjects or there is same object in current `Idf` object, duplications in input are removed. Default: `TRUE`.

**Usage**:

- Insert objects without new names: `model1$insert(model2$Material)`.
- Insert an object without new name: `model1$insert(my_material = model2$Material[[1]])`.
- Insert objects but keep duplications: `model1$insert(model1$Output_Variable)`.
- Variable input: `mat <-model2$Material; names(mat) <-c("mat1","mat2"); model1$insert(mat)`.

**Load Objects from characters or data.frames:**

```
model$load(..., .unique = TRUE, .default = TRUE)
```

`$load()` is similar to `$insert()` except it takes directly character vectors or data.frames of IdfObject definitions, insert corresponding objects into current `Idf` object and returns a list of newly added IdfObjects. This makes it easy to create objects using the output from `$to_string()` and `$to_table` method from Idd, IddObject, also Idf and IdfObject class.

For object definitions in character vector format, they follow the same rules as normal IDF file. Each object starts with a class name and a comma (`,`), separates each values with a comma (`,`) and ends with a semicolon (`;`). Noted that you can also provide headers 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, IddObject, `Idf` and IdfObject class. A valid definition requires at least three columns described below. Note that column order does not matter.

- `class`: **Mandatory**. Character type. Valid class names in the underlying Idd object. You can get all valid class names using `use_idd(model$version())$class_name()`
- `index`: **Mandatory**. Integer type. Valid field indexes for each class.
- `value`: **Mandatory**. Character type or list type. The value of each field to be added.
  - If `value` is a character column, usually when `string_value` is `TRUE` in method `$to_table()` in `Idf` and IdfObject class. Each value should be given as a string even if the corresponding field is a numeric type.

  - If `value` is a list column, 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 during if current validation level includes invalid-type checking. For what kind of validation components to be performed during modifications, please see `level_checks()`.

- `id`: **Optional**. Integer type. If input data.frame includes multiple object definitions in a same class, the value in `id` 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**:

- `$load()` assume all definitions are from the same version as current `Idf` object. If input definition is from different version, parsing error may occur.

**Argument**:

- `...`: Character vectors or data.frames of object definitions For details, see above.
- `.unique`: If there are duplications in input IdfObjects or there is same object in current Idf, duplications in input are removed. Default: `TRUE`.
- `.default`: If `TRUE`, default values are used for those blank fields if possible. Default: `TRUE`.

**Usage**:

- Load objects from string definitions:

```
model$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 <- model1$to_table(class = "Material")
dt[field == "thickness", value := "0.5"]
model$load(dt)
```

**Update Objects from characters or data.frames:**

```
model$update(..., .unique = TRUE, .default = TRUE)
```

`$update()` is similar to `$load()` except it update field values. This makes it easy to update object values using the output from `$to_string()` and `$to_table` method from Idd, IddObject, also Idf and IdfObject class.

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 it can be achieved using data.frame format as it uses object IDs instead of object names to locate objects.

For object definitions in data.frame format, it is highly recommended to use `$to_table()` method in Idd, IddObject, `Idf` and IdfObject class. A valid definition requires four columns described below. Note that column order does not matter.

- `id`: Integer type. The value in `id` should be valid object IDs for current `Idf`.
- `class`: Character type. Valid class names for current `Idf`.
- `index`: Integer type. Valid field indexes for each class. If input field does not exist in specified object, it will be added on the fly.
- `value`: Character type or list type. The value of each field to be added.
  - If `value` is a character column, usually when `string_value` is `TRUE` in method `$to_table()` in `Idf` and IdfObject class. Each value should be given as a string even if the corresponding field is a numeric type.
  - If `value` is a list column, usually when `string_value` is set to `FALSE` in method `$to_table()` in `Idf` and IdfObject class. Each value should have the right type as the corresponding field definition. Otherwise, errors will be issued during if current validation level includes invalid-type checking. For what kind of validation components to be performed during modifications, please see `level_checks()`.

**Note**:

- `$update()` assume all definitions are from the same version as current `Idf` object. If input definition is from different version, parsing error may occur.

**Argument**:

- `...`: Character vectors or data.frames of object definitions For details, see above.
- `.default`: If `TRUE`, default values are used for those blank fields if possible. Default: `TRUE`.

**Usage**:

- Update objects from string definitions:

```
str <- model$Material[[1]]$to_string()
str[4] <- "0.8"
model$update(str)
```

- Update objects from data.frame definitions:

```
dt <- model1$to_table(class = "Material")
dt[field == "thickness", value := "0.5"]
model$update(dt)
```

### Paste Objects from IDF Editor:

```
model$paste(in_ip = FALSE, ver = NULL, unique = TRUE)
```

`$paste()` reads the contents (from clipboard) of copied objects from IDF Editor (after hitting `Copy Obj` button), parses it and inserts corresponding objects into current Idf. As IDF Editor only available on Windows platform, `$paste()` only works on Windows too.

