R for Building Energy Simulation

Adrian Chong

2021-10-13

Contents

Preface

Introduction

This is the book for the module **PF4213–Building Energy Analysis and Simulation** at the Department of the Built Environment, National University of Singapore. This book will introduce you to building energy simulation and how to perform data analytics with energy models using R. You will learn how to get your model into R, simulate it, transform the inputs and outputs, and visualize and explore them. This book is designed to be interactive and for you to **learn by doing**. It is highly recommended that you do not copy and paste all the code in this book but instead type them out. Copying and pasting snippets of code isn't the best way to learn because more often than not you are just reading the code at an abstract level without understanding what it does. In contrast, typing the code forces you to try to understand what the code is doing.

Note that this book is still work in progress and some chapters in the book are currently just placeholders that will be completed over time.

Structure of this book

The book is divided into six parts. The code in each chapter is self-containing. Each chapter will contain all the packages and objects needed to run the code and objects created in previous chapters need not be carried over for them to work.

Part I introduces R's basic syntax, which should be sufficient to get most readers started on the subsequent parts of the book. Part II introduces the basic usage and will show you how to get your EnergyPlus model into R, run the simulation, extract the predefined reports and visuaize them. Part III provide details on extracting and modifying various model inputs/outputs, and provide examples of data exploration that can be applied to the model inputs/outputs.

Part IV shows how to programmatically define energy efficient measures using functions and demonstrates the process of performing parametric simulation and it's analysis. Part V shows example application of advanced techniques such as sensitivity analysis, optimization, and Bayesian calibration to building energy simulation. Part VI introduces workflows and tools for reproducing building energy simulation using R.

Prerequisites

This book assumes some theoretical knowledge of building physics and building energy simulation. Some programming experience is preferred but not necessary. To run the code in this book, you will need to install **EnergyPlus**, **R**, **RStudio**, and a handful of R packages that we will introduce as we go along in this book.

R and RStudio

To download and install the most recent version of R, I would recommend the precompiled binary distribution of the base system and contributed packages that you will find at the top of R's project webpage at https://cloud.r-project.org. R Studio is an Integrated Development Environment (IDE) for R.

R Studio increases your productivity when working with the R programming language by combining different features such as syntax highlighting and auto-completion into a single application. You can download and install RStudio from https://www.rstudio.com/products/rstudio/download.

Once you have downloaded and installed RStudio, you will see a similar window to that shown in image below that comprises of:

- Console window: This is the window where you type in R code, press enter, and the results are returned.

- Workspace window: The *Environment* tab in this window is where all the objects that you have created will show.

- Results window: This window is typically used for visualizing graphs (*Plots* tab) and for getting help by accessing the documentation pages for R (*Help* tab).

R Packages

R has thousands of packages hosted on the **C**omprehensive **R** **A**rchieve **N**etwork (CRAN). These packages contains functions developed and shared by the community. In this book we will be using several R packages. The main packages include the **tidyverse** and **eplusr** package. We will introduce the other packages used in this book as we go along.

To install any R package, open R Studio and type

```
install.packages("<package name>")
```

For example, to install the tidyverse package

```
install.packages("tidyverse")
```

Note that you will not be able to use the R package until you load it into your environment using the `library()` function. To load both the tidyverse package, type

```
library("tidyverse")
library("eplusr")
```

EnergyPlus

EnergyPlus is an open-source whole-building energy simulation engine that is widely used by the research community and industry practitioners. EnergyPlus is also the simulation engine supporting many energy simulation applications [noa, a], and it has been and continues to be supported by the U.S. Department of Energy (DOE) [noa, b].

Different releases of EnergyPlus for different platforms (Windows, Linus, or Mac) can be downloaded from https://energyplus.net/downloads. In this book, we will be using EnergyPlus version 9.4.0.

You can also install EnergyPlus via RStudio using the eplusr package. First, install eplusr

```
install.packages("eplusr")
```

Then load the library onto RStudio

```
library(eplusr)
```

You can then install EnergyPlus version 9.4.0 by typing

```
install_eplus(ver = 9.4)
```

Conventions

In this book, code that you can type and run directly in R will appear within grey boxes `like this` or in the form of code blocks like this:

```
1 + 1
## [1] 2
```

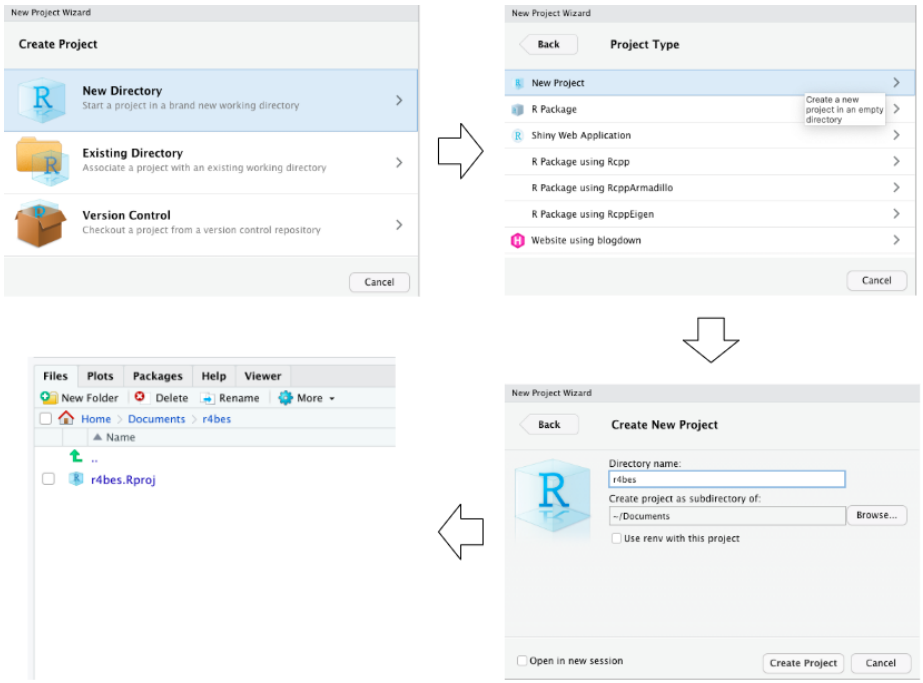
When presenting R code, prompts `>` will not be added and the output from running the code will be commented out with `##` as shown in the example above. This is for your convenience so that it is easy to type or copy the code so that you can run it yourself.

Before you begin

R Project

The Figure below shows the steps to create a project called `r4bes` that you will be using for this book. To create a project, select `File > New Project` from RStudio's menu bar. This launches a window that provides you with options for creating a project. Select the `New Directory` option followed by `New Project`. You can then specify the `Directory name` for your new project and the directory where it will reside. Notice the file with `.Rproj` extension that was created along with the project. This is the project file that you double-click to open a project.

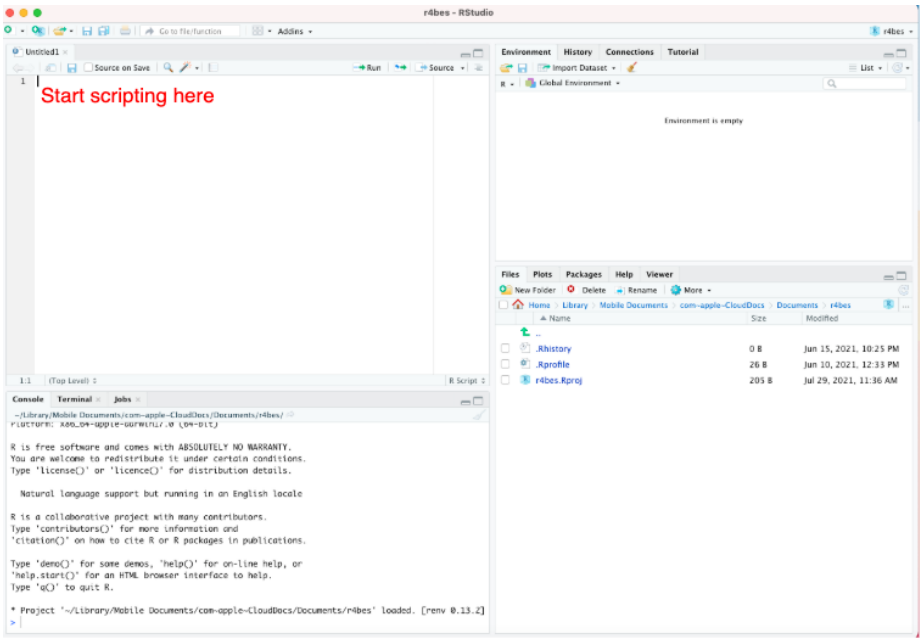
When you open a project, the current working directory is automatically set to the project directory.



Try it now! Quite RStuido and then navigate to and open the `r4bes.Rproj` file.

Scripting

Once the project folder is set up, you can create a new R Script with `File > New File > R Script`. An R Script is basically a text file with a `.R` extension that allows you to keep track and save your `R` code.



Example files

To ensure the reproducibility of the examples in this book, you will download the data folder from this link and place it in the project folder you have just created.

Your project folder should now have the following file structure.

```
./r4bes/
  +-- r4bes.Rproj
  +-- data/
      +-- building_meter.csv
      +-- building.csv
      +-- epw
          +-- USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw
      +-- idf
          +-- RefBldgMediumOfficeNew2004_Chicago.idf
      +-- iris.csv
```

`building_meter.csv` is an example energy meter output file from the the U.S. DOE medium office energy model (i.e., `RefBldgMediumOfficeNew2004_Chicago.idf`). The dataset has a temporal resolution of 1 hour and comprises of 10 variables (the Date/Time and 9 other energy meters) and 8760 observations.

`building.csv` is a timeseries dataset from an actual building consisting of 1324 observations and 2 variables (date/time and building electricity consumption). The dataset has a temporal resolution of 30 minutes.

`USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw` is the latest Typical Meteorological Year weather dataset for the Chicago Ohare International Airport.

`RefBldgMediumOfficeNew2004_Chicago.idf` is a building energy model developed by the U.S. Department of Energy (DOE) to act as a Commercial Reference Building for medium offices [Deru et al., 2011]. We use the DOE reference building because it is well established and widely used. More importantly, it is publicly available and comes distributed with EnergyPlus as example files, making the examples and code in this book easier to follow and reproduce.

`iris.csv` is a popular machine learning dataset about the Iris flower. The dataset consists of 50 observations and 5 variables.

Once you have downloaded the `data` folder into your R project directure, the following code should all return `TRUE`.

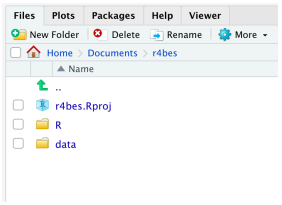
```
library("here")
## here() starts at /Users/adrianchong/Documents/GitHub/r4bes

file.exists(here("data", "iris.csv"))
## [1] TRUE
file.exists(here("data", "building.csv"))
## [1] TRUE
file.exists(here("data", "building_meter.csv"))
## [1] TRUE
file.exists(here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf"))
## [1] TRUE
file.exists(here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw"))
## [1] TRUE
```

Project structure

You should also create a new folder called `R` in the newly created R project. This folder will be used to house all `R` code that you will write as you go through this book. You can create a new folder in RStudio by navigating to the RStudio's results window, select the `Files` tab and click on `New Folder` and enter `R`.

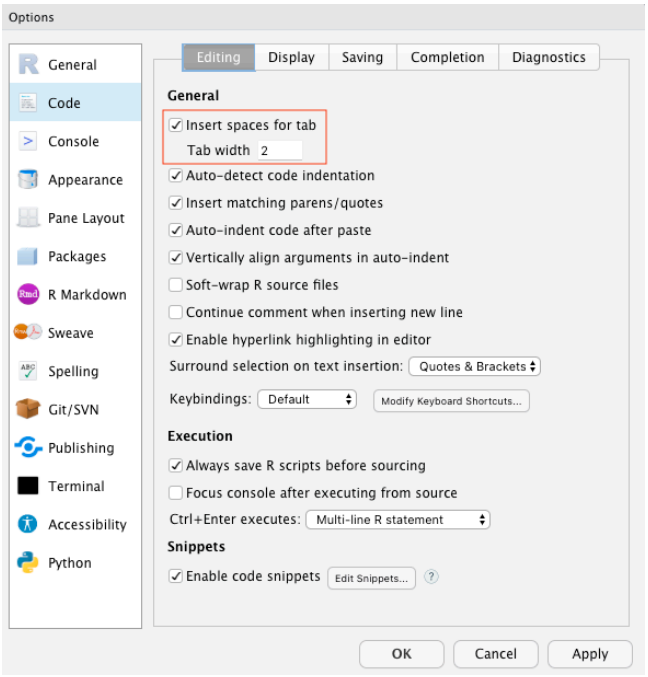
Your project folder should now look like this.



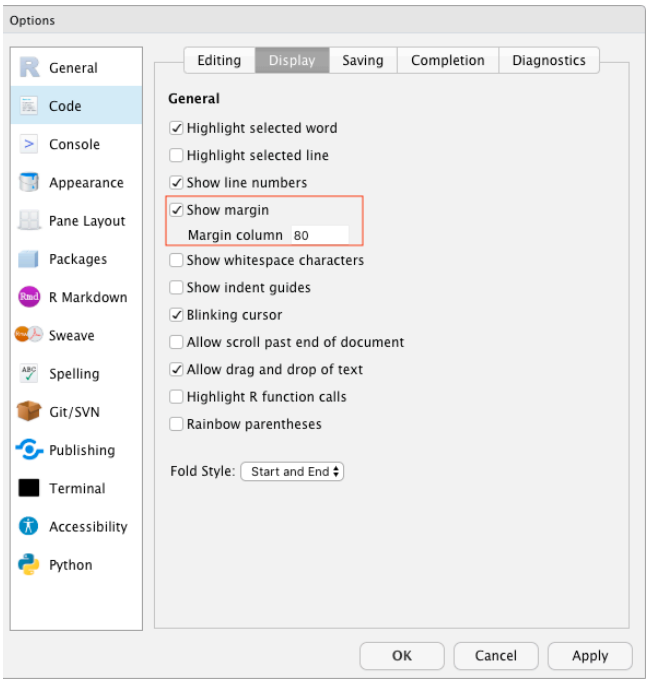
Style guide

We recommend using the Tidyverse Style Guide by Hadley Wickham. To summarize we recommend the following

- R File names should be meaningful and end in `.R`. Avoid using special characters when naming files and use only `lowercase`, hyphens `-`, and underscores `_`.
- `snake_case` (only lowercase letters, numbers, and `_`) for all object and function names. Function names should also reflect what it does.
- Use two white spaces when indenting your code



- Line length should not exceed 80 characters


Options
General
Code
Console
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Terminal
Accessibility
Python
Editing
Display
Saving
Completion
Diagnostics
General
Highlight selected word
Highlight selected line
Show line numbers
Show margin
Margin column 80
Show whitespace characters
Show indent guides
Blinking cursor
Allow scroll past end of document
Allow drag and drop of text
Highlight R function calls
Rainbow parentheses
Fold Style: Start and End
OK
Cancel
Apply

Part I

R Basics

Chapter 1

Introduction

This part of the book is intended to provide a sufficient introduction to R to get you started on the subsequent parts of this book. For a more comprehensive treatment on the subject, I would recommend R for Data Science by Hadley Wickham & Garrett Grolemund [Wickham and Grolemund, 2016]. The R Graphics Cookbook by Winston Chang [Chang, 2018] is also a useful reference that provides recipes that allows you to quickly generate plots in R using the ggplot2 package.

Chapter 2

Basics

2.1 Prerequisites

We introduce the basics in R programming in this chapter. We will review the basic operators and data types in this chapter. We also provide an introduction to the basic R data structures (including tibbles and data tables).

Most of this chapter involves working with R basic operators and data types, which do not require any extra packages. We will also introduce the **tibble** package, which forms part of the tidyverse package in section 2.5, and the **data.table** package in section 2.6

```
library(tidyverse)
library(data.table)
```

2.2 Basic operators

Basic arithmetic operators (`+`, `-`, `*`, `/`, `^`, '%') would work like your calculator

```
3 + 2 # addition
## [1] 5
3 - 2 # subtraction
## [1] 1
3 * 2 # multiplication
## [1] 6
3 / 2 # division
## [1] 1.5
3^2 # exponent
## [1] 9
3 %/% 2 # integer division
## [1] 1
```

```
3 %% 2 # mod (remainder of a division)
## [1] 1
```

R uses the `<-` operator for assignments. You can read the following code as assigning the outcome of `2 + 3`, `5`, and `3` to the object `value_a`,`value_b`, and `value_c` respectively, which stores it for later use.

```
value_a <- 2 + 3
value_b <- 5
value_c <- 3
```

You can `print` what is stored in the object to console with

```
print(value_a)
## [1] 5
```

You can use relational operators to compare how one object relates to another.

```
2 + 3 == 5 # TRUE that 2 + 3 equals 5
## [1] TRUE
2 + 3 != 5 # FALSE that 2 + 3 not equals to 5
## [1] FALSE
2 + 3 < 3 # FASLE that 2 + 3 is less than 3
## [1] FALSE
2 + 3 > 3 # TRUE that 2 + 3 is more than 3
## [1] TRUE
2 + 3 <= 5 # TRUE that 2 + 3 is less than or equal to 5
## [1] TRUE
2 + 3 >= 5 # TRUE that 2 + 3 is more than or equal to 5
## [1] TRUE
```

You can use logical operators to connect two or more expressions. For example, to connect the results of the comparisons made using relational operators.

```
(2 + 3 == 5) && (2 + 3 < 3) # logical AND operator
## [1] FALSE
(2 + 3 == 5) || (2 + 3 >= 3) # logical OR operator
## [1] TRUE
```

Note that the logical `&&` and `||` only examines the first element of a vector.

```
x <- c(TRUE, TRUE, FALSE)
y <- c(FALSE, TRUE, FALSE)
x && y
## [1] FALSE
x || y
## [1] TRUE
```

To perform element-wise logical operations, use `&` and `|` instead

```
x & y
## [1] FALSE  TRUE FALSE
x | y
## [1]  TRUE  TRUE FALSE
!y
## [1]  TRUE FALSE  TRUE
```

2.3 Basic data types

There are basic data types (also known as atomic data types) in R in order to use them.

Data Type	Examples	Additional Information
Logical	`TRUE`, `FALSE`	Boolean values
Numeric	`1`, `999.9`	Default data type for numbers
Integer	`1L`, `999L`	`L` is used to denote an integer
Character	`"a"`, `"R for BES"`	Data type for one or more characters
Complex	`2 + 3i`	Data type for numbers with a real and imaginary component
Raw	`charToRaw("R for BES")`	Not commonly used data type used to store raw bytes

2.4 Basic data structures

The basic data structures in R include factors, atomic vectors, lists, matrices, and `data.frames`

Factors are used in R to represent categorical variables. Although they appear similar to character vectors they are actually stored as integers. You can use the function `levels()` to output the categorical variables and `nlevels()` to check the number of categorical variables.

```
eye_color <- factor(c("brown", "black", "green", "brown", "black", "blue"))
nlevels(eye_color)
## [1] 4
levels(eye_color)
## [1] "black" "blue"  "brown" "green"
```

Atomic vectors or more frequently referred to as vectors are a data structure that is used to store multiple objects of the same data type (logical, numeric, integer, character, complex, or raw). Vectors are one-indexed (i.e., the first element is indexed using `[1]`) and you can get the number of elements in the vector using the function `length()`. The function `class()` can be used to reveal the class of any object in R.

```r
vec_num <- c(1, 2, 3, 4)
class(vec_num)
## [1] "numeric"

vec_char <- c("R", "for", "BES")
class(vec_char)
## [1] "character"

# coercion if data types are mixed
vec_mix <- c("R", 4, "BES")
class(vec_mix)
## [1] "character"

# you can easily combine vectors using the c function
c(vec_char, vec_mix)
## [1] "R"   "for" "BES" "R"   "4"   "BES"

# vector length
length(vec_num)
## [1] 4

# access first element of vector
vec_num[1]
## [1] 1
```

Lists are an ordered data structure that is used to store multiple R objects of different types. The function `list()` is used to create a list and a list in R can be accessed using a single `[]` or double brackets `[[]]`. Using `[]` returns a list of the selected element while using `[[]]` returns the selected element. Using the function `length()`, you can obtain the number of objects in a list.

```r
my_list <- list(
  c(1, 2, 3, 4),
  c("a", "b"),
  1L,
  matrix(1:9, ncol = 3)
)

my_list_a <- my_list[1]
class(my_list_a)
## [1] "list"

my_list_b <- my_list[[1]]
class(my_list_b)
## [1] "numeric"
```

```
length(my_list)
## [1] 4
```

If you have named the elements in your list, you could also access them by specifying their names in the brackets or using the `$` operator.

```
named_list <- list(
  a = c(1, 2, 3, 4),
  b = c("a", "b"),
  c = 1L,
  d = matrix(1:9, ncol = 3)
)

class(named_list["a"])
## [1] "list"

class(named_list[["a"]])
## [1] "numeric"

class(named_list$a)
## [1] "numeric"
```

A matrix is a two dimensional data structure that is used to store multiple objects. You can use the function `matrix()` to create a matrix using the `ncol` and `nrow` argument to specify the number of columns and rows respectively, and the `byrow` argument to specify how the data in would be ordered.

```
matrix(1:12, ncol = 3, byrow = FALSE)
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12

matrix(1:12, nrow = 3, byrow = FALSE)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

matrix(1:12, ncol = 3, byrow = TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

Aside from numeric data types, a matrix can also be used to store other data types as long as they are homogeneous. To store heterogeneous data types, you should use a data frame which is introduced next.