**Note**:

- There is no version data copied to the clipboard when copying objects in IDF Editor. It is possible that IDF Editor opens an IDF with different version than current IDF. 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.

**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 opened 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 there are duplications in copied objects from IDF Editor or there is same object in current Idf, duplications in input are removed. Default: `TRUE`.

**Usage**:

- Paste objects from same version: `model$paste()`.
- Paste objects from different version: `model$paste(ver = "version")`.
- Paste objects that are viewed in IP units in IDF Editor: `model$paste(in_ip = TRUE)`.
- Paste objects but also keep duplications: `model$paste(unique = FALSE)`.

**Search and Replace Values:**

```
model$search_value(pattern, class = NULL, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBytes =
model$replace_value(pattern, class = NULL, replacement, ignore.case = FALSE, perl = FALSE, fixed = FAL
```

`$search_value()` returns a list of IdfObjects that contain values which match the given pattern. If no matched found, `NULL` is returned invisibly.

`$replace_value()` returns a list of IdfObjects whose values have been replace with given pattern. If no matched found, `NULL` is returned invisibly.

**Note**:

- During matching, all values are treated as characters, including numeric values.
- Replacing values using regular expression is not recommended, because it is error prone. Validation rules also apply during replacing.

**Arguments**:

- `class`: A character vector of invalid class names in current `Idf` object to search for values. If `NULL`, all classes are used.
- `pattern`, `replacement`, `ignore.case`, `perl`, `fixed` and `useBytes`: All of them are directly passed to base::grepl and base::gsub.

**Usage**:

- Search values that contains `supply`: `model$search_value("supply")`
- Search values that contains `supply` or `demand` in class `Branch`: `model$search_value("supply|demand","Branch")`
- Search values that contains `win` and replace them with `windows`: `model$replace_value("win","windows")`

## Validation

```
model$validate(level = eplusr_option("validate_level"))
model$is_valid(level = eplusr_option("validate_level"))
```

`$validate()` checks if there are errors in current `Idf` under specified validation level and returns an `IdfValidity` object which contains data of invalid field values. Different validation 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 validation checks in specified validation 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 validation results show that after all validation components performed under current validation level, 2 invalid field values are found. All of them are in 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 set those two fields to correct value by doing `idf$set(..176 = list(`Maximum Flow per Zone Floor Area During Reheat`= "autosize",`Maximum Flow Fraction During Reheat = "autosize" ))`

`$is_valid()` returns `TRUE` if there is no error in current `Idf` object under specified validation level and `FALSE` otherwise.

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. In total, There are 10 different validation check components. To get the meaning of each component, please see `level_checks()` and `custom_validate()`.

- `missing_object`
- `duplicate_object`
- `conflict_name`
- `incomplete_extensible`
- `missing_value`
- `invalid_autosize`
- `invalid_autocalculate`
- `invalid_character`
- `invalid_numeric`
- `invalid_integer`
- `invalid_choice`
- `invalid_range`
- `invalid_reference`

Except `missing_object`, which is a character vector, 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.

### Data Extraction

```
model$to_table(which = NULL, class = NULL, string_value = TRUE, unit = FALSE, wide = FALSE)
model$to_string(which = NULL, class = NULL, comment = TRUE, header = TRUE, format = eplusr_option("save_
```

$to_table() returns a data.table that contains core data of specified objects. 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. Field values.

$to_string() returns the text format of an IDF file.

**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. Default: FALSE.
- 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.

### Save

```
model$is_unsaved()
model$save(path = NULL, format = eplusr_option("save_format"), overwrite = FALSE, copy_external = TRUE)
```

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

$save() saves the Idf object as an IDF file.

**Arguments**:

- path: A path where to save the model. If NULL, the path of the Idf itself, i.e. model$path(), will be used.
- format: A string to specify the saving format. Should be one of "asis", "sorted", "new_top", and "new_bot".
  - If "asis", the model will be saved in the same format as it was when first read. If the model 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 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.

### Clone

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

$clone() returns the exactly the same cloned model. Because Idf uses R6::R6Class() under the hook which has "modify-in-place" semantics, idf_2 <-idf_1 does not copy idf_1 at all but only create a new binding to idf_1. Modify idf_1 will also affect idf_2 as well, as these two are exactly the same thing underneath. In order to create a complete cloned copy, use $clone(deep = TRUE).

**Arguments**:

- deep: Has to be TRUE if a complete cloned copy is desired. Default: TRUE.

### Run Model

```
model$run(weather, dir = NULL, wait = TRUE, force = FALSE, copy_external = FALSE)
```

`$run()` calls corresponding version of EnergyPlus to run the current model 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.

**Note**:

- 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.
- eplusr uses the 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.

**Arguments**:

- `weather`: A path to an `.epw` file or an Epw object.
- `dir`: The directory to save the simulation results. If `NULL`, the model folder will be used. Default: NULL
- `wait`: Whether to wait until the simulation completes and print the standard output and error of EnergyPlus to the screen. 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. 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`.

### Print

```
model$print(zoom = c("object", "class", "group", "field"), order = TRUE)
print(model)
```

`$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.

**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.

## Author(s)

Hongyuan Jia

## See Also

IdfObject class

## Examples

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

# ===== MODEL BASIC INFO =====
# get version
idf$version()

# get path
idf$path()

# get names of all groups in current model
str(idf$group_name())

# get names of all defined groups in the IDD
str(idf$group_name(all = TRUE))

# get names of all classes in current model
str(idf$class_name())

# get names of all defined classes in the IDD
str(idf$class_name(all = TRUE))

# check if input is a valid group name in current model
idf$is_valid_group("Schedules")
idf$is_valid_group("Compliance Objects")

# check if input is a valid group name in IDD
idf$is_valid_group("Compliance Objects", all = TRUE)
```

```
# check if input is a valid class name in current model
idf$is_valid_class("Building")
idf$is_valid_class("ShadowCalculation")

# check if input is a valid class name in IDD
idf$is_valid_class("ShadowCalculation", all = TRUE)

# ===== OBJECT DEFINITION (IDDOBJECT) =====
# get the a list of underlying IddObjects
idf$definition("Version")

# ===== OBJECT INFO =====
# get IDs of objects in classes
idf$object_id(c("Version", "Zone"))

# when `simplify` is TRUE, an integer vector will be returned instead of a
# named list
idf$object_id(c("Version", "Zone"), simplify = TRUE)

# get names of objects in classes
# NA will be returned if targeted class does not have a name attribute
idf$object_name(c("Building", "Zone", "Version"))

# if `simplify` is TRUE, a character vector will be returned instead of a
# named list
idf$object_name(c("Building", "Zone", "Version"), simplify = TRUE)

# get number of objects in classes
idf$object_num(c("Zone", "Schedule:Compact"))

# check if input is a valid object ID, i.e. there is an object whose ID is
# the same with input integer
idf$is_valid_id(c(51, 1000))

# check if input is a valid object name, i.e., there is an object whose name is
# the same with input string
idf$is_valid_name(c("Simple One Zone (Wireframe DXF)", "ZONE ONE"))

# ===== OBJECT QUERY =====
# get single object using object ID or name
# NOTE: object name matching is case-insensitive
idf$object(3)
idf$object("simple one zone (wireframe dxf)")

# get objects using object IDs or names
# NOTE: object name matching is case-insensitive
idf$objects(c(3,10))
idf$objects(c("Simple One Zone (Wireframe DXF)", "zone one"))

# the names of returned list are object names
names(idf$objects(c("Simple One Zone (Wireframe DXF)", "zone one")))

# get all objects in classes in a named list
```

```
idf$objects_in_class("Zone")
names(idf$objects_in_class("Zone"))

# OR using shortcuts
idf$Zone
idf[["Zone"]]

# get a single object in unique-object class
idf$object_unique("SimulationControl")
idf$SimulationControl
idf[["SimulationControl"]]

# search objects using regular expression
length(idf$search_object("R13"))
names(idf$search_object("R13"))

# search objects using regular expression in specifc class
length(idf$search_object("R13", class = "Construction"))
names(idf$search_object("R13", class = "Construction"))

# get more controls on matching
length(idf$search_object("r\\d", ignore.case = TRUE, class = "Construction"))
names(idf$search_object("r\\d", ignore.case = TRUE, class = "Construction"))

# ===== DUPLICATE OBJECTS =====
# duplicate objects in "Construction" class
names(idf$Construction)
idf$dup("R13WALL")

# new objects will have the same names as the duplicated objects but with a
# suffix "_1", "_2" and etc.
names(idf$Construction)

# new names can also be explicitly specified
idf$dup(My_R13Wall = "R13WALL")

# duplicate an object multiple times
idf$dup(rep("R13WALL", times = 10))

# ===== ADD OBJECTS =====
# add two new objects in "RunPeriod" class
idf$add(
    RunPeriod = list("rp_test_1", 1, 1, 2, 1,
        .comment = c("Comment for new object 1", "Another comment")
    ),