```
matrix(c("brown", "black", "green", "brown", "black", "blue"), ncol = 2)
##      [,1]    [,2]
## [1,] "brown" "brown"
## [2,] "black" "black"
## [3,] "green" "blue"

matrix(c("TRUE", "TRUE", "FALSE", "FALSE"), ncol = 2)
##      [,1]   [,2]
## [1,] "TRUE" "FALSE"
## [2,] "TRUE" "FALSE"

matrix(c(1L, 2L, 3L, 4L), ncol = 2)
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

You can access the values in a matrix by providing the row and column index. For example, `[1, 2]` would return the value stored in the first row, second column of the matrix. `[3, 1]` would return the value stored in the third row, first of the matrix. You can retrieve all the values in a column by leaving the row index empty and likewise retrieve all the values in a row by leaving the column index empty. For example, `[, 2]` would return all the values in column 2 and `[2, ]` would return all the values in row 2.

```
m <- matrix(1:12, nrow = 3, byrow = FALSE)

m[1, 2]
## [1] 4

m[3, 1]
## [1] 3

m[, 2]
## [1] 4 5 6

m[2, ]
## [1]  2  5  8 11
```

A `data.frame` is a two-dimensional data structures that are used to store heterogeneous data types in R. As a result of it's convenience, data frames are a commonly used data structure in R. You can use the function `data.frame()` to create a data frame.

```r
df <- data.frame(
  x = c(1, 2, 3),
  y = c("red", "green", "blue"),
  z = c(TRUE, FALSE, TRUE)
)
```

You can access elements of a data.frame like a list `[]`, `[[]]` or `$`. Using `[]` returns a `data.frame` of the selected element while using `[[]]` or `$` will reduce it to a vector.

```r
df
##   x     y     z
## 1 1   red  TRUE
## 2 2 green FALSE
## 3 3  blue  TRUE

df["y"]
##       y
## 1   red
## 2 green
## 3  blue

df[["y"]]
## [1] "red"   "green" "blue"

df$y
## [1] "red"   "green" "blue"
```

You can also access a `data.frame` like a matrix.

```r
df
##   x     y     z
## 1 1   red  TRUE
## 2 2 green FALSE
## 3 3  blue  TRUE

df[1, 2]
## [1] "red"

df[, 2]
## [1] "red"   "green" "blue"

df[2, ]
##   x     y     z
## 2 2 green FALSE
```

2.5 Tibbles

Tibbles are basically a modified version of R's `data.frame`. Therefore, you would also access tibbles like how you would access a `data.frame`. You can create a tibble using the function `tibble()`. Alternatively, you can coerce a data frame into a tibble using `as_tibble()`.

```
tibble(
  x = c(1, 2, 3),
  y = c("red", "green", "blue"),
  z = c(TRUE, FALSE, TRUE)
)
## # A tibble: 3 x 3
##       x y     z
##   <dbl> <chr> <lgl>
## 1     1 red   TRUE
## 2     2 green FALSE
## 3     3 blue  TRUE

as_tibble(iris)
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
....
```

A key difference lies in how tibbles are printed. Printing a tibble only results in the first ten rows being displayed with an explicit reporting of each column's data type.

```
tb <- as_tibble(iris)
print(tb)
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
```

```
##  7          4.6         3.4          1.4         0.3 setosa
....

df <- as.data.frame(iris)
print(df)
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            4.9         3.0          1.4         0.2     setosa
## 3            4.7         3.2          1.3         0.2     setosa
## 4            4.6         3.1          1.5         0.2     setosa
## 5            5.0         3.6          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 7            4.6         3.4          1.4         0.3     setosa
## 8            5.0         3.4          1.5         0.2     setosa
## 9            4.4         2.9          1.4         0.2     setosa
....
```

Unlike a `data.frame`, tibbles provides clarity on the data structure that it returns. When indexing with tibbles, `[` always return another tibble while `[[` and `$` alway returns a vector. In contrast, single column data frames are often converted into atomic vectors in R unless `drop = FALSE` is specified.

```
class(tb[, 1])
## [1] "tbl_df"     "tbl"        "data.frame"

class(tb[[1]])
## [1] "numeric"

class(tb$Sepal.Length)
## [1] "numeric"

class(df[, 1])
## [1] "numeric"

class(df[, 1, drop = FALSE])
## [1] "data.frame"
```

Additionally, tibbles do not do partial matching and raises a warning unless the variable specified is an exact match.

```
tb$Sepal.Lengt
## Warning: Unknown or uninitialised column: `Sepal.Lengt`.
## NULL

df$Sepal.Lengt
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
```

```
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

You can read more about tibbles by typing vignette("tibble") in your console.

2.6 data.table

Like tibbles, `data.table`s are an enhanced version of `data.frame`s. You can create a `data.table` using the function `data.table()`. You can also coerce existing R objects into a `data.table` with `setDT()` for `data.frame`s and lists, and `as.data.table()` for other data structures. Note that `as.data.table()` also works with `data.frame`s and lists. However `setDT()` is more memory efficient because it does not create a copy of the original data frame or list but instead returns a data table by reference.

```
dt <- data.table(
  x = c(1, 2, 3),
  y = c("red", "green", "blue"),
  z = c(TRUE, FALSE, TRUE)
)

class(
  setDT(
    data.frame(c(1, 2, 3))
  )
)
## [1] "data.table" "data.frame"
```

`data.table`s provide additional functionality through the way it is queried. The general form for working with a data table is `[i, j, by]`, which can be read as subset rows using `i`, operate on `j`, and grouped by `by`.

Lets see how this work using the `iris` example dataset.

```
dt <- as.data.table(iris)

print(dt)
##      Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##   1:          5.1         3.5          1.4         0.2    setosa
##   2:          4.9         3.0          1.4         0.2    setosa
##   3:          4.7         3.2          1.3         0.2    setosa
##   4:          4.6         3.1          1.5         0.2    setosa
```

```
##   5:          5.0         3.6          1.4         0.2    setosa
##  ---
## 146:          6.7         3.0          5.2         2.3 virginica
## 147:          6.3         2.5          5.0         1.9 virginica
## 148:          6.5         3.0          5.2         2.0 virginica
....
```

You can filter the rows to only contain `Species == "virginica`.

```
dt[Species == "virginica"]
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##  1:          6.3         3.3          6.0         2.5 virginica
##  2:          5.8         2.7          5.1         1.9 virginica
##  3:          7.1         3.0          5.9         2.1 virginica
##  4:          6.3         2.9          5.6         1.8 virginica
##  5:          6.5         3.0          5.8         2.2 virginica
##  6:          7.6         3.0          6.6         2.1 virginica
##  7:          4.9         2.5          4.5         1.7 virginica
##  8:          7.3         2.9          6.3         1.8 virginica
##  9:          6.7         2.5          5.8         1.8 virginica
....
```

You can select the columns using the `j` expression. Notice that wrapping the variables within `list()` or `.()` ensures that a `data.table` is returned. In contrast, an atomic vector is returned when `list()` or `.()` is not used. `.()` is an alias for `list()` and therefore the two are the same.

```
dt[, Sepal.Length]
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9

class(dt[, Sepal.Length])
## [1] "numeric"

class(dt[, list(Sepal.Length)])
## [1] "data.table" "data.frame"

class(dt[, .(Sepal.Length)])
## [1] "data.table" "data.frame"
```

You can select multiple columns with `list()` or `.()`.

```
dt[, .(Sepal.Length, Species)]
##      Sepal.Length   Species
##   1:          5.1    setosa
##   2:          4.9    setosa
##   3:          4.7    setosa
##   4:          4.6    setosa
##   5:          5.0    setosa
##  ---
## 146:          6.7 virginica
## 147:          6.3 virginica
## 148:          6.5 virginica
....
```

Aside from selecting columns using `j`, you can carry out computations on `j` involving one or more columns and a subset of rows using `i`.

```
dt[, mean(Sepal.Length)]
## [1] 5.843333

dt[, .(
  Sepal.Length.Mean = mean(Sepal.Length),
  Sepal.With.Mean = mean(Sepal.Width)
)]
##    Sepal.Length.Mean Sepal.With.Mean
## 1:          5.843333        3.057333

dt[
  Species == "virginica" & Sepal.Length < 6,
  .(
    Sepal.Length.Mean = mean(Sepal.Length),
    Sepal.With.Mean = mean(Sepal.Width)
  )
]
##    Sepal.Length.Mean Sepal.With.Mean
## 1:          5.642857        2.714286
```

You can then use the `by` expression in data tables to perform computations by groups.

```
dt[, .(
  Sepal.Length.Mean = mean(Sepal.Length),
  Sepal.With.Mean = mean(Sepal.Width)
),
by = Species
```

```
]
##       Species Sepal.Length.Mean Sepal.With.Mean
## 1:     setosa             5.006           3.428
## 2: versicolor             5.936           2.770
## 3:  virginica             6.588           2.974
```

The `.N` variable that counts the number of instances is particularly useful when combined with `by`.

```
dt[, .N, by = Species]
##       Species  N
## 1:     setosa 50
## 2: versicolor 50
## 3:  virginica 50
```

You can also apply it to multiple columns using the `list()` or `.()` notation. You can read the code below as calculating the mean of `Speal.Length` and the number of instances (given by `.N`) grouped by their `Species` and whether `Sepal.Length < 6`.

```
dt[, .(Sepal.Length.Mean = mean(Sepal.Length), .N),
  by = .(Species, Sepal.Length < 6)
]
##       Species Sepal.Length < 6 Sepal.Length.Mean  N
## 1:     setosa             TRUE          5.006000 50
## 2: versicolor            FALSE          6.375000 24
## 3: versicolor             TRUE          5.530769 26
## 4:  virginica            FALSE          6.741860 43
## 5:  virginica             TRUE          5.642857  7
```

`data.table`s add, update, and delete columns by reference to avoid redundant copies for performance improvements. You can use the `:=` operator to add, update, and delete columns in `j` by reference. There are two forms for using `:=` and they are: `[, LSH := RHS]` and `[,:=(LHS = RHS)]`.

```
dt <- as.data.table(iris)

dt[, Sepal.Sum := .(Sepal.Length + Sepal.Width)]
head(dt)
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Sum
## 1:          5.1         3.5          1.4         0.2  setosa       8.6
## 2:          4.9         3.0          1.4         0.2  setosa       7.9
## 3:          4.7         3.2          1.3         0.2  setosa       7.9
## 4:          4.6         3.1          1.5         0.2  setosa       7.7
## 5:          5.0         3.6          1.4         0.2  setosa       8.6
## 6:          5.4         3.9          1.7         0.4  setosa       9.3
```

```
dt[, `:=`(Petal.Sum = Petal.Length + Petal.Width)]
head(dt)
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Sum
## 1:          5.1         3.5          1.4         0.2  setosa       8.6
## 2:          4.9         3.0          1.4         0.2  setosa       7.9
## 3:          4.7         3.2          1.3         0.2  setosa       7.9
## 4:          4.6         3.1          1.5         0.2  setosa       7.7
## 5:          5.0         3.6          1.4         0.2  setosa       8.6
## 6:          5.4         3.9          1.7         0.4  setosa       9.3
##    Petal.Sum
## 1:       1.6
## 2:       1.6
....
```

Note that in the above code, we do not need to make any assignments back to a variable because the modification is done by reference or in place. In other words we are modifying `dt` and not a copy of `dt`. Therefore, you will also see that if you run the entire code chunk above, `dt` will contain both columns `Sepal.Sum` and `Petal.Sum`.

Since `:=` is used in `j`, it can be combined with `i` and `by` as we have seen in the earlier parts of this sub-section.

```
dt[Species == "versicolor" | Species == "virginica", Sepal.Length := 0]
head(dt)
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Sum
## 1:          5.1         3.5          1.4         0.2  setosa       8.6
## 2:          4.9         3.0          1.4         0.2  setosa       7.9
## 3:          4.7         3.2          1.3         0.2  setosa       7.9
## 4:          4.6         3.1          1.5         0.2  setosa       7.7
## 5:          5.0         3.6          1.4         0.2  setosa       8.6
## 6:          5.4         3.9          1.7         0.4  setosa       9.3
##    Petal.Sum
## 1:       1.6
## 2:       1.6
....

dt[, Sepal.Length.Mean := mean(Sepal.Length),
  by = .(Species)
]
head(dt)
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Sum
## 1:          5.1         3.5          1.4         0.2  setosa       8.6
## 2:          4.9         3.0          1.4         0.2  setosa       7.9
## 3:          4.7         3.2          1.3         0.2  setosa       7.9
## 4:          4.6         3.1          1.5         0.2  setosa       7.7
```

```
## 5:          5.0         3.6          1.4         0.2  setosa      8.6
## 6:          5.4         3.9          1.7         0.4  setosa      9.3
##    Petal.Sum Sepal.Length.Mean
## 1:       1.6              5.006
## 2:       1.6              5.006
....
```

You can find out more about `data.tables` by typing `vignette(package = "data.table")` into the console.

Chapter 3

Dates and Times

3.1 Prerequisites

In this chapter, we will introduce the **lubridate** package that is designed to make it easier to work with dates and times in R.

```
library(lubridate)
```

3.2 Parsing dates and times

You can easily transform dates and times stored as character vectors to Date or POSIXct (or date-time) objects in R using lubridate. All you need to do is specify the order of the year `y`, month `m`, and the day `d` and lubridate will automatically figure out the format.

```
dmy("3/12/1985")
## [1] "1985-12-03"
ymd("85/12/03")
## [1] "1985-12-03"
mdy("Dec 3rd 1985")
## [1] "1985-12-03"
ydm("85-3rd-december")
## [1] "1985-12-03"
```

The same functions can also convert numeric vectors.

```
ymd(19851203)
## [1] "1985-12-03"
dmy(31285)
## [1] "1985-12-03"
```

You can create Date objects with the time component using a POSIXct or date-time object simply by adding an underscore followed by the order of the hour `h`, minute `m` and second `s`.

```
ymd_h("1985/12/03 21")
## [1] "1985-12-03 21:00:00 UTC"
ymd_hm("1985/12/03 21:05")
## [1] "1985-12-03 21:05:00 UTC"
ymd_hms("1985/12/03 21:05:30")
## [1] "1985-12-03 21:05:30 UTC"
```

You can also specify the time zone by providing inputs to the argument `tz`. You can find out more about time zones in R by typing `?timezones` into the console.

```
ymd_hms("1985/12/03 21:05:30", tz = "Singapore")
## [1] "1985-12-03 21:05:30 +08"
```

Date vs date-time objects that are created using lubridate. By default, dates will be created as Date objects without the time component. You can force the creation of a date-time object by including the timezone `tz` argument.

```
class(ymd("1985/12/03"))
## [1] "Date"
class(ymd("1985/12/03", tz = "Singapore"))
## [1] "POSIXct" "POSIXt"
class(ymd_hms("1985/12/03 21:05:30"))
## [1] "POSIXct" "POSIXt"
```

3.3 Extracting components

Lubridate provides simple functions that allows you to easily get different components of a date or date-time object. These functions are especially useful when analyzing time-series data and when you want to group your data by a particular time period.

```
my_dt <- ymd_hms("1985/12/03 21:05:30")
year(my_dt)
## [1] 1985
month(my_dt)
## [1] 12
day(my_dt)
## [1] 3
hour(my_dt)
## [1] 21
minute(my_dt)
## [1] 5
second(my_dt)
```

```
## [1] 30
wday(my_dt, label = TRUE)
## [1] Tue
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Chapter 4

Importing Data

4.1 Prerequisites

In this chapter, we focus on the use of the **readr** package that forms a part of the **tidyverse** package. **readr** provides a fast and easy way to parse a file because it uses a sophisticated parser that guesses the data type for each column along with the flexibility to specify what to parse. You will also need the **here** package (to easily provide relative file paths) and the **lubridate** package (to work with dates and times).

```
library(tidyverse)
library(lubridate)
library(here)
```

In this chapter we will also be using the following datasets. If the following code returns `FALSE`, you should refer to the section Finding your file in the Preface on where you can download the datasets.

```
file.exists(here("data", "iris.csv"))
file.exists(here("data", "building.csv"))
```

4.2 Finding your file

Before importing data, you need to first let R know where to find the file by providing the file path. You can provide file paths relative to the top-level directory of the current R project with the `here()` function.

```
here()
## [1] "/Users/adrianchong/Documents/GitHub/r4bes"
```

To reference a file named `iris.csv` located in the `data` folder that is located in

the top-level directory of your project

```
here("data", "iris.csv")
## [1] "/Users/adrianchong/Documents/GitHub/r4bes/data/iris.csv"
```

4.3 Parsing a csv file

We will focus on the `read_csv()` function because building data are typically stored in `csv` format. You can find out more about other file formats in readr's documentation.

The easiest way to parse a file is by providing the file path. In most cases, it will just work as expected where `readr` correctly guesses the column specification (that gets printed to the console) and you get a tibble as specified. `readr_example()` is a function that makes it easy to access example files to demonstrate `readr`'s capabilities.

```
read_csv(here("data", "iris.csv"))
##
## -- Column specification --------------------------------------------------------
## cols(
##   Sepal.Length = col_double(),
##   Sepal.Width = col_double(),
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
....
```

4.4 Building data

Data from buildings are often messy without a standardized format. Therefore, it is not uncommon if `readr` does not guess correctly. Parsing date/times is particularly important when working with time-series building data.

For example, with the `building.csv` dataset, you can see that there are prob-

lems parsing the `timestamp` column. Specifically, column `timestamp` was parsed as a character vector when it actually contains a date/time. The `skip` argument is particularly useful since building data often contains meta-data that you want to exclude. A character vector can also be supplied to specify the column names.

```
read_csv(
  here("data", "building.csv"),
  col_names = c(
    "datetime",
    "power"
  ),
  skip = 2
)
##
## -- Column specification --------------------------------------------------------
## cols(
##   datetime = col_character(),
##   power = col_number()
## )
## # A tibble: 1,324 x 2
##    datetime       power
##    <chr>          <dbl>
##  1 1/9/19 0:00 2210185.
##  2 1/9/19 0:30 2210196.
##  3 1/9/19 1:00 2210208.
##  4 1/9/19 1:30 2210219
##  5 1/9/19 2:00 2210231
##  6 1/9/19 2:30 2210242.
##  7 1/9/19 3:00 2210254.
....
```

You can correct this by taking advantage of the `lubridate` package that was covered in Chapter 3. To recap, you can transform a character vector into a datetime object by specifying the order of the year `y`, month `m`, day `d`, hour `h`, minute `m` and second `s` with the date (`y`, `m`, and `d`) and the time (`h`, `m`, and `s`) separated by an underscore `_`.

```
bldg <- read_csv(
  here("data", "building.csv"),
  col_names = c(
    "datetime",
    "power"
  ),
  skip = 2
)
##
```

```
## -- Column specification --------------------------------------------------------
## cols(
##   datetime = col_character(),
##   power = col_number()
## )

# transform the datetime character vector into a datetime object using lubridate
bldg$datetime <- dmy_hm(bldg$datetime)
```

Chapter 5

Regular expressions

Regular expressions are patterns that are used to match combinations of characters in a string. Before we begin, just a cautionary note that if your regular expressions are becoming too complex, perhaps it is time to step back and think about whether it is necessary and if they can be represented multiple expressions that are easier to understand.