    RunPeriod = list(name = "rp_test_2",
        begin_month = 3,
        begin_day_of_month = 1,
        end_month = 4,
        end_day_of_month = 1,
        .comment = "Comment for new object 2"
     )
```

```
)

# ===== LOAD OBJECTS =====
# load objects from character vector
idf$load("RunPeriod, rp_test_3, 1, 1, 2, 1;")

# load objects from data.frames
dt <- idf$definition("RunPeriod")$to_table()

## (a) values can be supplied as characters
dt[1:5, value := c("rp_test_4", "2", "1", "3", "1")]
idf$load(dt)

## (b) values can be supplied as list
dt[1:5, value := list("rp_test_5", 3, 1, 4, 1)]
idf$load(dt)

# ===== INSERT OBJECTS =====
# insert objects from other Idf object
idf_1 <- read_idf(system.file("extdata/1ZoneUncontrolled.idf", package = "eplusr"),
    idd = use_idd(8.8, download = "auto"))

idf_1$object_name("Material")

# rename material name from "C5 - 4 IN HW CONCRETE" to "test", otherwise
# insertion will be aborted as there will be two materials with the same name
# in the idf
idf_1$Material$`C5 - 4 IN HW CONCRETE`$set(name = "test")

# insert the object
idf$insert(idf_1$Material$test)

# check if material named "test" is there
idf$object_name("Material")

# $insert() is useful when importing design days from a ".ddy" file
idf$insert(read_idf("foo.ddy"))

# ===== SET OBJECTS =====
# set the thickness of newly inserted material "test" to 0.2 m
idf$set(test = list(thickness = 0.2))
idf$Material$test$Thickness

# set thermal absorptance of all material to 0.85
val <- rep(list(list(thermal_absorptance = 0.85)), idf$object_num("Material"))
names(val) <- idf$object_name("Material", simplify = TRUE)
idf$set(val)

# check results
lapply(idf$Material, function (mat) mat$Thermal_Absorptance)

# reset thermal absorptance of all material to the default
val <- rep(list(list(thermal_absorptance = NULL)), idf$object_num("Material"))
```

```
names(val) <- idf$object_name("Material", simplify = TRUE)
idf$set(val)

# check results
lapply(idf$Material, function (mat) mat$Thermal_Absorptance)

# ===== UPDATE OBJECTS =====
# update roughness of new added material "test" to "Smooth"
idf$update("Material, test, smooth;")

# update solar absorptance of all materials to 0.8
dt <- idf$to_table(class = "Material", string_value = TRUE)
idf$update(dt[field == "Solar Absorptance", value := "0.8"])

# ===== RENAME OBJECTS =====
idf$rename(new_test = "test")
idf$object_name("Material")

# ===== DELELTE OBJECTS =====
# delete the added run period "rp_test_1", "rp_test_2" and "new_test" from above
idf$del(c("new_test", "rp_test_1", "rp_test_2"))
names(idf$Material)
names(idf$RunPeriod)

# In "final" validate level, delete will be aborted if the target objects are
# referenced by other objects.
# get objects that referenced material "R13LAYER"
eplusr_option("validate_level")

idf$Material_NoMass$R13LAYER$ref_by_object()
length(idf$Material_NoMass$R13LAYER$ref_by_object())

idf$del("R13LAYER") # will give an error in "final" validate level

# objects referencing target objects can also be deleted by setting
# `referenced` to TRUE
idf$del("R13LAYER", .ref_by = TRUE) # will give an error in "final" validate level

# it is possible to force delete objects
idf$del("R13LAYER", .ref_by = TRUE, .force = TRUE)

# ===== SEARCH ADN REPLACE OBJECT VALUES =====
# get objects whose field values contains both "VAV" and "Node"
idf$search_value("WALL")
length(idf$search_value("WALL"))
names(idf$search_value("WALL"))

# replace values using regular expression
idf$replace_value("WALL", "A_WALL")

# ===== VALIDATE MODEL =====
# check if there are errors in current model
idf$validate()
```

```
idf$is_valid()

# change validate level to "none", which will enable invalid modifications
eplusr_option(validate_level = "none")

# change the outside layer of floor to an invalid material
idf$set(FLOOR = list(outside_layer = "wrong_layer"))

# change validate level back to "final" and validate the model again
eplusr_option(validate_level = "final")

idf$validate()
idf$is_valid()

# get IDs of all objects that contains invalid reference fields
idf$validate()$invalid_reference$object_id

# fix the error
idf$set(..16 = list(outside_layer = idf$Material[[1]]$name()))
idf$validate()
idf$is_valid()

# ===== FORMAT MODEL =====
# get text format of the model
head(idf$to_string())

# get text format of the model, excluding the header and all comments
head(idf$to_string(comment = FALSE, header = FALSE))

# ===== SAVE MODEL =====
# check if the model has been modified since read or last saved
idf$is_unsaved()

# save and overwrite current model
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
idf$save(path = file.path(tempdir(), "test.idf"))

# 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)

# the path of this model will be changed to the saved path
idf$path()

# ===== CLONE MODEL =====
# Idf object are modified in place and has reference semantic.
idf_2 <- idf
idf_2$object_name("Building")
```

```
idf$object_name("Building")

# modify idf_2 will also affect idf as well
idf_2$Building$set(name = "Building_Name_Changed")
idf_2$object_name("Building")
idf$object_name("Building")

# in order to make a copy of an Idf object, use $clone() method
idf_3 <- idf$clone(deep = TRUE)
idf_3$Building$set(name = "Building_Name_Changed_Again")
idf_3$object_name("Building")

idf$object_name("Building")

# ===== RUN MODEL =====
if (is_avail_eplus(8.8)) {

    # 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]
    basename(epw)
    # [1] "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    # 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()
}
```

```
# ===== PRINT MODEL =====
idf$print("group")
idf$print("class")
idf$print("object")
idf$print("field")

## 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 be created using function idf_object() or from methods of a parent Idf object, using $object(), $objects_in_class() and equivalent.

## Usage

```
idfobj <- model$object(which)
idfobj <- idf_object(model, which, class = NULL)
idfobj$version()
idfobj$id()
idfobj$name()
idfobj$definition()
idfobj$comment(comment, append = TRUE, width = 0L)
idfobj$value(which = NULL, all = FALSE, simplify = FALSE, unit = FALSE)
idfobj$value_possible(which = NULL, type = c("auto", "default", "choice", "range", "source"))
idfobj$FieldName
idfobj[[Field]]
idfobj$set(..., .defaults = TRUE)
idfobj$FieldName <- Value
idfobj[[Field]] <- Value
idfobj$value_relation(which = NULL, direction = c("all", "ref_to", "ref_by", "node"), recursive = FALSE
idfobj$ref_to_object(which = NULL, class = NULL, recursive = FALSE, depth = 1L)
idfobj$ref_by_object(which = NULL, class = NULL, recursive = FALSE, depth = 1L)
idfobj$ref_to_node(which = NULL, class = NULL, recursive = FALSE, depth = 1L)
idfobj$has_ref_to(which = NULL, class = NULL)
idfobj$has_ref_by(which = NULL, class = NULL)
idfobj$has_ref_node(which = NULL, class = NULL)
idfobj$has_ref(which)
idfobj$validate(level = eplusr_option("validate_level"))
idfobj$is_valid(level = eplusr_option("validate_level"))
idfobj$to_table(all = FALSE, unit = TRUE, wide = FALSE, string_value = TRUE)
idfobj$to_string(comment = TRUE, leading = 4L, sep_at = 29L, all = FALSE)
idfobj$print(comment = TRUE, auto_sep = FALSE, brief = FALSE)
print(iddobj)
```

### Basic Info

```
idfobj <- model$object(which)
idfobj <- idf_object(model, which, class = NULL)
idfobj$version()
idfobj$id()
idfobj$name()
idfobj$group_name()
idfobj$class_name()
```

`$version()` returns the version of parent Idf object in a numeric_version format.

`$id()` returns current `IdfObject` ID.

`$name()` returns current `IdfObject` name. If the class does not have name attribute, `NA` is returned.

`$group_name()` returns the group name of current IdfObject belongs to.

`$class_name()` returns the class name of current IdfObject belongs to.

**Arguments**:

- `model`: An Idf object.
- `class`: A single string of valid class name.
- `which`: A valid object ID (an integer) or name (a string).

### Definition

```
idfobj$definition()
```

`$definition()` returns an IddObject of current class. IddObject contains all data used for parsing and creating an IdfObject. For details, please see IddObject class.

### Getting and Setting Comments

```
idfobj$comment(comment, append = TRUE, width = 0L)
```

`$comment()` returns current `IdfObject` comment if `comment` is not given, or modifies current `IdfObject` comment if `comment` is given.

**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`.

### Get Field Values

```
idfobj$value(which = NULL, all = FALSE, simplify = FALSE, unit = FALSE)
idfobj$value_possible(which = NULL, type = c("auto", "default", "choice", "range", "source"))
idfobj$FieldName
idfobj[[Field]]
```

`$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]]`.

`$possible_value()` 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()` and `idfobj$definition()$is_autosizable`
- `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.