5.1 Prerequisites

The functions `str_view()` and `str_view_all()` from the **stringr** package (part of tidyverse) will be used to learn regular expressions interactively. `str_view()` shows the first match while `str_view_all()` shows all the matches.

```
library(tidyverse)
```

5.2 Basic matches

The most basic form of matching is to match exact strings

```
x <- c("abc ABC\n123. !?\\(){}")
writeLines(x)
## abc ABC
## 123. !?\(){}
```

```
str_view(x, "abc")
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

```
str_view(x, "123")
```

5.3 Character classes

Character classes allows you to specify a list of characters for matching.

A bracket `[...]` can be used to specify a list of character. Therefore, it will match any characters that was specified within the brackets.

```
str_view_all(x, "[bcde]")
```

If a caret `^` is added to the start of the list of characters, it will match any characters that are NOT in the list.

```
str_view_all(x, "[^bcde]")
```

You can also specify a range expression using a hyphen `-` between two characters.

```
str_view_all(x, "[a-zA-Z]")
```

You can also specify character classes using pre-defined names of these classes with a bracket expression.

Regex	What it matches
[:alnum:]	letters and numbers
[:alpha:]	letters
[:upper:]	uppercase letters
[:lower:]	lowercase letters
[:digit:]	numbers
[:punct:]	punctuations
[:space:]	space characters

Try out some of the named classes using the function `str_view_all()`

```
str_view_all(x, "[:alnum:]")
```

```
str_view_all(x, "[:punct:]")
```

There are also special metacharacters you can use to match entire classes of characters.

Regex	What it matches
`.`	any character except newline `"\n"`
`\d`	digit
`\D`	non-digit
`\s`	whitespace
`\S`	non-whitespace
`\t`	tab
`\n`	newline
`\w`	“word” i.e., letters (a-z and A-Z), digits (0-9) or (_)

Regex	What it matches
\W	non-“word”

Note that to include a \ in a regular expression, you need to escape it using \\. This is explained in the next subsection on escaping .

```
str_view_all(x, ".")
```

```
str_view_all(x, "\\d")
```

```
str_view_all(x, "\\D")
```

5.4 Escaping

In regular expressions, the backslash \ is used as an *escape* character and is used to “escape” any special characters that comes after the backslash. However, in R, the same backslashes \ are also used as an *escape* character in strings. For example, the string `"abc ABC 123.\n!?\\(){}"` is used to represent the characters `abc ABC 123.\n!?\(){}`. You will see that there is any additional backslash \ in the string representation. This is because \ is a special character for strings in R. Therefore, to represent a backslash \ in a string, another backslash needs to be added to escape the special representation of a \ in strings. This means that to create a string containing ””, you need to write `"\\"`.

```
x <- c("abc ABC\n123. !?\\(){}")
writeLines(x)
## abc ABC
## 123. !?\(){}
```

Therefore, to create any regular expressions that contains a backslash, you would need to use a string that contains another backslash \ to escape the backslash \ that forms a part of the regular expression.

For example, how would you create a regular expression to match the character `"."` if it is defined to match any character except newline. You would need to escape it with `\.`. However, the backslash is a special character in a string. Therefore you need the string `"\\."` to represent the regular expression `\.`. The same logic is applied when representing metacharacters such as `\d`, `\D`, `\w`, `\W`, etc. You need to use “\” to represent \ in regular expressions.

```
str_view_all(x, ".")
```

```
str_view_all(x, "\\.")
```

To represent a backslash \ as a regular expression, two levels of escape would be required! To elaborate, to represent a \ as a regular expression, you would need to escape it by creating the regular expression \\. To represent each of

these \ you need to use a string, which also requires you to add an additional \ to escape it. Therefore, to match a \ you need to write \\\\.

```
str_view(x, "\\\\")
```

5.5 Anchors

Regular expressions will match any part of a string unless you use anchors to specify positions such as the start or end of the string. Instead of characters, anchors are used to specify position.

Regex	Position
^	start of string
$	end of string
\b	word boundaries
\B	non-word boundaries

You can use ^ to match the start of a string and $ to match the end of a string. For a complete match, you could anchor a regular expression with ^...$.

```
str_view_all(c("apple", "apple pie", "juicy apple"), "apple")
```

\b is used to match a position known as word boundary. You can use \b to match the start or end of a word. You can think of \B as the inverse of \b and basically it matches any position that \b does not.

```
str_view_all(c("apple", "pineapple", "juicy apple"), "\\bapple\\b")
```

5.6 Quantifies

You can use quantifiers to specify the number of times a pattern matches.

Regex	Number of times
+	one or more
*	zero or more
?	zero or one
{m}	exactly m
{m,}	at least m (m or more)
{m,n}	between m and n (both inclusive)

```
str_view(c("a", "abb", "abbb"), "ab+")
```

```
str_view(c("a", "abb", "abbb"), "ab*")
```

```
str_view(c("a", "abb", "abbb"), "ab?")
```

```
str_view(c("a", "abb", "abbb"), "ab{1}")
```

```
str_view(c("a", "abb", "abbb"), "ab{1,}")
```

```
str_view(c("a", "abb", "abbb"), "ab{1,2}")
```

By default, quantifies are applied to a single character. You can use `(...)` to apply quantifies to more than one character.

```
str_view(c("ab", "abab", "ababab"), "ab+")
```

```
str_view(c("ab", "abab", "ababab"), "(ab)+")
```

Chapter 6

Functions

Writing functions is a simple way to automate commonly used code in a more general way. Therefore, you should write a function when you realized that you are copying and pasting a code block more than once. Particularly, you will see the power of functions as we start writing functions to define building energy efficient measures that we wish to apply to the building energy simulation.

Additionally, using functions helps you to avoid making mistakes that happen all too often when copying and pasting code. Additionally, as your requirements change, you only need to update the code in one place (i.e., within the function) instead of searching and updating pieces of code, which is again error-prone.

6.1 Prerequisites

We will use base R code to illustrate creating user-defined functions. Therefore, no additional packages are required.

6.2 User-defined function

The code block below shows the syntax for creating an R user-defined function using the `function` keyword. In general, a function * Function name (`function_name`): This is the name of the function that you will be creating and the name that you will use to call the function. Following section **??**, function names will use `snake_case` and be reflective of what it does. * Function arguments (`arg1`, `arg2`, `...`): These are inputs to the function that will be used by the code that you place inside the function. * Function body: You place your code inside the body of a function that is enclosed by braces `{Function body}` * Return statement: Unlike other programming languages, it is not necessary to include the `return()` statement in R. R automatically returns the last line of the body of the function.

```
function_name <- function(arg1, arg2, ...){
  Function body
}
```

6.3 Examples

The Coefficient of Variation of the Root-Mean-Square Error (CVRMSE) (equation (6.1)) and Normalized mean bias error (NMBE) (equation (6.2)) are two of the most widely used indices when evaluating how well an energy models describes the variability in measured data [ASHRAE, 2014]. As an example, we demonstrate how these equations would translate to a user-defined function in R.

$$\sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{(n-p)}} \Big/ \bar{y} \tag{6.1}$$

$$\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{(n-p) \times \bar{y}} \tag{6.2}$$

where n is the number of observations; y_i is the ith observation; $\hat{y}_i$ is the ith prediction; $\bar{y}$ is the mean of the observed values.

You can use a function to compute CVRMSE as follows:

```
cvrmse <- function(meas, pred, p = 1) {
  se <- (meas - pred)^2
  n <- length(se)
  sqrt(sum(se) / (n - p)) / mean(meas) # last line automatically returned
}

cvrmse(c(1, 2, 3), c(2, 4, 6))
## [1] 1.322876
```

You can use a function to compute NMBE as follows:

```
nmbe <- function(meas, pred, p = 1) {
  be <- (meas - pred)
  n <- length(be)
  (sum(be) / (n - p)) / mean(meas) # last line automatically returned
}
nmbe(c(1, 2, 3), c(2, 4, 6))
## [1] -1.5
```

Chapter 7

Manipulating Data

Data-preprocessing is an important step in any data science process. Rarely will you receive data that is in the exact form that you need. More often than not, you would need to pre-process the data by transforming and manipulating them so that they are easier to work with. This will be the focus in this chapter.

7.1 Prerequisites

In this chapter we focus on the use of the **dplyr** package that forms a core part of the **tidyverse** package. **dplyr** provides function that alleviates the challenges of data manipulation.

```
library(tidyverse)
library(lubridate)
library(here)
```

To explore the basic data manipulation capabilities of dplyr package, we will use the `building_meter` dataset. If the following code returns `FALSE`, you should refer to the section Finding your file in the Preface on where you can download the datasets.

This dataset contains 8,760 time-series observations of typical building meter measurements that is simulated using the Department of Energy's (DOE) reference building energy model for medium sized offices [Deru et al., 2011].

```
bldg <- read_csv(here("data", "building_meter.csv"))
##
## -- Column specification --------------------------------------------------------
## cols(
##   `Date/Time` = col_datetime(format = ""),
##   `Cooling:Electricity [J](Hourly)` = col_double(),
```

```
##   `Heating:NaturalGas [J](Hourly)` = col_double(),
##   `Heating:Electricity [J](Hourly)` = col_double(),
##   `InteriorLights:Electricity [J](Hourly)` = col_double(),
##   `ExteriorLights:Electricity [J](Hourly)` = col_double(),
##   `InteriorEquipment:Electricity [J](Hourly)` = col_double(),
##   `Fans:Electricity [J](Hourly)` = col_double(),
##   `Pumps:Electricity [J](Hourly)` = col_double(),
##   `WaterSystems:NaturalGas [J](Hourly)` = col_double()
## )
bldg
## # A tibble: 8,760 x 10
##    `Date/Time`         `Cooling:Electrici~ `Heating:NaturalG~ `Heating:Electric~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-01-01 01:00:00                   0          13517796.          182286476.
##  2 2020-01-01 02:00:00                   0          17717617.          177367416.
##  3 2020-01-01 03:00:00                   0          25310615.          235785509
##  4 2020-01-01 04:00:00                   0          20001989.          184762495.
##  5 2020-01-01 05:00:00                   0          27925599.          245249657.
##  6 2020-01-01 06:00:00                   0          21886933.          190670331.
##  7 2020-01-01 07:00:00                   0          29745424.          252272477.
....
```

Non-word characters such as spaces may make it difficult to reference columns. Therefore, a basic rule is to avoid naming columns using names that contain these characters in the first place. However, sometimes it is out of your control because you did not create the dataset but received it from source where the naming was done by someone else.

Take for example the `building.csv` data that we just parsed into R. `names()` when applied to a `data.frame()` returns the column names of the `data.frame`

```
names(bldg)
##  [1] "Date/Time"
##  [2] "Cooling:Electricity [J](Hourly)"
##  [3] "Heating:NaturalGas [J](Hourly)"
##  [4] "Heating:Electricity [J](Hourly)"
##  [5] "InteriorLights:Electricity [J](Hourly)"
##  [6] "ExteriorLights:Electricity [J](Hourly)"
##  [7] "InteriorEquipment:Electricity [J](Hourly)"
##  [8] "Fans:Electricity [J](Hourly)"
##  [9] "Pumps:Electricity [J](Hourly)"
## [10] "WaterSystems:NaturalGas [J](Hourly)"
....
```

An easy way to remove non-word characters is to use the `str_replace_all()` function to replace non-word characters with an underscore `_`. Referring back to Chapter @ref{regex}, remember that `\\W` can be used to match non-word class

of characters while + is a quantifies that can be used to match a pattern one or more times. Therefore, `str_replace_all(names(bldg), "\\W+", "_")` would replace all non-word characters that appear one or more times in the character vector `names(bldg)` with an underscore `_`.

```
new_name <- str_replace_all(names(bldg), "\\W+", "_")
```

You can also remove parts of the character vector that you do not want using the function `str_remove_all()`.

```
names(bldg) <- str_remove_all(new_name, "_Hourly_")
```

7.2 Data transformation

The cheat sheet for dplyr provides a good summary of data transformation functions covered in this chapter.

- `select()`: To select columns based on their names.
- `filter()`: To subset rows based on their values.
- `mutate()`: To add new variables whose values are a function of existing variables.
- `arrange()`: To sort the dataset based on the values of selected columns
- `summarise()`:
- `join.R`: generic functions that joins two tables together

7.2.1 `select()`

You can use the function `select()` to choose the variables to retain in the dataset based on their names. It is not uncommon to get datasets with hundred or thousands of variables. An example is when working with data from the building management system. Under these scenarios, it is often very useful to narrow down the dataset to contain only the variables of interest. `select()` allows you to do this very easily, to zoom in on the variables or columns of interest.

```
select(bldg, Date_Time, Cooling_Electricity_J, Heating_Electricity_J)
## # A tibble: 8,760 x 3
##    Date_Time           Cooling_Electricity_J Heating_Electricity_J
##    <dttm>                              <dbl>                 <dbl>
##  1 2020-01-01 01:00:00                     0             182286476.
##  2 2020-01-01 02:00:00                     0             177367416.
##  3 2020-01-01 03:00:00                     0             235785509
##  4 2020-01-01 04:00:00                     0             184762495.
##  5 2020-01-01 05:00:00                     0             245249657.
##  6 2020-01-01 06:00:00                     0             190670331.
##  7 2020-01-01 07:00:00                     0             252272477.
....
```

You can also optionally rename columns with the `select()` function.

```
select(bldg,
  datetime = Date_Time,
  cooling = Cooling_Electricity_J,
  heating = Heating_Electricity_J
)
## # A tibble: 8,760 x 3
##    datetime            cooling    heating
##    <dttm>                <dbl>      <dbl>
##  1 2020-01-01 01:00:00       0 182286476.
##  2 2020-01-01 02:00:00       0 177367416.
##  3 2020-01-01 03:00:00       0 235785509
##  4 2020-01-01 04:00:00       0 184762495.
##  5 2020-01-01 05:00:00       0 245249657.
##  6 2020-01-01 06:00:00       0 190670331.
##  7 2020-01-01 07:00:00       0 252272477.
....
```

Additionally, tidyverse provides helper functions that lets you select columns by matching patterns in their names. For the following selection helper functions, a character vector is provided for the match. If `length()` of the character vector is larger than 1, the logical union (OR `|` operator) is taken.

- `starts_with()`: Select columns that starts with a specific character vector

```
select(bldg, starts_with(c("Heating", "Cooling")))
## # A tibble: 8,760 x 3
##    Heating_NaturalGas_J Heating_Electricity_J Cooling_Electricity_J
##                   <dbl>                 <dbl>                 <dbl>
##  1            13517796.             182286476.                     0
##  2            17717617.             177367416.                     0
##  3            25310615.             235785509                      0
##  4            20001989.             184762495.                     0
##  5            27925599.             245249657.                     0
##  6            21886933.             190670331.                     0
##  7            29745424.             252272477.                     0
....
```

- `ends_with()`: Select columns that ends with a specific character vector

```
select(bldg, ends_with("Time"))
## # A tibble: 8,760 x 1
##    Date_Time
##    <dttm>
##  1 2020-01-01 01:00:00
##  2 2020-01-01 02:00:00
##  3 2020-01-01 03:00:00
```

```
##  4 2020-01-01 04:00:00
##  5 2020-01-01 05:00:00
##  6 2020-01-01 06:00:00
##  7 2020-01-01 07:00:00
....
```

- `contains()`: Select columns that contains a specific character vector

```
select(
  bldg,
  contains(
    c(
      "Time",
      "electricity"
    )
  )
)
## # A tibble: 8,760 x 8
##    Date_Time           Cooling_Electrici~ Heating_Electrici~ InteriorLights_Ele~
##    <dttm>                          <dbl>              <dbl>               <dbl>
##  1 2020-01-01 01:00:00                 0         182286476.             9649498.
##  2 2020-01-01 02:00:00                 0         177367416.             9649498.
##  3 2020-01-01 03:00:00                 0         235785509              9649498.
##  4 2020-01-01 04:00:00                 0         184762495.             9649498.
##  5 2020-01-01 05:00:00                 0         245249657.             9649498.
##  6 2020-01-01 06:00:00                 0         190670331.             9649498.
##  7 2020-01-01 07:00:00                 0         252272477.             9649498.
....
```

- `matches()`: Select columns that matches a regular expression. `rename()` is a dplyr function that allows you to change the names of individual columns using `new_name = old_name`. Referring back to Chapter @ref{regex}, remember that the `.` is used to match any character except newline `\n` while `?` is a quantifies that can be used to match a pattern zero or one times.

```
reg_ex <- "[dD]ate.?[tT]ime"

select(bldg, matches(reg_ex, ignore.case = FALSE))
## # A tibble: 8,760 x 1
##    Date_Time
##    <dttm>
##  1 2020-01-01 01:00:00
##  2 2020-01-01 02:00:00
##  3 2020-01-01 03:00:00
##  4 2020-01-01 04:00:00
##  5 2020-01-01 05:00:00
```

```
##  6 2020-01-01 06:00:00
##  7 2020-01-01 07:00:00
....

bldg_rename_a <- rename(bldg, `Date/Time` = `Date_Time`)
select(bldg_rename_a, matches(reg_ex, ignore.case = FALSE))
## # A tibble: 8,760 x 1
##    `Date/Time`
##    <dttm>
##  1 2020-01-01 01:00:00
##  2 2020-01-01 02:00:00
##  3 2020-01-01 03:00:00
##  4 2020-01-01 04:00:00
##  5 2020-01-01 05:00:00
##  6 2020-01-01 06:00:00
##  7 2020-01-01 07:00:00
....


bldg_rename_b <- rename(bldg, `datetime` = `Date_Time`)
select(bldg_rename_b, matches(reg_ex, ignore.case = FALSE))
## # A tibble: 8,760 x 1
##    datetime
##    <dttm>
##  1 2020-01-01 01:00:00
##  2 2020-01-01 02:00:00
##  3 2020-01-01 03:00:00
##  4 2020-01-01 04:00:00
##  5 2020-01-01 05:00:00
##  6 2020-01-01 06:00:00
##  7 2020-01-01 07:00:00
....
```

- `all_of()` and `any_of()`: You can also select variables from a character vector using the selection helper functions `all_of()` and `any_of()`. The key difference is that `all_of()` is used for strict selection where an error is thrown for variable names that do not exist. In contrast, `any_of()` does not check for missing variable names, and thus is often used to ensure a specific column is removed.

```
vars <- c("Heating_NaturalGas_J", "WaterSystems_NaturalGas_J")
select(bldg, all_of(vars))
## # A tibble: 8,760 x 2
##    Heating_NaturalGas_J WaterSystems_NaturalGas_J
##                   <dbl>                     <dbl>
##  1            13517796.                     72000
```

```
##  2            17717617.                   72000
##  3            25310615.                 3970732.
##  4            20001989.                   72000
##  5            27925599.                 3970995.
##  6            21886933.                   72000
##  7            29745424.                 3970782.
....

select(bldg, -any_of(vars))
## # A tibble: 8,760 x 8
##    Date_Time           Cooling_Electrici~ Heating_Electrici~ InteriorLights_Ele~
##    <dttm>                           <dbl>              <dbl>               <dbl>
##  1 2020-01-01 01:00:00                  0         182286476.            9649498.
##  2 2020-01-01 02:00:00                  0         177367416.            9649498.
##  3 2020-01-01 03:00:00                  0         235785509             9649498.
##  4 2020-01-01 04:00:00                  0         184762495.            9649498.
##  5 2020-01-01 05:00:00                  0         245249657.            9649498.
##  6 2020-01-01 06:00:00                  0         190670331.            9649498.
##  7 2020-01-01 07:00:00                  0         252272477.            9649498.
....
```

- `where()` is a very useful selection helper function when you want to apply a function (that returns `TRUE` or `FALSE`) to all columns and select only those columns where the function returns `TRUE`.