`$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.

**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.
- FieldName: A single length character vector of one valid field name where all characters except letters and numbers are replaced by underscores.
- Field: A single length character vector of one valid field name or a single length integer vector of one valid field index. Same as above, field names should be given in a style where all characters except letters and numbers are replaced by underscores.
- 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.

### Set Field Values

```
idfobj$set(..., .default = TRUE)
idfobj$FieldName <- Value
idfobj[[Field]] <- Value
```

$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().

**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.
- 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)`.

**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`.
- `FieldName`: A single length character vector of one valid field name where all characters except letters and numbers are replaced by underscores.
- `Field`: A single length character vector of one valid field name or a single length integer vector of one valid field index. Same as above, field names should be given in a style where all characters except letters and numbers are replaced by underscores.
- `Value`: A single length vector of value to set.

### Field Value Relation

```
idfobj$value_relation(which = NULL, direction = c("all", "ref_to", "ref_by", "node"), recursive = FALSE
idfobj$ref_to_object(which = NULL, class = NULL, recursive = FALSE, depth = 1L)
idfobj$ref_by_object(which = NULL, class = NULL, recursive = FALSE, depth = 1L)
idfobj$ref_to_node(which = NULL, class = NULL, recursive = FALSE, depth = 1L)
idfobj$has_ref_to(which = NULL, class = NULL)
idfobj$has_ref_by(which = NULL, class = NULL)
idfobj$has_ref_node(which = NULL, class = NULL)
idfobj$has_ref(which)
```

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 futher easily extracted using $ref_by_object() method.

Note that $value_relation() shows all fields specified, even some of them may do not have 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.

$ref_by_object() takes an integer vector of field indexes or a character vector of field names, and returns a list of IdfObjects that refer to specified fields.

$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.

$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.

$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.

$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.

$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.

**Arguments**:

- which: An integer vector of field indexes or a character vector of field names.
- class: A character vector of class names.
- direciton: The relation direction to extract. Should be either "all", "ref_to" or "ref_by".
- 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.
- depth: Only applicable when recursive is TRUE. This is a depth to when searching value relations recursively. If NULL, all recursive relations are returned. Default: 1.

### Validation

```
idfobj$validate(level = eplusr_option("validate_level"))
idfobj$is_valid(level = eplusr_option("validate_level"))
```

$validate() checks if there are errors in values in current IdfObject under specified validation level and returns an IdfValidity object which contains data of invalid field values. Different validation 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 validation checks in specified validation 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 validation results show that after all validation components performed under current validation level, 2 invalid field values are found. All of them are in 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 set those two fields to correct value by doing `idf$set(..176 = list(Maximum Flow per Zone Floor Area During Reheat= "autosize",Maximum Flow Fraction During Reheat = "autosize" ))`

`$is_valid()` returns `TRUE` if there is no error in current `IdfObject` object under specified validation level and `FALSE` otherwise.

Underneath, an `IdfValidity` object which `$validate()` returns is a list of 13 element. For details about the underlying structure of `IdfValidity`, please `$validate()` in Idf class.

### Data Extraction

```
idfobj$to_table(string_value = TRUE, unit = TRUE, wide = FALSE, all = FALSE)
idfobj$to_string(comment = TRUE, leading = 4L, sep_at = 29L, all = FALSE)
```

`$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`. Field values.

`$to_string()` returns the text format of an `IdfObject`.

**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`: 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. Default: `FALSE`.
- `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`

### Print

```
idfobj$print(comment = TRUE, auto_sep = FALSE)
print(idfobj)
```

`$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>`.

#### 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`.

### Author(s)

Hongyuan Jia

### See Also

Idf class

### Examples

```
## Not run:
# read an IDF file
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"]]

# get object ID
```

```
mat$id()

# get object name
mat$name()

# NA will be returned if the class does not have name attribute. For example,
# "Version" class
idf$Version$name()

# get underlying IddObject of current class
mat$definition()

# get object comments
mat$comment()

# add new object comments
mat$comment(c("This is a material named `WD01`", "This object has an ID of 47"))
mat$comment()

# append new comments
mat$comment("This is an appended comment")
mat$comment()

# prepend new comments
mat$comment("This is a prepended comment", append = FALSE)
mat$comment()

# wrap long comments
mat$comment("This is a very long comment that is needed to be wrapped.", width = 30)
mat$comment()

# delete old comments and add new one
mat$comment("This is the only comment", append = NULL)
mat$comment()

# delete all comments
mat$comment(NULL)
mat$comment()

# 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(idf$Zone$`ZONE ONE`$value())

str(idf$Zone$`ZONE ONE`$value(all = TRUE))
```

```
# get field values using shortcuts
mat$Roughness
mat[["Specific_Heat"]]
mat[c(1,2)]
mat[c("Name", "Density")]

# 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

# check validate
mat$validate()
mat$is_valid()

# if we set density to a negative number
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()
# get other objects that this object refereces
mat$ref_to_object() # not referencing other objects
mat$has_ref_to()

# get other objects that reference this object
mat$ref_by_object() # referenced by construction "FLOOR"
names(mat$ref_by_object())

mat$has_ref_by()

# check if having any referenced objects or is referenced by other objects
mat$has_ref()

# 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))

# 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)

# 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)
```