```
select(bldg, where(is.numeric))
## # A tibble: 8,760 x 9
##    Cooling_Electrici~ Heating_NaturalGa~ Heating_Electrici~ InteriorLights_Elec~
##                 <dbl>              <dbl>              <dbl>               <dbl>
##  1                  0          13517796.         182286476.            9649498.
##  2                  0          17717617.         177367416.            9649498.
##  3                  0          25310615.         235785509             9649498.
##  4                  0          20001989.         184762495.            9649498.
##  5                  0          27925599.         245249657.            9649498.
##  6                  0          21886933.         190670331.            9649498.
##  7                  0          29745424.         252272477.            9649498.
....
```

You can also define the function to be applied within `where()`

```
select(
  bldg,
  where(
    function(x) is.numeric(x)
  )
)
```

```
## # A tibble: 8,760 x 9
##    Cooling_Electrici~ Heating_NaturalGa~ Heating_Electrici~ InteriorLights_Elec~
##                 <dbl>              <dbl>              <dbl>                <dbl>
##  1                  0          13517796.          182286476.            9649498.
##  2                  0          17717617.          177367416.            9649498.
##  3                  0          25310615.          235785509             9649498.
##  4                  0          20001989.          184762495.            9649498.
##  5                  0          27925599.          245249657.            9649498.
##  6                  0          21886933.          190670331.            9649498.
##  7                  0          29745424.          252272477.            9649498.
....
```

7.2.2 `filter()`

You can use `filter()` to subset rows based on their values. dplyr's `filter()` function retains rows that return `TRUE` for all the conditions specified. The first argument is the data frame and the subsequent arguments are the conditions that are used to subset the observations. In other words, multiple arguments to `filter()` are combined by the `&` (and) logical operator. To work with other logical operators such as `|` (or) and `!` (not), you would have to combine them within a single argument.

```
filter(bldg, Cooling_Electricity_J != 0)
## # A tibble: 2,678 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-02-11 15:00:00            4354833.                  0           91807743.
##  2 2020-02-11 16:00:00            2387624.                  0           94118327.
##  3 2020-02-23 09:00:00            7606370.                  0          102607149.
##  4 2020-02-23 10:00:00           21074975.                  0           76304715.
##  5 2020-02-23 11:00:00           23052733.                  0           63182232.
##  6 2020-02-23 12:00:00           23185185.                  0           53130886.
##  7 2020-02-23 13:00:00           20501352.                  0           68037068.
....
filter(bldg, InteriorLights_Electricity_J > 100000000)
## # A tibble: 2,520 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-01-02 09:00:00                   0          128483630.          473529413.
##  2 2020-01-02 10:00:00                   0          109562756.          398893378.
##  3 2020-01-02 11:00:00                   0           88363562.          350234124.
##  4 2020-01-02 12:00:00                   0           75198754.          305357974
##  5 2020-01-02 13:00:00                   0           70573992.          300958898.
##  6 2020-01-02 14:00:00                   0           84277786.          249050033.
##  7 2020-01-02 15:00:00                   0           99713290.          238506555.
....
```

```
filter(
  bldg, Cooling_Electricity_J != 0,
  Cooling_Electricity_J < 4000000 |
    Cooling_Electricity_J > 22000000,
)
## # A tibble: 2,428 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                           <dbl>              <dbl>              <dbl>
##  1 2020-02-11 16:00:00            2387624.                  0           94118327.
##  2 2020-02-23 11:00:00           23052733.                  0           63182232.
##  3 2020-02-23 12:00:00           23185185.                  0           53130886.
##  4 2020-02-23 20:00:00            1387715.                  0          101207501.
##  5 2020-03-10 12:00:00           25148879.                  0           17802883.
##  6 2020-03-10 13:00:00           31261163.                  0           15048079.
##  7 2020-03-10 14:00:00           30172661.                  0            1791300.
....
```

Often, the ability to subset observations based on their timestamp is a very useful feature when working with timeseries data

```
filter(bldg, month(Date_Time) == 1, day(Date_Time) == 1)
## # A tibble: 24 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                           <dbl>              <dbl>              <dbl>
##  1 2020-01-01 01:00:00                  0           13517796.          182286476.
##  2 2020-01-01 02:00:00                  0           17717617.          177367416.
##  3 2020-01-01 03:00:00                  0           25310615.          235785509
##  4 2020-01-01 04:00:00                  0           20001989.          184762495.
##  5 2020-01-01 05:00:00                  0           27925599.          245249657.
##  6 2020-01-01 06:00:00                  0           21886933.          190670331.
##  7 2020-01-01 07:00:00                  0           29745424.          252272477.
....
filter(bldg, day(Date_Time) == 1)
## # A tibble: 287 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                           <dbl>              <dbl>              <dbl>
##  1 2020-01-01 01:00:00                  0           13517796.          182286476.
##  2 2020-01-01 02:00:00                  0           17717617.          177367416.
##  3 2020-01-01 03:00:00                  0           25310615.          235785509
##  4 2020-01-01 04:00:00                  0           20001989.          184762495.
##  5 2020-01-01 05:00:00                  0           27925599.          245249657.
##  6 2020-01-01 06:00:00                  0           21886933.          190670331.
##  7 2020-01-01 07:00:00                  0           29745424.          252272477.
....
filter(bldg, month(Date_Time) > 4, month(Date_Time) < 10)
```

```
## # A tibble: 3,672 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-05-01 00:00:00                   0                  0                  0
##  2 2020-05-01 01:00:00                   0                  0                  0
##  3 2020-05-01 02:00:00                   0                  0                  0
##  4 2020-05-01 03:00:00                   0                  0                  0
##  5 2020-05-01 04:00:00                   0                  0                  0
##  6 2020-05-01 05:00:00                   0                  0                  0
##  7 2020-05-01 06:00:00                   0                  0          252935532.
....
filter(bldg, month(Date_Time) > 4 & month(Date_Time) < 10)
## # A tibble: 3,672 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-05-01 00:00:00                   0                  0                  0
##  2 2020-05-01 01:00:00                   0                  0                  0
##  3 2020-05-01 02:00:00                   0                  0                  0
##  4 2020-05-01 03:00:00                   0                  0                  0
##  5 2020-05-01 04:00:00                   0                  0                  0
##  6 2020-05-01 05:00:00                   0                  0                  0
##  7 2020-05-01 06:00:00                   0                  0          252935532.
....
```

You can also combine `filter()` with `grepl()` to subset rows using regular expressions.

7.2.3 `arrange()`

You can use `arrange()` to sort the dataset based on the values of selected columns. The first argument is the data frame and the second is the column name to sort the data frame by. If more than one column name is provided, subsequent column names will be used to break ties in the preceeding columns.

```
arrange(bldg, Cooling_Electricity_J)
## # A tibble: 8,760 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-01-01 01:00:00                   0          13517796.          182286476.
##  2 2020-01-01 02:00:00                   0          17717617.          177367416.
##  3 2020-01-01 03:00:00                   0          25310615.          235785509
##  4 2020-01-01 04:00:00                   0          20001989.          184762495.
##  5 2020-01-01 05:00:00                   0          27925599.          245249657.
##  6 2020-01-01 06:00:00                   0          21886933.          190670331.
##  7 2020-01-01 07:00:00                   0          29745424.          252272477.
....
```

You can use `desc()` to order a column in descending order

```
arrange(bldg, desc(Cooling_Electricity_J))
## # A tibble: 8,760 x 10
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                           <dbl>              <dbl>              <dbl>
##  1 2020-07-19 16:00:00          356330907.                  0                  0
##  2 2020-07-19 15:00:00          349038364.                  0                  0
##  3 2020-07-18 16:00:00          339376366.                  0                  0
##  4 2020-07-19 17:00:00          338918865.                  0                  0
##  5 2020-07-17 15:00:00          337245696.                  0                  0
##  6 2020-07-18 14:00:00          336127683.                  0                  0
##  7 2020-07-17 16:00:00          335013419.                  0                  0
....
```

7.2.4 `mutate()`

You can use `mutate()` to add new variables whose values are a function of existing variables (Note that you can also refer to newly created variables). The newly created variables will be added to the end of the dataset.

```
mutate(bldg,
  Total_Heating_J = Heating_NaturalGas_J + Heating_Electricity_J,
  Cooling_Heating_kWh = (Total_Heating_J + Cooling_Electricity_J) * 2.77778e-7
)
## # A tibble: 8,760 x 12
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                           <dbl>              <dbl>              <dbl>
##  1 2020-01-01 01:00:00                  0          13517796.          182286476.
##  2 2020-01-01 02:00:00                  0          17717617.          177367416.
##  3 2020-01-01 03:00:00                  0          25310615.          235785509
##  4 2020-01-01 04:00:00                  0          20001989.          184762495.
##  5 2020-01-01 05:00:00                  0          27925599.          245249657.
##  6 2020-01-01 06:00:00                  0          21886933.          190670331.
##  7 2020-01-01 07:00:00                  0          29745424.          252272477.
....
```

You can use `transmute()` if you only want to keep the new variables in your dataset.

```
transmute(bldg,
  Total_Heating_J = Heating_NaturalGas_J + Heating_Electricity_J,
  Cooling_Heating_kWh = (Total_Heating_J + Cooling_Electricity_J) * 2.77778e-7
)
## # A tibble: 8,760 x 2
##    Total_Heating_J Cooling_Heating_kWh
##              <dbl>               <dbl>
```

```
##  1      195804272.                54.4
##  2      195085034.                54.2
##  3      261096124.                72.5
##  4      204764484.                56.9
##  5      273175257.                75.9
##  6      212557265.                59.0
##  7      282017901.                78.3
....
```

Useful mutate and transmute functions include

- Arithmetic operators (+, -, *,/, etc.) and logarithms (log(),log2()‘, etc.)

```
heat <- transmute(bldg,
  Total_Heating_J = Heating_NaturalGas_J + Heating_Electricity_J
  )
heat
## # A tibble: 8,760 x 1
##    Total_Heating_J
##              <dbl>
##  1      195804272.
##  2      195085034.
##  3      261096124.
##  4      204764484.
##  5      273175257.
##  6      212557265.
##  7      282017901.
....
```

- `lead()` and `lag()`, allowing you to create lead and lag variables that are useful especially when working with timeseries data.

```
mutate(
  heat,
  lag_1 = lag(Total_Heating_J),
  lag_2 = lag(lag_1),
  lag_3 = lag(Total_Heating_J, 3)
)
## # A tibble: 8,760 x 4
##    Total_Heating_J      lag_1      lag_2      lag_3
##              <dbl>      <dbl>      <dbl>      <dbl>
##  1      195804272.         NA         NA         NA
##  2      195085034. 195804272.         NA         NA
##  3      261096124. 195085034. 195804272.         NA
##  4      204764484. 261096124. 195085034. 195804272.
##  5      273175257. 204764484. 261096124. 195085034.
```

```
##  6      212557265. 273175257. 204764484. 261096124.
##  7      282017901. 212557265. 273175257. 204764484.
....
```

- `if_else()` is function of the form `if_else(condition, true, false)` that allows you to test a condition that you provide as a first argument to the function. If the condition is `TRUE`, `if_else()` will return the second argument. By contrast, if the condition is `FALSE`, it will return the third argument. In the example below, we are using mutate to create a new variable/column `New_Total_Heating_J`. This new column will contain the characters "low" if the condition `Total_Heating_J < 200000000` is `TRUE` and "high" if it is `FALSE`.

```
mutate(
  heat,
  New_Total_Heating_J = if_else(Total_Heating_J < 200000000, "low", "high")
  )
## # A tibble: 8,760 x 2
##    Total_Heating_J New_Total_Heating_J
##              <dbl> <chr>
##  1      195804272. low
##  2      195085034. low
##  3      261096124. high
##  4      204764484. high
##  5      273175257. high
##  6      212557265. high
##  7      282017901. high
....
```

- `case_when()` is an extension of `if_else()` by allowing you to write multiple `if_else()` statements. Below, we assign a `"low"` when `Total_Heating_J < 200000000`, `"medium"` when `200000000 <= Total_Heating_J <= 400000000`, and high when `Total_Heating_J > 400000000`.

```
mutate(
  heat,
  New_Total_Heating_J = case_when(
    Total_Heating_J < 200000000 ~ "low",
    Total_Heating_J >= 200000000 & Total_Heating_J <= 400000000 ~ "medium",
    Total_Heating_J > 400000000 ~ "high")
  )
## # A tibble: 8,760 x 2
##    Total_Heating_J New_Total_Heating_J
##              <dbl> <chr>
##  1      195804272. low
##  2      195085034. low
```

```
##  3      261096124. medium
##  4      204764484. medium
##  5      273175257. medium
##  6      212557265. medium
##  7      282017901. medium
....
```

7.2.5 summarise() and group_by()

summarise() or summarize() collapse a data frame into one row for each combination of grouping variables. Therefore, it is not very useful by itself.

```
summarise(bldg,
  Heating_Total = sum(Heating_Electricity_J),
  Heating_Avg = mean(Heating_Electricity_J),
  Heating_Peak = max(Heating_Electricity_J)
)
## # A tibble: 1 x 3
##   Heating_Total Heating_Avg Heating_Peak
##           <dbl>       <dbl>        <dbl>
## 1 508713824519.   58072354.   1002989103
```

However, summarise() combined with group_by() will allow you to easily summarise data based on individual groups.

```
bldg_month <- mutate(bldg,
  Year = year(Date_Time),
  Month = month(Date_Time)
)

bldg_by_month <- group_by(bldg_month, Year, Month)

summarise(bldg_by_month,
  Heating_Total = sum(Heating_Electricity_J),
  Heating_Avg = mean(Heating_Electricity_J),
  Heating_Peak = max(Heating_Electricity_J)
)
## `summarise()` has grouped output by 'Year'. You can override using the `.groups` ar
## # A tibble: 13 x 5
## # Groups:   Year [2]
##     Year Month Heating_Total Heating_Avg Heating_Peak
##    <dbl> <dbl>         <dbl>       <dbl>        <dbl>
##  1  2020     1 124527837066.  167601396.   978585992.
##  2  2020     2  89836688374.  133486907.  1002989103
##  3  2020     3  55674915533.   74932592.   795581602.
##  4  2020     4  32036849861.   44495625.   639342141.
```

```
##  5  2020     5   7726081874.   10384519.   252935532.
##  6  2020     6    624415414.     867244.    39920827.
....
```

7.2.6 across()

`across()` makes it possible to apply functions across multiple columns within the functions `summarise()` and `mutate()`, using the semantics in `filter()` (see section 7.2.1) that makes it easy to refer to columns based on their names.

```
mutate(bldg,
  total = rowSums(across(where(is.numeric)))
)
## # A tibble: 8,760 x 11
##    Date_Time           Cooling_Electricit~ Heating_NaturalGa~ Heating_Electrici~
##    <dttm>                            <dbl>              <dbl>              <dbl>
##  1 2020-01-01 01:00:00                   0          13517796.          182286476.
##  2 2020-01-01 02:00:00                   0          17717617.          177367416.
##  3 2020-01-01 03:00:00                   0          25310615.          235785509
##  4 2020-01-01 04:00:00                   0          20001989.          184762495.
##  5 2020-01-01 05:00:00                   0          27925599.          245249657.
##  6 2020-01-01 06:00:00                   0          21886933.          190670331.
##  7 2020-01-01 07:00:00                   0          29745424.          252272477.
....
```

You may also pass a list of functions to be applied to each of the selected columns.

```
summarise(
  bldg_by_month,
  across(
    contains("electricity"),
    list(Total = sum, Avg = mean, Peak = max)
  )
)
## `summarise()` has grouped output by 'Year'. You can override using the `.groups` argument.
## # A tibble: 13 x 23
## # Groups:   Year [2]
##     Year Month Cooling_Electricity_J~ Cooling_Electricity~ Cooling_Electricity_~
##    <dbl> <dbl>                 <dbl>               <dbl>                <dbl>
##  1  2020     1                     0                   0                    0
##  2  2020     2             181014017.              268966.           23185185.
##  3  2020     3             775169988.             1043297.           79053134.
##  4  2020     4           11551697867.            16044025.          226399417.
##  5  2020     5           22952133253.            30849641.          191404162.
##  6  2020     6           53405646191.            74174509.          302585966
```

```
....
```

7.3 Pipes

The `%>%` is a forward pipe operator from the `magrittr` package that comes loaded with the `tidyverse` package. The pipe operator `%>%` is a very handy that allows operations to be performed sequentially. In general, you can think of it as sending the output of one function as an input to the next .

With the pipe operator,

Instead of many intermediary steps

```
first_output <- first(data)
second_output <- second(first_output)
final_result <- third(second_output)
```

You can connect multiple operators sequentially using the pipe operator

```
final_result <- first(data) %>%
  second() %>%
  third()
```

Figure 7.1 shows what is happening graphically.

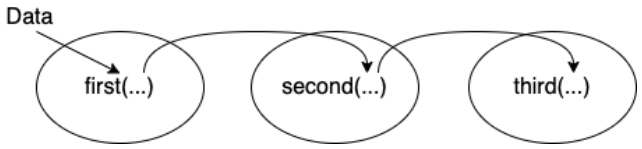


Figure 7.1: Piping data from one function to the next sequentially

An example using the `bldg` dataset.

```
bldg %>%
  select(Date_Time, contains("Heating")) %>%
  mutate(
    Month = month(Date_Time),
    Total_Heating_J = rowSums(across(where(is.numeric)))
  ) %>%
  group_by(Month) %>%
  summarise(across(
    where(is.numeric),
    list(Total = sum, Peak = max)
  ))
## # A tibble: 12 x 7
##    Month Heating_NaturalG~ Heating_NaturalG~ Heating_Electric~ Heating_Electric~
```

```
##    <dbl>          <dbl>          <dbl>          <dbl>          <dbl>
##  1     1   59660445562.    527825997.    124648935250.    978585992.
##  2     2   41029360544.    530570172.     89836688374.   1002989103
##  3     3    9377325423.    230991815.     55674915533.    795581602.
##  4     4    3352997015.    156525821      32036849861.    639342141.
##  5     5       2144014.      2144014.      7726081874.    252935532.
##  6     6              0             0       624415414.     39920827.
##  7     7              0             0       128302310.     16757642
....
```

Part II

Get Started

Chapter 8

Introduction

The goal of this part of the book is to get you started working with EnergyPlus in R as quickly as possible. In this part of the book, you will learn how to get your model into R and run the simulation (Chapter 9). You will then learn how to extract predifined annual summary reports and manipulate them so that the focus is placed on the important variables and observations (Chapter 10). Finally, we provide a gentle introduction to visualizing the data from the summary reports in R (Chapter 11).

Understanding the details of EnergyPlus inputs and outputs and how they are structured are an important part of building energy simulation. We will come back to that in the next part of the book once we have gotten you started up with the basics.

Chapter 9

Parse then simulate

9.1 Prerequisites

In this chapter we will introduce the **eplusr** package that was designed to interface with EnergyPlus and ease the application of data driven analytics. If you like to learn more about the underlying theory of how data is structured under the hood, you might enjoy the *eplusr* paper published in the Journal Energy and Buildings [Jia and Chong, 2021].

```
library(eplusr)
library(here)
```

In this book, you will learn using the U.S. Department of Energy (DOE) Commercial Reference Building for medium office energy model [Deru et al., 2011]. We use the DOE reference building because it is well established and widely used. More importantly, it is publicly available and comes distributed with EnergyPlus as example files, making the examples and code in this book easier to follow and reproduce.

For your convenience, the model can be found at `./data/idf/RefBldgMediumOfficeNew2004_Chicago.idf` relative to the project root folder. Likewise, the weather file that we will be using can be found at `./data/epw/USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw` relative to the project root folder.

9.2 Key EnergyPlus files

Before starting to work with EnergyPlus, there are three main EnergyPlus files that you should know: the input data dictionary (IDD), the input data file (IDF), and the EnergyPlus weather file (EPW). All three files are ASCII files also know as text files. The IDD lists all possible objects in EnergyPlus along with their respective input requirements. The IDF provides a description of the

building to be simulated. The EPW contains the hourly or sub-hourly weather data (i.e., the boundary conditions) needed for the simulation.

9.3 Parsing the model

Before starting, you need to first tell R where EnergyPlus is located.