---

`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", 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. |
| force | Whether to install EnergyPlus even if it has already been installed. |
| dir | Where to save EnergyPlus installer file. For install_eplus(), the installer will be saved into tempdir() |

## 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 the installation process requires administrative privileges during the installation 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.

## 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:

# for the latest version of EnergyPlus
download_eplus("latest", dir = tempdir())
install_eplus("latest")

# for a specific version of EnergyPlus
download_eplus(8.8, dir = tempdir())
install_eplus(8.8)

## 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 nine element, e.g. `required_object`, `unique_object`, `unique_name`, `extensible`, `required_field`, `autofield`, `type`, `choice`, `range` and `reference`. For the meaning of each validation component, see`custom_validate()`.

## Examples

```
level_checks("draft")
level_checks("final")
level_checks(custom_validate(autofield = 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.

## 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.

## Usage

```
param <- param_job(idf, epw)
param$version()
param$seed()
param$weater()
param$apply_measure(measure, ..., .names = NULL, .mix = FALSE)
param$run(dir = NULL, wait = TRUE, force = FALSE, copy_external = FALSE)
param$kill()
param$status()
param$errors(info = FALSE)
param$output_dir(which = NULL)
param$locate_output(which = NULL, suffix = ".err", strict = TRUE)
param$report_data_dict(which = NULL)
param$report_data(which = NULL, key_value = NULL, name = NULL, year = NULL, tz = "UTC", case = "auto", al
                  period = NULL, month = NULL, day = NULL, hour = NULL, minute = NULL,
                interval = NULL, simulation_days = NULL, day_type = NULL, environment_name = NULL)
param$tabular_data(which, report_name = NULL, report_for = NULL, table_name = NULL, column_name = NULL,
param$print()
```

### Create

```
param <- param_job(idf, epw)
```

#### Arguments

- `idf`: Path to EnergyPlus IDF file or an `Idf` object.
- `epw`: Path to EnergyPlus EPW file or an `Epw` object.

### Get Seed Model and Weather

```
param$version()
param$seed()
param$weather()
```

`$version()` returns the version of input Idf object.

`$seed()` returns the input Idf object.

`$weather()` returns the input Epw object.

### Apply Design Alternatives

```
param$apply_measure(measure, ..., .names = NULL)
```

`$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 arguments as input, and returns a modified Idf object as output. Use `...` to supply different arguments to that measure. Under the hook, `base::mapply()` is used to create multiple Idfs according to the input values.

#### Arguments

- `measure`: A function that takes an `Idf` and other arguments as input and returns an `Idf` object as output.
- `...`: Other arguments 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`.

### Run and Collect Results

```
param$run(dir = NULL, wait = TRUE, force = FALSE, copy_external = FALSE)
param$kill()
param$status()
param$errors(info = FALSE)
param$output_dir(which = NULL)
param$locate_output(which = NULL, suffix = ".err", strict = TRUE)
param$report_data_dict(which = NULL)
param$report_data(which = NULL, key_value = NULL, name = NULL, year = NULL, tz = "UTC", case = "auto", al
                  period = NULL, month = NULL, day = NULL, hour = NULL, minute = NULL,
                interval = NULL, simulation_days = NULL, day_type = NULL, environment_name = NULL)
param$tabular_data(which, report_name = NULL, report_for = NULL, table_name = NULL, column_name = NULL,
```

All those functions have the same meaning as in EplusJob class, except that they only return the results of specified simulations. Most arguments have the same meanings as in EplusJob class.

`$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.

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

`$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 last simulation ended successfully. `FALSE` otherwise. `NA` if the job has not been run yet.
- `changed_after`: `TRUE` if the *seed model* has been modified since last simulation. `FALSE` otherwise.

$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.

`$output_dir()` returns the output directory of specified simulations.

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

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

`$report_data()` extracts the report data in a data.table using key values, variable names and other arguments.

`$tabular_data()` extracts tabular data in a data.table.

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

For `$report_data_dict()`, `$report_data()` and `$tabular_data()`, the returned data.table has a `case` column in the returned data.table that indicates the names of parametric models. For detailed documentation of those methods, please see EplusSql.

**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`.
- `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 every 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.
- suffix: A string that indicates the file extension of simulation output. Default: ".err".
- strict: If TRUE, it checks if the simulation was terminated, is still running or the file does not exist. Default: TRUE.
- info: If FALSE,only warnings and errors are printed. Default: FALSE.
- 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.
- 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. All possible environment_name for current simulation output can be obtained using $read_table("EnvironmentPeriods"). If NULL, all environment data 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.

### Printing