```
use_eplus("/Applications/EnergyPlus-9-4-0")
## Configure data of EnergyPlus v9.4.0 located at '/Applications/EnergyPlus-9-4-0' alr
```

To get started, you need to first be familiar with the three basic classes (`Idf`, `Epw`, and the `EplusJob`) of the eplusr package.

The `Idf` class is used to represent the EnergyPlus model. It contains methods that allows you to easily query and modify the model. You can use the `read_idf()` function to parse the EnergyPlus model into R by providing the file path to the EnergyPlus IDF file. You can use the function `here()` to easily reference the file path relative to the project root folder.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)
## IDD v9.4.0 has not been parsed before.
## Try to locate 'Energy+.idd' in EnergyPlus v9.4.0 installation folder '/Applications/
## IDD file found: '/Applications/EnergyPlus-9-4-0/Energy+.idd'.
## Start parsing...
## Parsing completed.
```

The `Epw` class is used to represent the weather data that will be used for the simulation. You can use the `read_epw()` function to parse an EPW weather file into R by providing the file path to the EnergyPlus EPW file. Likewise, the function `here()` is used to easily reference the file path relative to the project root folder.

```
path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)
```

The `EplusJob` is used extract the simulation results. When you run the model, a `EplusJob` object is returned. One way to run the model is to use the `$run()` method of the `Idf` class, and provide the weather file in the form of an `epw` object.

```
job <- model$run(weather = epw)
```

9.4 Simulating the model

You should check the SimulationControl and RunPeriod objects in EnergyPlus before simulating the model. The SimulationControl object

allows the user to specify the type of simulations that will be performed. You can view this object in R by navigating to and opening the `RefBldgMediumOfficeNew2004_Chicago.idf` file in R followed by searching for the string `SimulationControl` (Figure 9.1). If the value for the `Run Simulation for Weather File Run Periods` field is `NO` as illustrated in Figure 9.1, the simulation will be **not** be run on for the weather file run periods (Figure 9.2). In other words, if you want to run the simulation based on the provided epw weather file for the period specify by the RunPeriod object, check that this field is set to `YES`.

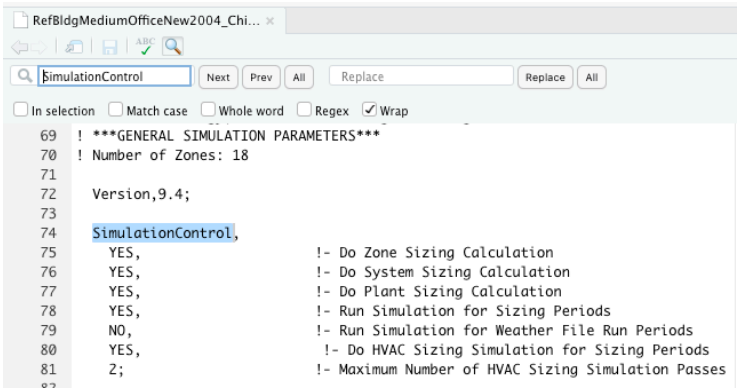


Figure 9.1: SimulationControl object in EnergyPlus.

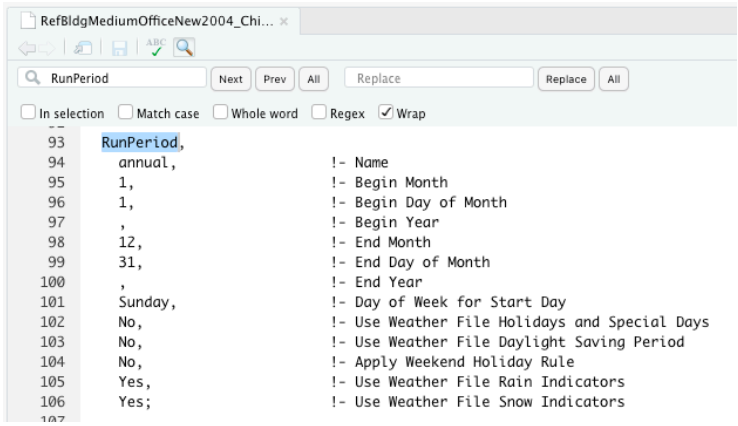


Figure 9.2: RunPeriod object in EnergyPlus.

You can use the `run()` method of the Idf class to run the simulation using EnergyPlus. You can choose the weather file to run the simulation by specifying an `Epw` object or a file path of an `.epw` file to the `weather` argument. You can also specify a directory to save the simulation results using the `dir` argument.

By default, the folder of your Idf file is used to save the simulation results.

```
job <- model$run(weather = epw,
                 dir = here("data", "idf", "run", "test_run.idf"))
```

If you do not want to save the simulation results locally, you can save the simulation output files to a temporary directory by specifying `dir = tempdir()`.

```
job <- model$run(epw, dir = tempdir())
## EnergyPlus Starting
## EnergyPlus, Version 9.4.0-998c4b761e, YMD=2021.10.13 23:01
##
## Could not find platform independent libraries <prefix>
## Could not find platform dependent libraries <exec_prefix>
## Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
## Initializing Response Factors
## Calculating CTFs for "STEEL FRAME NON-RES EXT WALL"
## Calculating CTFs for "IEAD NON-RES ROOF"
## Calculating CTFs for "EXT-SLAB"
....
```

Chapter 10

Summary reports

10.1 Prerequisites

In this chapter we will focus on how to set and get EnergyPlus simulation outputs using the eplusr package. We will illustrate the manipulation of simulation data using tidyverse, and use ggplot2 to visualize the simulation data

```
library(eplusr)
library(tidyverse)
library(ggplot2)
library(RColorBrewer)
library(here)
```

We will be working with the IDF and EPW file that pertains to the U.S. Department of Energy (DOE) Commercial Reference Building and Chicago's TMY3 respectively.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)

path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)
```

10.2 Output summary reports

To output the summary reports from EnergyPlus, you need to first run the simulation. The object `job` in the code below belongs to the `EplusJob` class. If you set `dir = NULL`, the simulation results will not be saved into a temporary directory but instead in the folder containing the IDF file. Do that and you will find the summary report in a html file as illustrated in Figure 10.1.

```
job <- model$run(epw, dir = tempdir())
## Replace the existing IDF located at /private/var/folders/61/8dz74knn0yd02c_bt8f11pk
## EnergyPlus Starting
## EnergyPlus, Version 9.4.0-998c4b761e, YMD=2021.10.13 23:02
##
## Could not find platform independent libraries <prefix>
## Could not find platform dependent libraries <exec_prefix>
## Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
## Initializing Response Factors
## Calculating CTFs for "STEEL FRAME NON-RES EXT WALL"
## Calculating CTFs for "IEAD NON-RES ROOF"
## Calculating CTFs for "EXT-SLAB"
....
class(job)
## [1] "EplusJob" "R6"
```

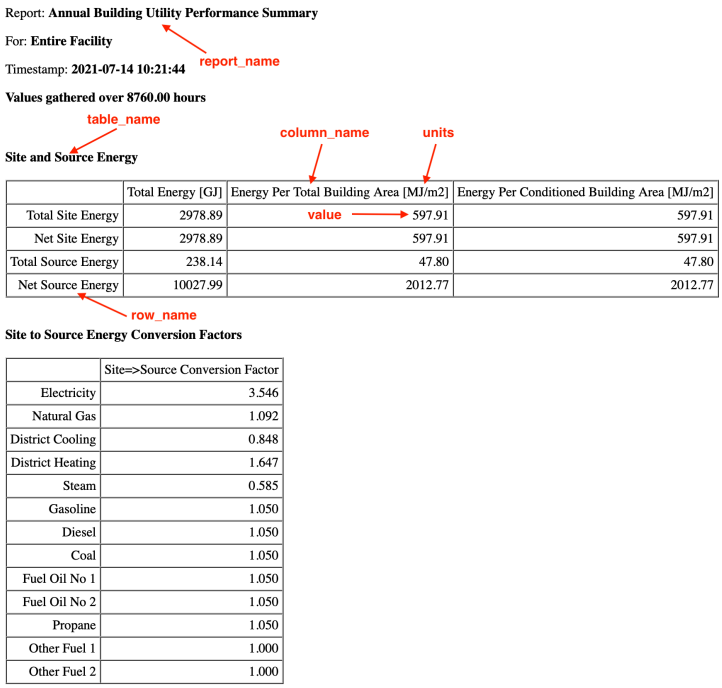


Figure 10.1: Report summary html file output from EnergyPlus

In eplusr, the `EplusJob` class provides different methods to extract simulation results. To bring the summary reports into R, you can use the `tabular_data()` method, which extracts all the EnergyPlus summary reports into a R `data.table`.

The data table consists of the following nine columns.

- case: Name of the simulation case. The case variable is useful for filtering the simulation results when running multiple parametric simulations. However, since we are only running one simulation in this example there is only one unique case.
- index: Row index for the table
- report_name: Names of the summary report that the observation belongs to
- report_for: The context that the summary report is for.
- table_name: The name of the table within the report that the observation belongs to.
- column_name: The name of the column within a table that the observation belongs to.
- row_name: The name of the row within a table that the observation belongs to.
- units: The unit of the observation.
- value: The value of the observation. Note that by default, this is stored as a character vector.

```
report <- job$tabular_data()
colnames(report)
## [1] "case"        "index"       "report_name" "report_for"  "table_name"
## [6] "column_name" "row_name"    "units"       "value"
```

To illustrate, refer back to Figure 10.1 on what the arguments `report_name`, `table_name`, `column_name`, `row_name`, `units`, and `value` would correspond to in the html summary report. You can easily retrieve the `Total Site Energy` in `Energy Per Total Building Area` from the `Site and Source Energy` table in the `AnnualBuildingUtilityPerformanceSummary` report. Note that there are no whitespaces when specifying the `report_name`.

```
report_example <- report %>%
    filter(report_name == "AnnualBuildingUtilityPerformanceSummary",
           table_name == "Site and Source Energy",
           column_name == "Energy Per Total Building Area",
           row_name == "Total Site Energy")
```

Other examples as follows.

You can retrieve end use energy consumption data and arrange them in descending value.

```
report_end_use <- report %>%
    filter(table_name == "End Uses",
           grepl("Electricity|Natural Gas", column_name, ignore.case = TRUE),
           !grepl("total", row_name, ignore.case = TRUE)) %>%
    mutate(value = as.numeric(value)*277.778,
```

```
           units = "kWh") %>%
    select(row_name, column_name, units, value) %>%
    rename(category = row_name, fuel = column_name) %>%
    arrange(desc(value)) %>%
    drop_na()
```

You can retrieve normalized metrics and group them by their fuel type.

```
report_norm <- report %>%
    filter(table_name == "Utility Use Per Conditioned Floor Area",
           row_name != "Total") %>%
    mutate(value = as.numeric(value) * 0.2777777777777778,
           units = "kWh/m2") %>%
    filter(value > 0.0) %>%
    select(category = row_name, fuel = column_name, units, value) %>%
    group_by(category) %>%
    summarise(total = sum(value))
```

You can also retrieve details of the model inputs from the summary reports.

Opaque exterior of the building envelope.

```
report_op_ext <- report %>%
    filter(table_name == "Opaque Exterior",
           grepl("Azimuth|U-Factor with Film", column_name)) %>%
    mutate(value = as.numeric(value)) %>%
    select(row_name, column_name, units, value)
```

Exterior fenestration of the building envelope.

```
report_ext_fenestration <- report %>%
    filter(table_name == "Exterior Fenestration",
           !grepl("total", row_name, ignore.case = TRUE)) %>%
    mutate(value = as.numeric(value)) %>%
    filter(value > 0) %>%
    select(row_name, column_name, units, value)
## Warning in mask$eval_all_mutate(quo): NAs introduced by coercion
```

You can refer to the section *"Output:Table:SummaryReports"* in EnergyPlus's *"Output Details and Examples"* documentation for the list of available reports.

Chapter 11

Visualize

11.1 Prerequisites

In this chapter, you will use the **eplusr** package to interface with EnergyPlus via R, the **tidyverse** package to manipulate the simulation results, and the **here** package to specify relative file paths.

Additionally, we will introduce **ggplot2**, an input-tidy visualization package that is based on the the grammar of graphics [Wilkinson, 2012]. We will also introduce **RColorBrewer** and **viridis** package that contains predefined color palettes that make it easy to pick the right one when creating graphics in R.

```
library(ggplot2)
library(RColorBrewer)
library(viridis)
```

We will also be working with the following R packages in this chapter.

```
library(eplusr)
library(tidyverse)
library(here)
library(lubridate)
```

In this chapter, you will also be working with the U.S. Department of Energy (DOE) Commercial Reference Building for medium office energy model [Deru et al., 2011] and the third and latest typical meteorological (TMY3) weather data for Chicago. You will first parse the IDf and EPW to R and run the simulation to extract the simulation results.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)
```

```
path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)

job <- model$run(weather = epw, dir = tempdir())
## EnergyPlus Starting
## EnergyPlus, Version 9.4.0-998c4b761e, YMD=2021.10.13 23:02
##
## Could not find platform independent libraries <prefix>
## Could not find platform dependent libraries <exec_prefix>
## Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
## Initializing Response Factors
## Calculating CTFs for "STEEL FRAME NON-RES EXT WALL"
## Calculating CTFs for "IEAD NON-RES ROOF"
## Calculating CTFs for "EXT-SLAB"
....
```

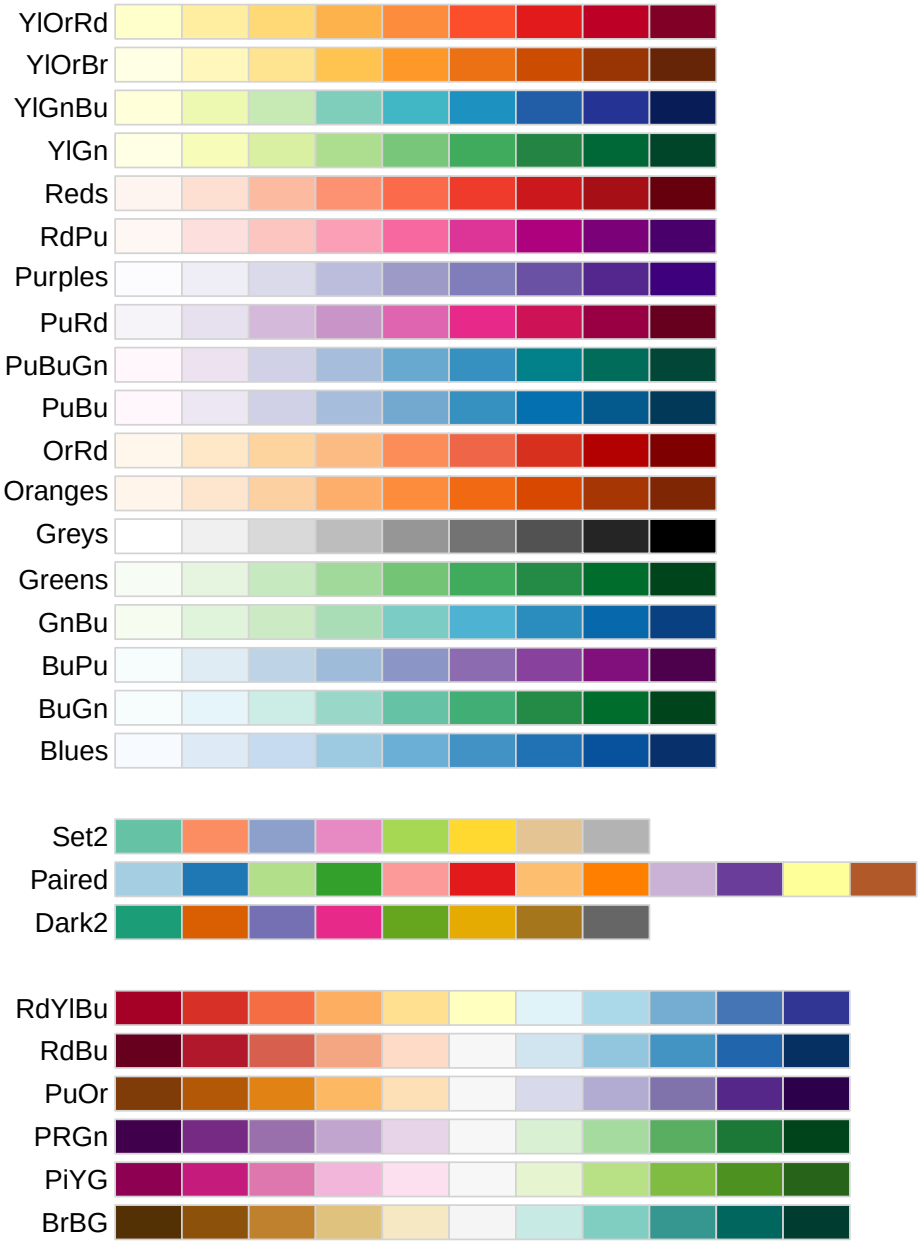
11.2 Colors

Choosing the right color scheme is an important aspect of data visualization because of the effects they might have on our relative visual perception of luminance. To find out more, I recommend reading Chapter 1 of Data Visualization: A practical introduction by Kieran Healy [Healy, 2018] that discusses aspects of perception and interpretation when creating graphics.

Fortunately, R has many pre-defined color palettes where careful thought has been put into their perception and aesthetic qualities. Specifically, the RColorBrewer package contains color palettes that have been carefully designed for discrete data.

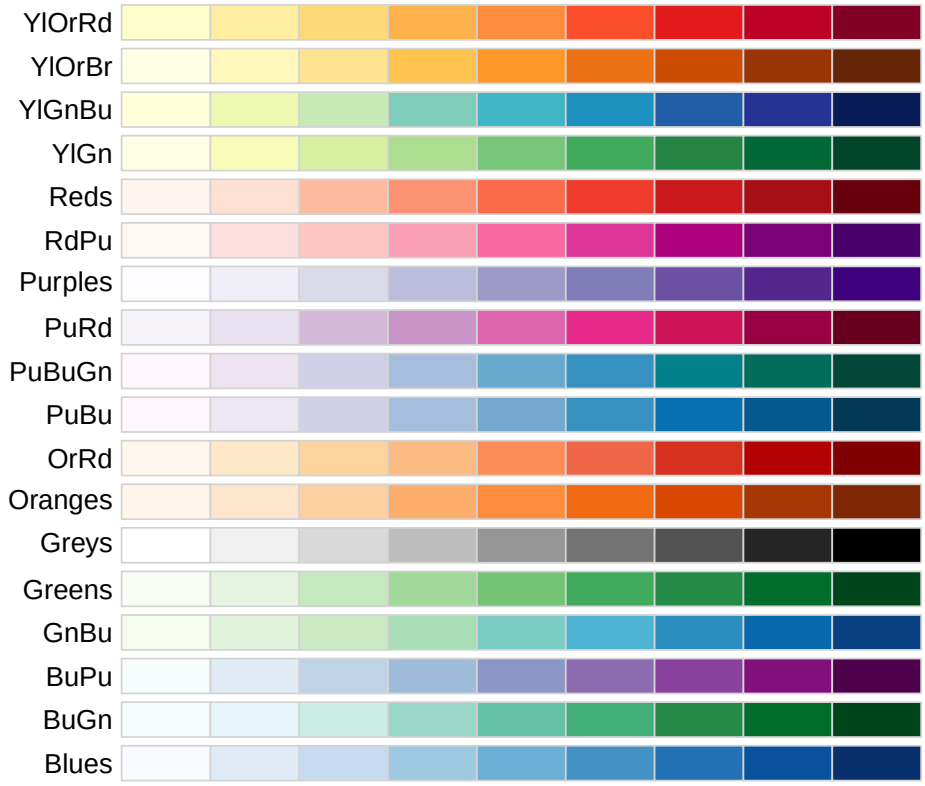
The RColorBrewer package provide three types of color palettes: sequential, diverging, and qualitative. You can view those palettes that are colorblind friendly.

```
display.brewer.all(colorblindFriendly = TRUE)
```



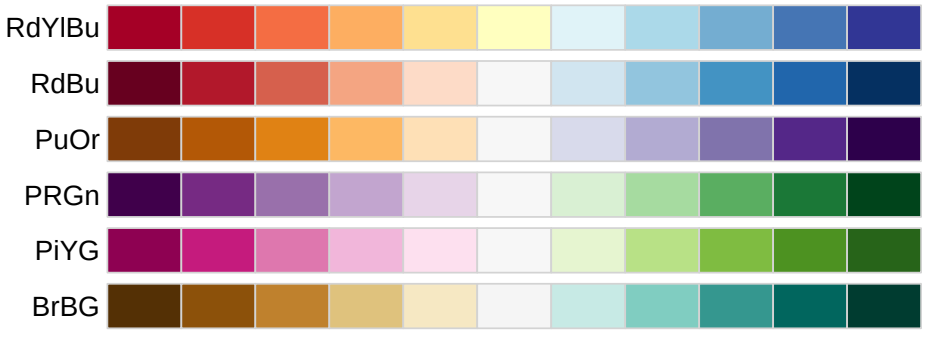
When representing gradients (such as temperature data), you want to use *sequential* color palletes that are perceptually uniform as the color progresses from low to high.

```
display.brewer.all(type="seq", colorblindFriendly=TRUE)
```



You will want to use *diverging* color palettes that uses a neutral mid-point that diverges at perceptually equal steps to both ends of the data. An example is using the `RdBu` color pallette to represent temperature data that diverges between cold and hot.

```
display.brewer.all(type="div", colorblindFriendly=TRUE)
```



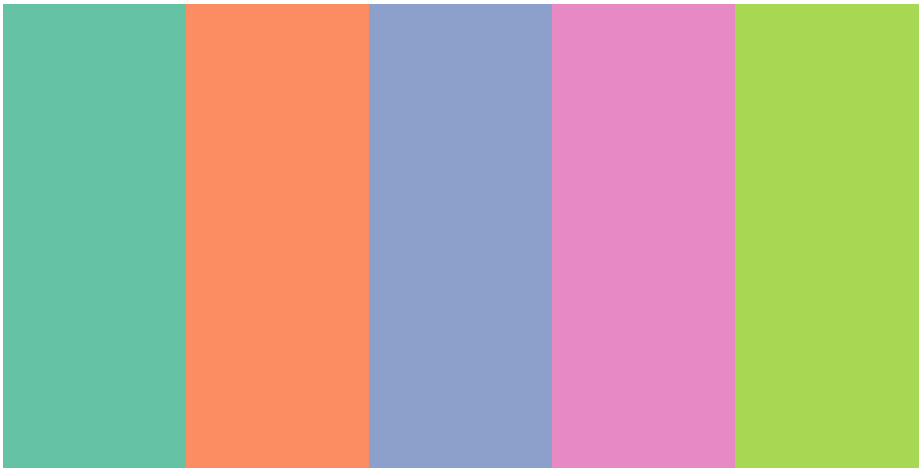
You will want to map categorical variables to *qualitative* color palettes that

are perceptually uniform and not have any variables standing out perceptually due to it's color. For instance, when visualizing the energy usage intensity of different buildings, you would want the colors used to represent each building to be easily distinguishable but at the same time not have one stand out from another even though they are numerically equivalent.

```
display.brewer.all(type="qual", colorblindFriendly=TRUE)
```

You can use the function `display.brewer.pal()` to visualize a single brewer palette and use the `brewer.pal()` function to obtain the hexadecimade color code of the palette.

```
# display 5 colors from the "Set2" color palette
display.brewer.pal(5, "Set2")
```



Set2 (qualitative)

```
# return the hexadecimal code for 5 olors of the "Set2" palette
brewer.pal(5, "Set2")
## [1] "#66C2A5" "#FC8D62" "#8DA0CB" "#E78AC3" "#A6D854"
```

Visualizing building related data such as electricity consumption often involves mapping continuous data onto the color or fill of the graphic you want to create. Although the brewer palettes can be extended to continuous data through

interpololation, I have found the color scales in the viridis package to be more robust because they are designed to be perceptually uniform while maintaining a large range. The viridis package contains eight different color scales with the viridis scale forming the default or primary color map.