```
param$print()
print(param)
```

`$print()` shows the core information of this `ParametricJob`, including the path of seed model and weather, the version and path of EnergyPlus used to run simulations, the measure that has been applied and parametric models generated, 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.

### Author(s)

Hongyuan Jia

### Examples

```
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
    param_job(idf_path, epw_path)

    # create from an Idf and an Epw object
    param_job(read_idf(idf_path), read_epw(epw_path))
}
```

---

## param_job *Create An EnergyPlus Parametric Simulation Job*

---

### Description

`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. |

## Value

A ParametricJob object.

## Author(s)

Hongyuan Jia

## See Also

eplus_job() for creating an EnergyPlus single simulation job.

## Examples

```
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
    param_job(idf_path, epw_path)

    # create from an Idf and an Epw object
    param_job(read_idf(idf_path), read_epw(epw_path))
}
```

---

# 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.

---

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, warning = FALSE)
```

## Arguments

| | |
|---|---|
| path | A path of an EnergyPlus EPW file. |
| warning | If TRUE, warnings are given if any missing data, out of range data and redundant data is found. Default: FALSE. All these data can be also retrieved using methods in Epw class. |

## Value

An Epw object.

## Author(s)

Hongyuan Jia

## See Also

Epw class

## Examples

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

## 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)

## 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 additional 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.
- `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)

# 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 3 additional 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)

# read the err file
read_rdd(job$locate_output(".rdd"))
read_mdd(job$locate_output(".mdd"))

## End(Not run)
```

run_idf *Run simulations of EnergyPlus models.*

## Description

`run_idf()` is a wrapper of EnergyPlus command line interface which enables to run EnergyPlus model with different options.

## 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, 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. Default: `FALSE`. |
| `annual` | Force design-day-only simulation. 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 processx::process object is returned. Note that, if `FALSE`, R is *not blocked* even when `echo` is `TRUE`. Default: `TRUE`. |
| `echo` | 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_multi()` provides the functionality of running multiple models in parallel.

later package is used to poll the standard output and error of background EnergyPlus process or background R process that handles parallel simulations. The print interval is set to 0.5 sec.

For `run_idf()`, a named list will be returned:

- `idf`: The path of IDF file
- `epw`: The path of EPW file
- `exit_status`: The exit code of the process if it has finished and NULL otherwise. Always being `NULL` if `wait` is FALSE, but you can manually get the exit code using the process object, i.e. `process$get_exit_status()` after simulation *completed*.
- `start_time`: When the EnergyPlus process started.
- `end_time`: When the EnergyPlus process stopped. All being `NULL` if `wait` is `FALSE`, but you can manually check EnergyPlus `stdout` to get the simulation time
- `output_dir`: The simulation output directory
- `energyplus`: The path of EnergyPlus executable
- `stdout`: All standard output from EnergyPlus. Always being `NULL` if `wait` is `FALSE`, but you can manually get all standard output using `process$read_all_output_lines()`.
- `stderr`: All standard error from EnergyPlus. Always being `NULL` if `wait` is `FALSE`, but you can manually get all standard output using `process$read_all_output_lines()`.
- `process`: A processx::process object of current EnergyPlus simulation

For `run_multi()`, if `wait` is `TRUE`, a data.table contains all data (excluding `process`) with same column names as above, and also another two columns:

- `index`: The index of simulation
- `status`: The status of simulation status. Should be one of below:
  - `"completed"`: the simulation job is completed. This only indicates that the calling of EnergyPlus was successfully and EnergyPlus was not terminated during simulation. Even `"completed"` is TRUE, the job can still end with error. Please check exit_status‘ to determine whether EnergyPlus ran successfully
  - `"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()`.

It is suggested to run simulations using EplusJob class and ParametricJob class, which provide much more detailed controls on the simulation and also methods to extract simulation outputs.

## Value

A list for `run_idf()`. For `rum_multi()`, a data.table if `wait` is `TRUE` or a process if `wait` is `FALSE`.

## 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)
```

---

# 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 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 stored in eplusr Idd cache.

## 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
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)
```

# Index