11.3 `ggplot()`

11.3.1 Functions and Components

All plots created by ggplot2 begins with a `ggplot()` function that initializes a ggplot plot object that can be used to specify how variables in the data are mapped to the "aesthetics" of the visualization. The function has two key

arguements. The first argument `data` is the data frame that will be used for the plot. The second argument `mapping` is used to specify how variables in the data are mapped to the "aesthetics" of the visualization. The function `aes()` is a quoting function (i.e., the inputs are evaluated in the context of the data). This means that you can name the variables of the data frame directly within the `aes()` function.

```
ggplot(data = <DATA>, mapping = aes(<x, y, ...>))
```

You can then specify the graph by adding one or more of the following components with `+`

- A layer that comprises of geometric objects or *geom*, statistical transformation or *stat*, and *position* adjustments. Typically, a layer will be created using a `geom_<function>()`
- *scale*s that map data values to visual properties such as color, fill, shape, and size.
- A *coord*inate system that specifies how the coordinates of the data maps to the plot. Typically, Cartesian coordinates are used. However, other coordinate systems includes polar coordinates and map projections.
- *facet*s that divides the data into subsets based on one or more discrete variables. These subsets of data are then displayed as subplots on the plot.
- A *theme* that can be used to customize the non-data components of the plot such as titles, labels, fonts, background, gridlines, and legends.

```
ggplot(data = <DATA>, mapping = aes(<x, y, ...>)) +
    <GEOM_FUNCTION>(stat = <STAT>, position = <POSITION>) +
    <COORD_FUNCTION>(...) +
    <SCALE_FUNCTION>(...) +
    <FACET_FUNCTION>(...) +
    <THEME_FUNCTION>(...)
```

You will see the use of `ggplot()` and the above mentioned components more concretely as we go through the visualization recipes in the subsequent sections.

11.3.2 Visualize end use

Bar graphs are a common way to visualize building simulation end use data. In this section, we will illustrate the use of ggplot and its various components using using the `report_end_use` data frame that was created in the preceding section.

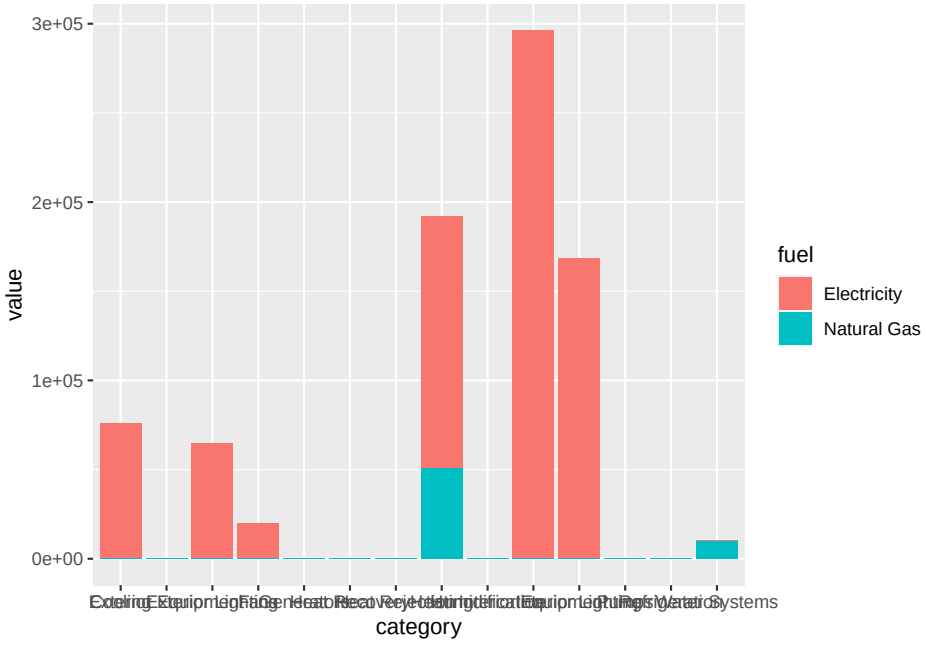
```
report <- job$tabular_data()

report_end_use <- report %>%
    filter(table_name == "End Uses",
           grepl("Electricity|Natural Gas", column_name, ignore.case = TRUE),
```

```
        !grepl("total", row_name, ignore.case = TRUE)) %>%
    mutate(value = as.numeric(value)*277.778,
           units = "kWh") %>%
    select(row_name, column_name, units, value) %>%
    rename(category = row_name, fuel = column_name) %>%
    arrange(desc(value)) %>%
    drop_na()
```

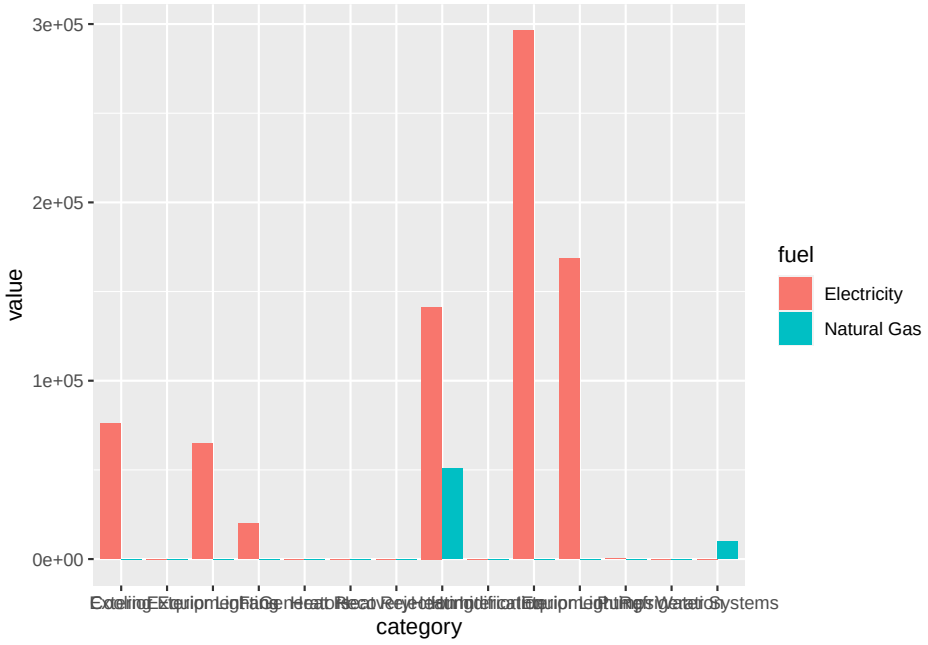
You can create bar graphs by adding `geom_bar()`. By default, `stat = bin` in `geom_bar()` which gives the count in each `x`. However, when the data contains `y` values, you would want to use `stat = "identity"`.

```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity")
```



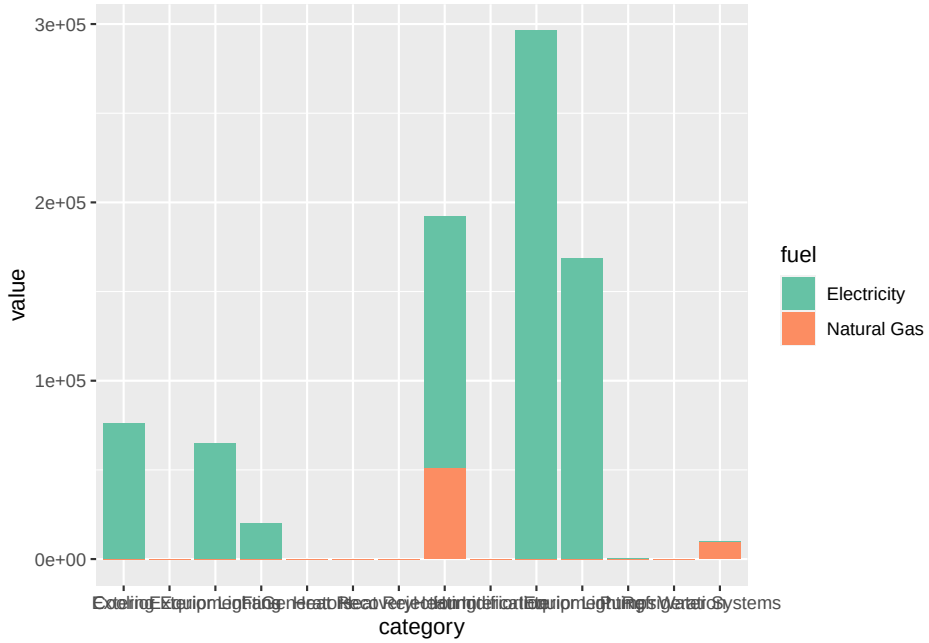
By default, the bar charts would be stacked. In this scenario, we only have two groups of fuel, electricity and natural gas. However, when there are many groups, stacked bar charts can be difficult to visualize. You can place them side by side instead using `position = position_dodge()`.

```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity", position = position_dodge())
```

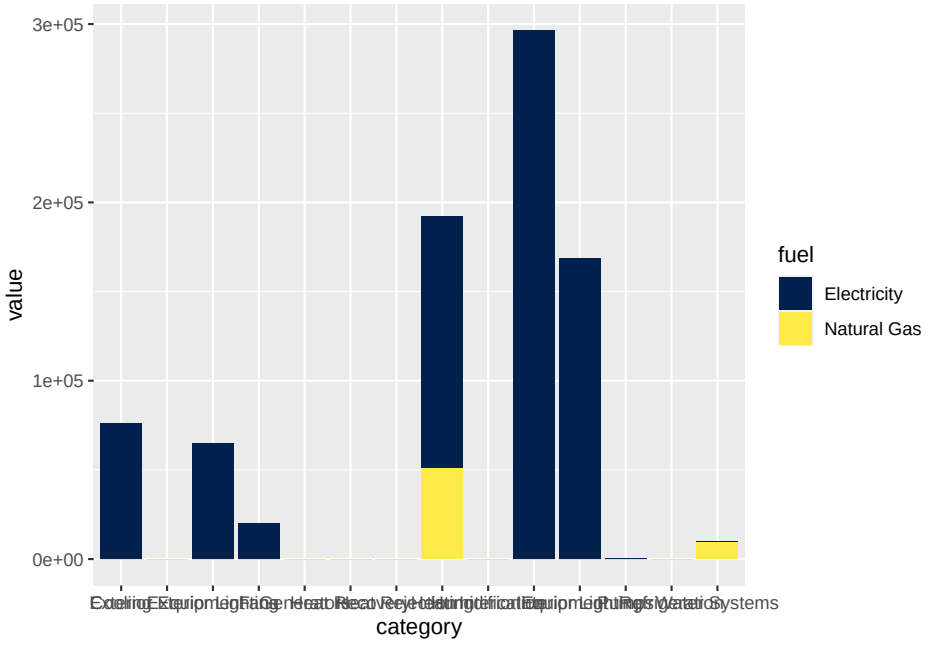


You can use scale to change the fill and colors of the bar chart. The `scale_*_brewer()` and `scale_*_viridis()` functions provides an easy way to specify palettes from the RColorBrewer and the viridis package respectively. Particularly, `scale_fill_brewer()` and `scale_fill_viridis()` provides mapping to ggplot's fill aesthetics while `scale_color_brewer()` and `scale_color_viridis()` provides mapping to ggplot's color aesthetics.

```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity") +
    # use palette argument to indicate the brewer palette to use
    scale_fill_brewer(palette = "Set2")
```

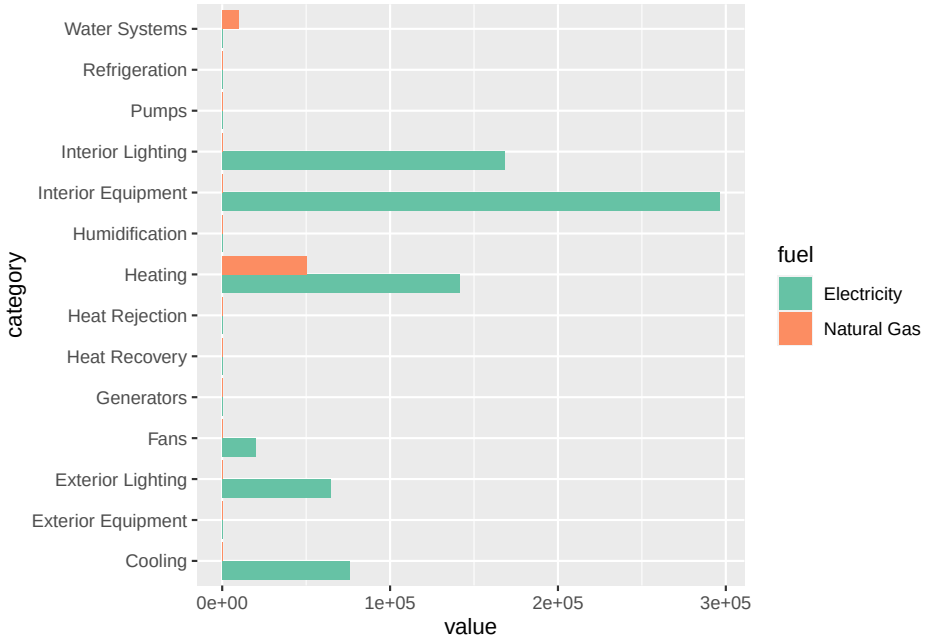



```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity") +
    # use option argument to indicate the color scale to use
    scale_fill_viridis(option = "cividis",
                       discrete = TRUE)
```



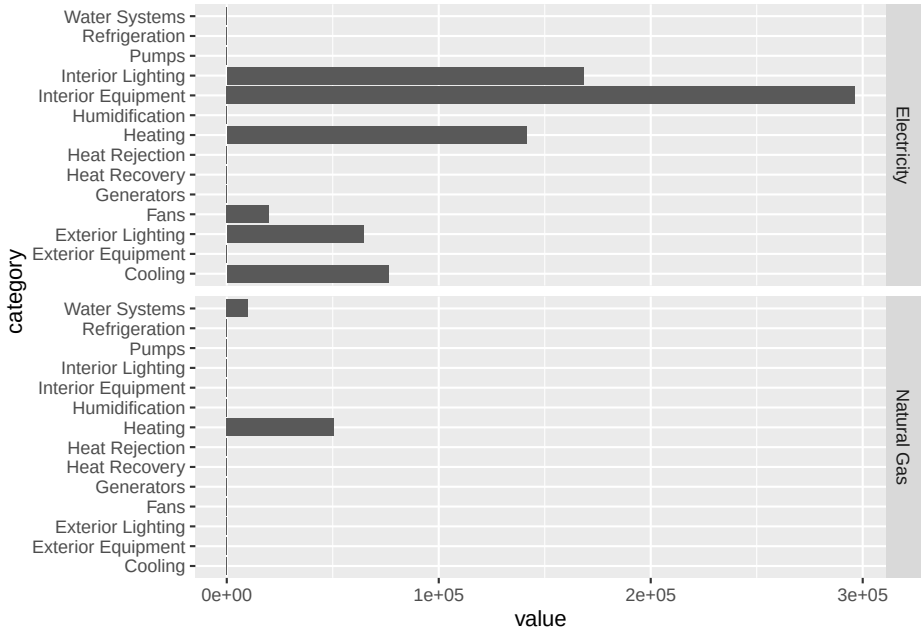
You can flip how the data coordinates maps to plot to get horizontal bar plots with `coord_filp()`.

```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity", position = position_dodge()) +
    scale_fill_brewer(palette = "Set2") +
    coord_flip()
```



You can divide the plot into various *facet*s by subsetting the plot based one or more discrete variables. In this example, we divide the plot row wise based on fuel type using `facet_grid()`.

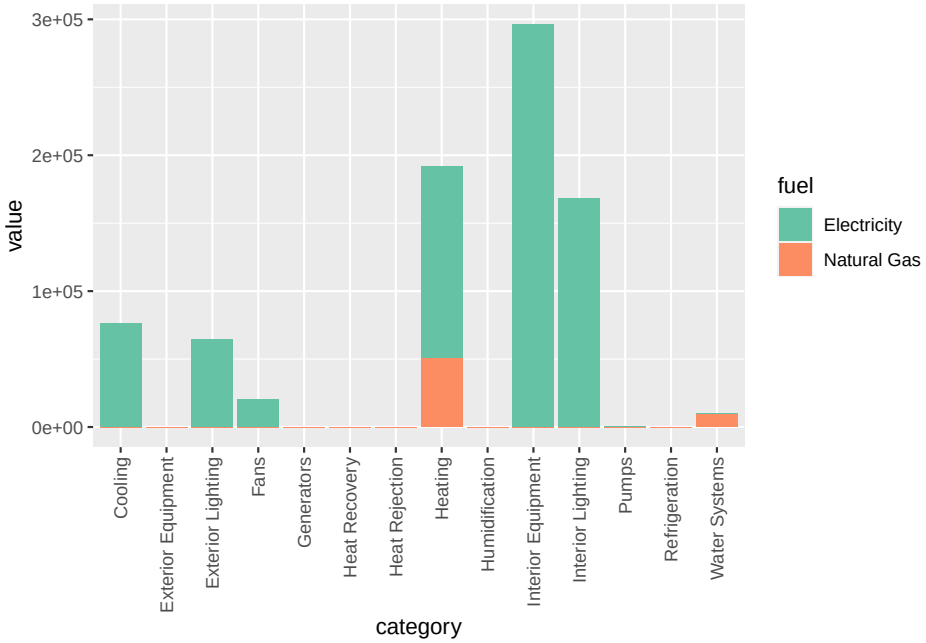
```
ggplot(data = report_end_use, aes(x = category, y = value)) +
    geom_bar(stat="identity", position = position_dodge()) +
    coord_flip() +
    facet_grid(rows = vars(fuel))
```



As you probably have noticed. The x-axis labels are not legible due to overlapping when plotting the data as vertical bar charts.

You can use `theme()` and `element_text()` to change how the x-axis labels appear. In this case we are rotating it counter-clockwise by 90 degrees (`angle = 90`), vertically center justify the text (`vjust = 0.5`), and horizontally right justify the text (`hjust = 1`). For `vjust` and `hjust`, 0 and 1 refers to left and right justify respectively.

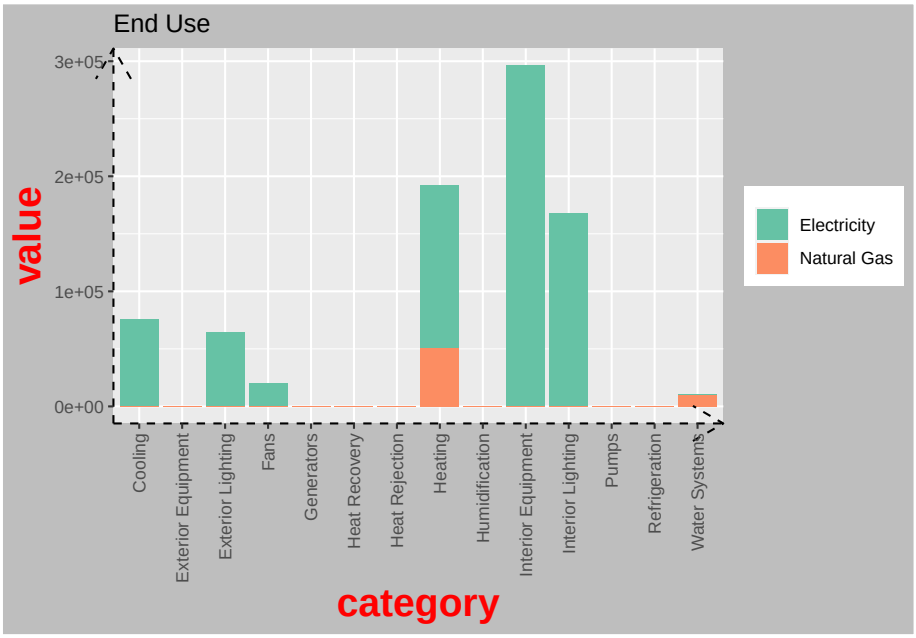
```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity") +
    scale_fill_brewer(palette = "Set2") +
    theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
```



You can also use `theme()` together with various `element_*` functions to control elements of the plot title, legend, axis labels, borders, background, etc. You can find out more about the possible arguments to each `element_function()` by typing `?margin` into your console. `element_*` functions are used with `theme()` to specify the non-data components of the plot. There are four element functions and they are:

- `element_blank()`: to assign a blank
- `element_rect()`: for specifying borders and background
- `element_line()`: for specifying lines
- `element_text()`: for specifying text

```
ggplot(data = report_end_use, aes(x = category, y = value, fill = fuel)) +
    geom_bar(stat="identity") +
    scale_fill_brewer(palette = "Set2") +
    ggtitle("End Use") +
    theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1),
          axis.title = element_text(face = "bold",
                                    colour = "red",
                                    size = 20),
          axis.line = element_line(linetype = "dashed",
                                   arrow = arrow()),
          plot.background = element_rect(fill = "grey"),
          legend.title = element_blank() # remove legend title
          )
```



11.3.3 Visualize weather data

Weather is a critical input to building energy simulation because it form the boundary conditions of the simulation. Weather for an EnergyPlus simulation comes in an **E**nergy**P**lus **W**eather (EPW) format and often contains 8760 hours (or 8784 hours for a leap year) of weather data. Therefore, being able to visualize weather data is an important component when exploring building energy simulation data.

Here, we demonstrate three useful graphics for visualizing weather data using outdoor dry bulb temperature as an example.

Before creating the graphics using ggplot, we need to first extract the weather data, which we can easily carry out using the `$data()` method since we have earlier parsed the EPW file into RStudio as an `Epw` object.

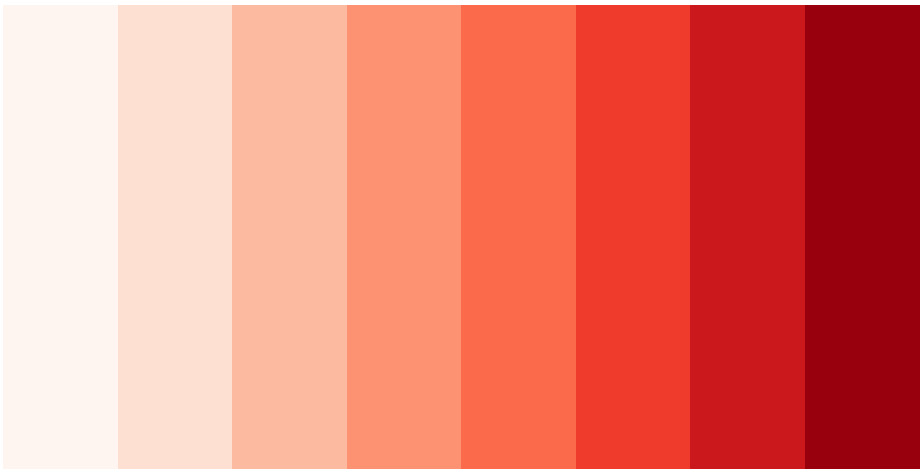
```
class(epw$data())
## [1] "data.table" "data.frame"
head(epw$data())
##              datetime year month day hour minute
## 1: 2017-01-01 01:00:00 1986     1   1    1      0
## 2: 2017-01-01 02:00:00 1986     1   1    2      0
## 3: 2017-01-01 03:00:00 1986     1   1    3      0
## 4: 2017-01-01 04:00:00 1986     1   1    4      0
## 5: 2017-01-01 05:00:00 1986     1   1    5      0
## 6: 2017-01-01 06:00:00 1986     1   1    6      0
```

```
##                                                 data_source dry_bulb_temperature
## 1: ?9?9?9?9E0?9?9?9?9?9?9?9?9?9?9?9?9?9?9?9*_*9*9*9*9*9*9               -12.2
## 2: ?9?9?9?9E0?9?9?9?9?9?9?9?9?9?9?9?9?9?9?9*_*9*9*9*9*9*9               -11.7
....
```

```
weather_data <- epw$data() %>%
    select(datetime, dry_bulb_temperature) %>%
    mutate(month = month(datetime, label = TRUE),
           day = day(datetime),
           wday = wday(datetime, label = TRUE),
           hour = hour(datetime))
```

In the subsequent graphics, we will be using various colors from the `Reds` color palette from RColorBrewer.

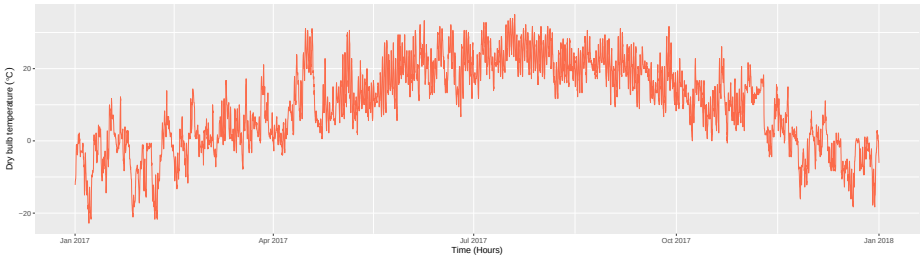
```
# return the hexadecimal code for 5 olors of the "Set2" palette
brewer.pal(8, "Reds")
## [1] "#FFF5F0" "#FEE0D2" "#FCBBA1" "#FC9272" "#FB6A4A" "#EF3B2C" "#CB181D"
## [8] "#99000D"
display.brewer.pal(8, "Reds")
```



Reds (sequential)

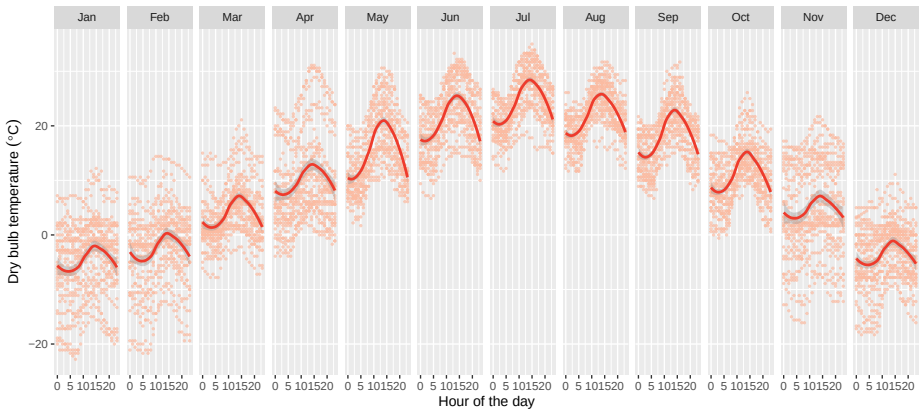
You can visualize the weather data (dry bulb temperature in this example) as a single time series plot. Here, `geom_line()` is used to connect the observations sequentially over time. `expression()` is used to express the degree symbol in the y-axis label.

```
ggplot(weather_data, aes(x = datetime, y = dry_bulb_temperature)) +
    geom_line(color = "#FB6A4A") +
    xlab("Time (Hours)") +
    ylab(expression("Dry bulb temperature " ( degree*C)))
```



You can also visualize the data by the hour of the day using scatterplots. To avoid overplotting, a typical problem with scatterplots, you can subset the data by their month using `facet_grid()`. To further aid the ability to visually identify patterns in the presence of overplotting, you can use `geom_smooth()` to add a smoothed line on top of the scatterplot.

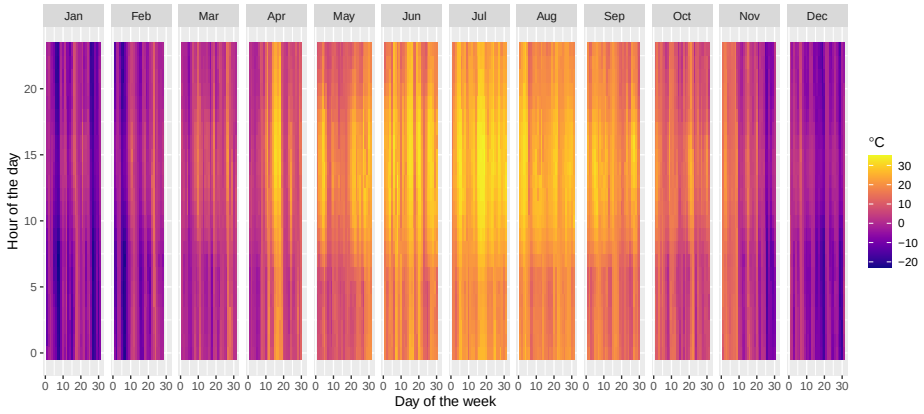
```
ggplot(weather_data, aes(x= hour, y = dry_bulb_temperature)) +
    geom_point(color = "#FCBBA1", alpha = 0.7, size = 0.5) +
    geom_smooth(color = "#EF3B2C") +
    facet_grid(cols = vars(month)) +
    xlab("Hour of the day") +
    ylab(expression("Dry bulb temperature " ( degree*C)))
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Last but not least, you can create heatmaps using the function `geom_tile()`.

```
ggplot(weather_data, aes(x = day, y = hour, fill = dry_bulb_temperature)) +
    geom_tile() +
    scale_fill_viridis(name = expression(degree*C),
                       option = "plasma") +
```

```
    facet_grid(cols = vars(month)) +
    ylab("Hour of the day") +
    xlab("Day of the week")
```



11.3.4 Saving Plots

You can use the `ggsave()` function to save the plot. By default, it saves the last plot that was displayed. You can specify the size of the graphic using the units ("in", "cm", "mm", or "px"), width and height argument.

```
ggsave("my_plot.pdf", width = 16, height = 24, units = "cm")
ggsave("my_plot.png", width = 6, height = 9, units = "in")
```

Part III

Inputs and Outputs

Chapter 12

Introduction

In this part of the book, you will learn EnergyPlus's input data structure (Chapter 13), and how to programmatically query and modify them in R (Chapter 14). We will also point you to the corresponding EnergyPlus reference documentation that will help you gain a better understanding of the inner workings of EnergyPlus. Understanding the inputs to a model is important as the proverb "garbage in, garbage out" clearly spells out. You can have the best data science workflows but your simulation results would only be as good the quality of your model and it's inputs.

Subsequently, you will learn how to extract the more detailed output files (Chapter 15. You will work with time-series simulation results and explore them using R's data transformation and data visualization capabilities (Chapter 16).

Chapter 13

Model Input Structure

13.1 Prerequisites

In this chapter, we will use the **eplusr** package to extract various inputs in an EnergyPlus model, and through this, learn how the inputs in EnergyPlus are structured.

```
library(eplusr)
library(here)
```

We will be working with the IDF and EPW file that pertains to the U.S. Department of Energy (DOE) Commercial Reference Building and Chicago's TMY3 respectively.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)

path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)
```

13.2 EnergyPlus Documentation

Although the focus of this chapter is to elucidate how the model inputs are structured to facilitate querying and modifying them, it is also important to understand what each input means. Fortunately, EnergyPlus is well documented and these documentation comes downloaded with any installation of EnergyPlus. You will find these documents within EnergyPlus's installation folder.By default, EnergyPlus will be installed in `C:\EnergyPlusVX-Y-0` on Windows, `/usr/local/EnergyPlus-X-Y-0` on Linux and `/Applications/EnergyPlus-X-Y-0` on macOS).

I would highly recommend referring to the EnergyPlus input output reference (`[EnergyPlus installation folder] > Documentation > InputOutputReference.pdf`), which provides a thorough description of every field for every input object in EnergyPlus.

For those who find that the input output reference does not satisfy your inquisitive mind, I would recommend reading the EnergyPlus engineering reference (`[EnergyPlus installation folder] > Documentation > EngineeringReference.pdf`), which provides insights into the theoretical basis behind various EnergyPlus calculations.

13.3 EnergyPlus input structure

In EnergyPlus, the inputs to the model can be categorized hierarchically where the top level is known as the class groups (Figure 13.1). Each class group consists of multiple input classes that are identifiable by their class names. Each class is then defined by fields that may or may not be required. Required fields as it's name suggest are field that must have a value. If the field is left blank and there are no pre-defined default values, an error will be raised when the simulation is run.

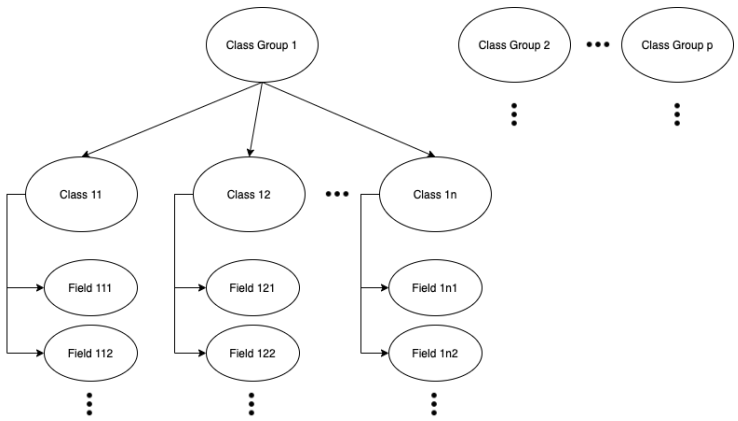


Figure 13.1: Hierarchical model inputs organizational structure.

Figure 13.2 illustrates this using the class group `Surface Construction Elements` in EnergyPlus. The `Surface Construction Elements` class groups consists of several classes that include the `Material:NoMass`, `Material:AirGap`, and the `WindowMaterial:SimpleGalzingSystem` class amongst others. Each class is then defined by their respective fields. For example, the `WindowMaterial:SimpleGalzingSystem` class is defined by a unique `Name`, the `U-Factor`, `SHGC` or Solar Heat gain Coefficient, and the 'Visible Transmittance of the glazing system.

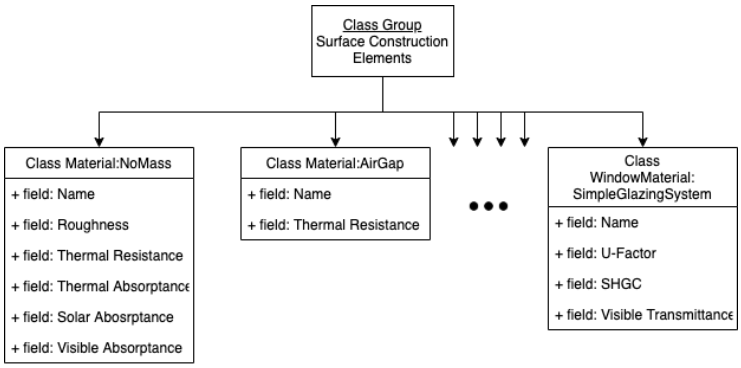


Figure 13.2: Categorization of EnergyPlus inputs

13.4 Class vs Object

In this book, we make a distinction between classes and objects following the terminologies of object-oriented programming (OOP). The difference between classes and objects is subtle yet conceptually simple. You can think of a class as a blueprint or template for creating objects. The created objects are therefore instances of the class. For example, when modeling glazing systems from the perspective of running an energy simulation, the thermal properties of the glazing system would be required for the calculations. Different glazing systems would have different thermal properties, which include the U-factor, solar heat gain coefficient (SHGC), and visible transmittance (T_{vis}). However, these properties although having different values, are a common characteristic across different glazing systems. Therefore, a glazing system class with the variables U-factor, SHGC, and T_{vis} can be used as a template to create various glazing objects.

13.5 Model Query

13.5.1 `Idf` class methods

Since the EnergyPlus model is organized hierarchically, we will demonstrate how to query the model hierarchically using the methods in the `Idf` class.

`Idf` Class Method	What it does
`$group_name()`	Query the names of class groups
`$class_name()`	Query the names of classes
`$object_name()`	Query the names of objects in one or more classes
`$objects()`	Query one or more objects using their names

You can see what class groups exists in your model with the `$group_name()` method

```
model$group_name()
##  [1] "Simulation Parameters"              "Location and Climate"
##  [3] "Schedules"                          "Surface Construction Elements"
##  [5] "Thermal Zones and Surfaces"         "Internal Gains"
##  [7] "Zone Airflow"                       "Exterior Equipment"
....
```

You can also view the list of classes in the model grouped by their corresponding class group

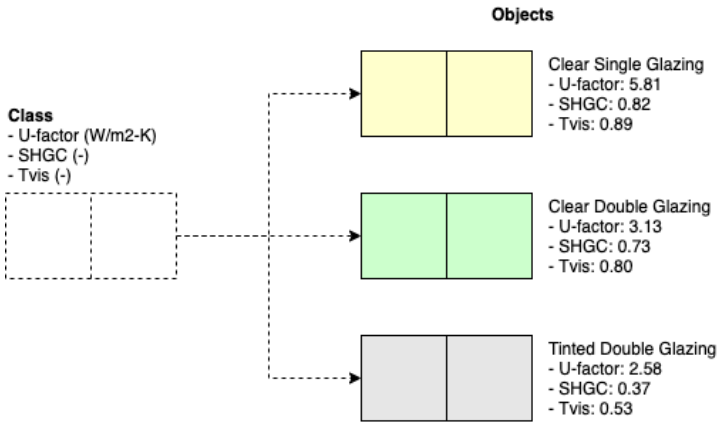


Figure 13.3: Figure shows how three different glazing objects can be created from a glazing class, which acts as the template.

```
##  [3] "Building"                          "ShadowCalculation"
##  [5] "SurfaceConvectionAlgorithm:Inside"  "SurfaceConvectionAlgorithm:Outside"
##  [7] "HeatBalanceAlgorithm"              "ZoneAirHeatBalanceAlgorithm"
##  [9] "Timestep"                          "ConvergenceLimits"
##
## $`Location and Climate`
## [1] "Site:Location"
## [2] "SizingPeriod:DesignDay"
....
```

You can also view the list of objects that have been defined in a particular class in the model using the `$object_name()` method. The example from the code chunk below tells us that there are currently two materials (`CP02 CARPET PAD` and `MAT-AIR-WALL`) that have been created using the `Material:NoMass` class. You can think of these classes as templates for creating objects that defines your model as explained in the previous sub-section. In this particular example, the two material objects `CP02 CARPET PAD` and `MAT-AIR-WALL` are instances in the model that have been defined based on the `Material:NoMass` class.

```
model$object_name("Material:NoMass")
## $`Material:NoMass`
## [1] "CP02 CARPET PAD" "MAT-AIR-WALL"
```

You can also query multiple classes.

```
model$object_name(c("Material", "Construction"))
## $Material
##  [1] "Steel Frame NonRes Wall Insulation" "IEAD NonRes Roof Insulation"
```

```
##  [3] "Std Wood 6inch"                      "Wood Siding"
##  [5] "1/2IN Gypsum"                        "1IN Stucco"
##  [7] "8IN CONCRETE HW"                     "Metal Siding"
##  [9] "HW CONCRETE"                         "Roof Membrane"
## [11] "Metal Decking"                       "Metal Roofing"
## [13] "MAT-CC05 4 HW CONCRETE"              "Std ACO2"
##
## $Construction
....
```

Finally, you can also query individual objects using the `$objects()` method.

```
model$objects(c("Steel Frame Non-res Ext Wall", "IEAD Non-res Roof"))
## $`Steel Frame Non-res Ext Wall`
## <IdfObject: 'Construction'> [ID:37] `Steel Frame Non-res Ext Wall`
## -- COMMENTS ----------------------------------------------------------------
## ! ***OPAQUE CONSTRUCTIONS AND MATERIALS***
## ! Exterior Walls
## -- VALUES ------------------------------------------------------------------
## Class: <Construction>
##   1*: "Steel Frame Non-res Ext Wall",  !- Name
##   2*: "Wood Siding",      !- Outside Layer
##   3 : "Steel Frame NonRes Wall Insulation",  !- Layer 2
....
```

13.5.2 $ operator

Since the model is stored as a named list in R, you can use the `$` operator to also query elements by name. For instance, you can access the objects defined using the `Material:NoMass` class with `$Material:NoMass`. Since class, object, and field names in an EnergyPlus often violets the variable naming restrictions in R (e.g., containing special characters such as `:`, white spaces, etc.), the identifiers need to be placed within a set of single back quotes `` ` `` to tell R that this is a variable name.

With `` $`Material:NoMass` `` will list all the objects in the `Material:NoMass` class.

```
model$`Material:NoMass`
## $`CP02 CARPET PAD`
## <IdfObject: 'Material:NoMass'> [ID:56] `CP02 CARPET PAD`
## Class: <Material:NoMass>
##   1*: "CP02 CARPET PAD", !- Name
##   2*: "VeryRough",       !- Roughness
##   3*: 0.2165,            !- Thermal Resistance {m2-K/W}
##   4 : 0.9,               !- Thermal Absorptance
##   5 : 0.7,               !- Solar Absorptance
```

```
##   6 : 0.8;                !- Visible Absorptance
##
....
```

Since the model is stored hierarchically, you can chain them by adding another `$` to access and modify lower levels. Using the same example, you will be able to retrieve the `CP02 CARPET PAD` object with `$`Material:NoMass`$`CP02 CARPET PAD``.

```
model$`Material:NoMass`$`CP02 CARPET PAD`
## <IdfObject: 'Material:NoMass'> [ID:56] `CP02 CARPET PAD`
## Class: <Material:NoMass>
##   1*: "CP02 CARPET PAD", !- Name
##   2*: "VeryRough",       !- Roughness
##   3*: 0.2165,            !- Thermal Resistance {m2-K/W}
##   4 : 0.9,               !- Thermal Absorptance
##   5 : 0.7,               !- Solar Absorptance
##   6 : 0.8;               !- Visible Absorptance
```

Likewise, you could go further down the hierarchy and extract the value of a particular field of an object by further chaining with another `$`. The example below shows that the `Thermal Resistance` value of the `CP02 CARPET PAD` object that was defined using the `Material:NoMass` class is `0.2165`.

```
model$`Material:NoMass`$`CP02 CARPET PAD`$`Thermal Resistance`
## [1] 0.2165
```

You can use the `definition` method that returns the fields needed as inputs to define a particular class Now, compare the output from the code chunk below with Figure 13.2. The required fields are indicated by `eplusr` using an asterix `*`. Using the `Material:NoMass` class as an example (Figure 13.4), the same information can be obtained from the IDD file (`Energy+.idd`) where the required fields are indicated with the `\required-field` tag. The definitions of the different tags can be found at the begining of the IDD file.

```
model$definition("Material:NoMass")
## <IddObject: 'Material:NoMass'>
## -- MEMO ------------------------------------------------------------------
##   "Regular materials properties described whose principal description is R (Thermal
##
## -- PROPERTIES ------------------------------------------------------------
##   * Group: 'Surface Construction Elements'
##   * Unique: FALSE
##   * Required: FALSE
##   * Total fields: 6
##
## -- FIELDS ----------------------------------------------------------------
```

```
##   1*: Name
##   2*: Roughness
##   3*: Thermal Resistance
##   4 : Thermal Absorptance
##   5 : Solar Absorptance
##   6 : Visible Absorptance
```

13.6 Object interdependencies

The inter-dependencies between different objects is a challenging aspect when working with energy models. It is important to note that changing an object not only affects the object but also the objects that reference it. A classic example is the relationship between the building's thermal zones, surfaces, construction, and materials. A single building energy model comprises of one or more thermal zones and each thermal zone has it's boundaries defined by several surfaces. Each surface is then assigned one construction where each layer of the construction is a material that is defined based on it's thermal properties.

Using the construction `Steel Frame Non-res Ext Wall` as an example. You can see that it references the materials `Wood Siding`, `Steel Frame NonRes Wall Insulation`, and `1/2IN Gypsum`. Likewise, it is referenced by the surfaces `Steel Frame Non-res Ext Wall`, `Perimeter_bot_Plenum_Wall_North`, ..., `Perimeter_top_ZN_4_Wall_West`. Therefore, changing the thermal properties of either of the three materials `Wood Siding`, `Steel Frame NonRes Wall Insulation`, and `1/2IN Gypsum` would change the thermal properties of the `Steel Frame Non-res Ext Wall` Construction and subsequently all the surfaces that references it. On a different note, changing an object's name would also require updating all the fields in the model that references the object. In this example, changing `Steel Frame Non-res Ext Wall` to `Steel Frame Non-res Wall` would require updating all the `Construction Name` field in the surfaces that references it, otherwise, a severe error would be raised when the simulation is run.

```
model$object_relation("Steel Frame Non-res Ext Wall")
## -- Refer to Others ------------------------------------------------------------
##  Class: <Construction>
##    Object [ID:37] <Steel Frame Non-res Ext Wall>
##       2: "Wood Siding", !- Outside Layer
##        v~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
##          Class: <Material>
##             Object [ID:50] <Wood Siding>
##                1: "Wood Siding"; !- Name
##
##       3: "Steel Frame NonRes Wall Insulation",  !- Layer 2
....
```

```
Material:NoMass,
    \memo Regular materials properties described whose principal des
     \min-fields 3
  A1 , \field Name
       \required-field
       \type alpha
       \reference MaterialName
  A2 , \field Roughness
       \required-field
       \type choice
       \key VeryRough
       \key Rough
       \key MediumRough
       \key MediumSmooth
       \key Smooth
       \key VerySmooth
  N1 , \field Thermal Resistance
       \required-field
       \units m2-K/W
       \type real
       \minimum .001
  N2 , \field Thermal Absorptance
       \type real
       \minimum> 0
       \default .9
       \maximum 0.99999
  N3 , \field Solar Absorptance
       \type real
       \minimum 0
       \default .7
       \maximum 1
  N4 ; \field Visible Absorptance
       \type real
       \minimum 0
       \default .7
       \maximum 1
```

Figure 13.4: Screen capture of the definition of the Material:NoMass class from the EnergyPlus IDD file (‘Energy+.idd‘)

In EnergyPlus, each object may be referenced by other

```
model$object_relation("Window Non-res Fixed")
## -- Refer to Others ----------------------------------------------------------
##  Class: <Construction>
##    Object [ID:45] <Window Non-res Fixed>
##        2: "NonRes Fixed Assembly Window";  !- Outside Layer
##         v~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
##           Class: <WindowMaterial:SimpleGlazingSystem>
##              Object [ID:46] <NonRes Fixed Assembly Window>
##                 1: "NonRes Fixed Assembly Window";  !- Name
##
##
....
```

Chapter 14

Modify model inputs

In this chapter, you will learn how modify the model inputs.

14.1 Prerequisites

In this chapter, we will use the **eplusr** package to query and modify various inputs in an EnergyPlus model. We will also combine what you have learned about data manipulation (with the **tidyverse** package) to make it easier to work with the simulation inputs.

```
library(eplusr)
library(tidyverse)
library(here)
```

We will be working with the IDF and EPW file that pertains to the U.S. Department of Energy (DOE) Commercial Reference Building and Chicago's TMY3 respectively.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)

path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)
```

14.2 Extract and Modify

Changing the values of the existing objects in the model is an important functionality in the application of data exploration and data science to EnergyPlus. There are many approaches to extract and modify an EnergyPlus model using eplusr. In this chapter we focus on

- Directly extracting and modifying the field values of a single object using the `[[` or `$` operator.
- Extracting and modifying multiple objects using the methods in the `Idf` class. Type `?Idf` in your console to see the `Idf` class methods.

14.2.1 Single object

Like how you can access the field values of the model using the `$` operator, you can likewise modify them simply by accessing and then replace the current value by assigning a different value. For instance, you can change the `Thermal Resistance` field value of the `CP02 CARPET PAD` object to `0.5` using the `<-` operator to assign `0.5` to the field `Thermal Resistance`.

```
model$`Material:NoMass`$`CP02 CARPET PAD`$`Thermal Resistance` <- 0.5
model$`Material:NoMass`$`CP02 CARPET PAD`
## <IdfObject: 'Material:NoMass'> [ID:56] `CP02 CARPET PAD`
## Class: <Material:NoMass>
##   1*: "CP02 CARPET PAD", !- Name
##   2*: "VeryRough",       !- Roughness
##   3*: 0.5,               !- Thermal Resistance {m2-K/W}
##   4 : 0.9,               !- Thermal Absorptance
##   5 : 0.7,               !- Solar Absorptance
##   6 : 0.8;               !- Visible Absorptance
```

Another example showing how you can modify the lighting power density (field name: `Watts_per_Zone_Floor_Area`) of a particular zone (object name: Perimeter_bot_ZN_1_Lights) from `10.76` (current value) to `15` (new value).

Existing field values of the object `Perimeter_bot_ZN_1_Lights`, an instance of the `Lights` class.

```
model$Lights$Perimeter_bot_ZN_1_Lights
## <IdfObject: 'Lights'> [ID:240] `Perimeter_bot_ZN_1_Lights`
## Class: <Lights>
##   01*: "Perimeter_bot_ZN_1_Lights",  !- Name
##   02*: "Perimeter_bot_ZN_1",  !- Zone or ZoneList Name
##   03*: "BLDG_LIGHT_SCH",   !- Schedule Name
##   04 : "Watts/Area",       !- Design Level Calculation Method
##   05 : <Blank>,            !- Lighting Level {W}
##   06 : 10.76,              !- Watts per Zone Floor Area {W/m2}
##   07 : <Blank>,            !- Watts per Person {W/person}
##   08 : 0.4,                !- Return Air Fraction
....
```

Changing the value of the field `Watts_per_Zone_Floor_Area` from it's existing value of `10.76` to `15`.

```
model$Lights$Perimeter_bot_ZN_1_Lights$Watts_per_Zone_Floor_Area <- 15
model$Lights$Perimeter_bot_ZN_1_Lights
## <IdfObject: 'Lights'> [ID:240] `Perimeter_bot_ZN_1_Lights`
## Class: <Lights>
##   01*: "Perimeter_bot_ZN_1_Lights",  !- Name
##   02*: "Perimeter_bot_ZN_1",  !- Zone or ZoneList Name
##   03*: "BLDG_LIGHT_SCH",   !- Schedule Name
##   04 : "Watts/Area",       !- Design Level Calculation Method
##   05 : <Blank>,            !- Lighting Level {W}
##   06 : 15,                 !- Watts per Zone Floor Area {W/m2}
##   07 : <Blank>,            !- Watts per Person {W/person}
##   08 : 0.4,                !- Return Air Fraction
....
```

14.2.2 Multiple objects

Methods to extract data:

Methods	Description
`$to_table()`	Extract specified objects of the model to data.frames
`$to_string()`	Extract specified objects of the model to strings

You can extract the EnergyPlus objects of interest into a R data frame using the `$to_table` method in the `Idf` class. If no arguments are provided to the method, the entire EnergyPlus model is converted to a data frame.

The data frame consists of the following six columns.

- id (integer): Integer identifiers of the EnergyPlus objects.
- name (character): Names of the EnergyPlus objects.
- class (character): The class that the EnergyPlus object belongs to.
- index (integer): Row index for the field of a particular EnergyPlus object.
- field (character): The name of the column within a table that the observation belongs to.
- value (character): The value of the field.

Take a minute now to open the EnergyPlus IDF file (`"RefBldgMediumOfficeNew2004_Chicago.idf"`). Compare the text file with the data frame below, and identify where the values for the columns `class`, `name`, and `field` would correspond to in the IDF text file. You will also notice that some of the EnergyPlus classes such as `Version` and `SimulationControl` do not have a `name` field, which explains why it is stored as `NA` in the data frame.

```
model$to_table()
##        id name             class index
##    1:   1 <NA>           Version     1
```

```
##    2:   2 <NA> SimulationControl     1
##    3:   2 <NA> SimulationControl     2
##    4:   2 <NA> SimulationControl     3
##    5:   2 <NA> SimulationControl     4
##   ---
## 7839: 681 <NA>       FuelFactors    32
## 7840: 681 <NA>       FuelFactors    33
## 7841: 681 <NA>       FuelFactors    34
....
```

- Subset by Class

Working with the entire model can be difficult due to the large number of rows. You can extract all the objects belonging to a class by supplying a character vector of class names that are of interest to the argument `class`.

```
model$to_table(class = c("Construction", "Lights"))
##       id                          name         class index
##   1:  37 Steel Frame Non-res Ext Wall Construction     1
##   2:  37 Steel Frame Non-res Ext Wall Construction     2
##   3:  37 Steel Frame Non-res Ext Wall Construction     3
##   4:  37 Steel Frame Non-res Ext Wall Construction     4
##   5:  39            IEAD Non-res Roof Construction     1
##  ---
## 217: 251    Perimeter_top_ZN_4_Lights       Lights     9
## 218: 251    Perimeter_top_ZN_4_Lights       Lights    10
## 219: 251    Perimeter_top_ZN_4_Lights       Lights    11
....
```

You can also extract particular objects from different classes by supplying a character vector of object names to the argument `which`.

```
model$to_table(which = c("Wood Siding",
                         "Steel Frame NonRes Wall Insulation",
                         "1/2IN Gypsum"))
##     id                                name    class index              field
##  1: 50                       Wood Siding Material     1               Name
##  2: 50                       Wood Siding Material     2          Roughness
##  3: 50                       Wood Siding Material     3          Thickness
##  4: 50                       Wood Siding Material     4       Conductivity
##  5: 50                       Wood Siding Material     5            Density
##  6: 50                       Wood Siding Material     6      Specific Heat
##  7: 50                       Wood Siding Material     7 Thermal Absorptance
##  8: 50                       Wood Siding Material     8   Solar Absorptance
##  9: 50                       Wood Siding Material     9 Visible Absorptance
....
```

```
model$to_table(which = c("Perimeter_bot_ZN_1_Lights"))
##      id                        name  class index
##  1: 240 Perimeter_bot_ZN_1_Lights Lights     1
##  2: 240 Perimeter_bot_ZN_1_Lights Lights     2
##  3: 240 Perimeter_bot_ZN_1_Lights Lights     3
##  4: 240 Perimeter_bot_ZN_1_Lights Lights     4
##  5: 240 Perimeter_bot_ZN_1_Lights Lights     5
##  6: 240 Perimeter_bot_ZN_1_Lights Lights     6
##  7: 240 Perimeter_bot_ZN_1_Lights Lights     7
##  8: 240 Perimeter_bot_ZN_1_Lights Lights     8
##  9: 240 Perimeter_bot_ZN_1_Lights Lights     9
....
```

Once the EnergyPlus objects are in a `data.frame` format, you can take advantage of the `tidyverse` library to easily manipulate and modify their values. For example, you can use the `filter()` function to subset the data frame to retain rows of interest and the `dplyr::mutate()` function to modify values within columns.

In the example below, we first extract all objects in the `Lights` class from the EnergyPlus model. We then filter the data frame to contain only the field `"Watts per Zone Floor Area"`, which is used to specify the lighting power density in W/m^2 in EnergyPlus. Lastly, we change the lighting power density of all the objects in the `Lights` class from $10.76W/m^2$ to $15W/m^2$.

```
lights_df <- model$to_table(class = c("Lights")) %>%
  mutate(value = if_else(field == "Watts per Zone Floor Area", # condition
                         "15", # value if TRUE
                         value) # value if FALSE
         )


lights_df
##       id                          name  class index
##   1: 237         Core_bottom_Lights Lights     1
##   2: 237         Core_bottom_Lights Lights     2
##   3: 237         Core_bottom_Lights Lights     3
##   4: 237         Core_bottom_Lights Lights     4
##   5: 237         Core_bottom_Lights Lights     5
##  ---
## 191: 251 Perimeter_top_ZN_4_Lights Lights     9
## 192: 251 Perimeter_top_ZN_4_Lights Lights    10
## 193: 251 Perimeter_top_ZN_4_Lights Lights    11
....
```

However, notice that you are not modifying the model just by changing the values of the data frame. This is not surprising since the extracted data frame

is a copy of the model and not the model itself.

```
model$`Lights`
## $Core_bottom_Lights
## <IdfObject: 'Lights'> [ID:237] `Core_bottom_Lights`
## -- COMMENTS ---------------------------------------------------------------
## ! ***LIGHTS***
## -- VALUES -----------------------------------------------------------------
## Class: <Lights>
##   01*: "Core_bottom_Lights",  !- Name
##   02*: "Core_bottom",         !- Zone or ZoneList Name
##   03*: "BLDG_LIGHT_SCH",      !- Schedule Name
##   04 : "Watts/Area",          !- Design Level Calculation Method
....
```

To modify the model, you need to use the `$update()` method in the `Idf` class, which can take the modified data frame as an argument.

```
model$update(lights_df)
## $Core_bottom_Lights
## <IdfObject: 'Lights'> [ID:237] `Core_bottom_Lights`
## -- COMMENTS ---------------------------------------------------------------
## ! ***LIGHTS***
## -- VALUES -----------------------------------------------------------------
## Class: <Lights>
##   01*: "Core_bottom_Lights",  !- Name
##   02*: "Core_bottom",         !- Zone or ZoneList Name
##   03*: "BLDG_LIGHT_SCH",      !- Schedule Name
##   04 : "Watts/Area",          !- Design Level Calculation Method
....
model$`Lights`
## $Core_bottom_Lights
## <IdfObject: 'Lights'> [ID:237] `Core_bottom_Lights`
## -- COMMENTS ---------------------------------------------------------------
## ! ***LIGHTS***
## -- VALUES -----------------------------------------------------------------
## Class: <Lights>
##   01*: "Core_bottom_Lights",  !- Name
##   02*: "Core_bottom",         !- Zone or ZoneList Name
##   03*: "BLDG_LIGHT_SCH",      !- Schedule Name
##   04 : "Watts/Area",          !- Design Level Calculation Method
....
```

- Subset by Objects

You can also extract data from the EnergyPlus model based on the object names by supplying the object names to the argument `which` of the `$to_table()` method. In the example that follows, we will show you how to do this by

changing the thermal properties of the external wall construction.

First, you can find out which `Construction` objects make up the external wall by scanning objects from the `BuildingSurface:Detailed` class, which describes all the surfaces of the model. In general, the external wall surfaces would be those that are of `Surface Type == Wall` and `Outside Boundary Conditions == Outdoors` (i.e., the surface is exposed to outside temperature conditions). In this model, the external wall is made up of the `Steel Frame Non-res Ext Wall` object of the `Construction` class.

```
ext_wall_surf <- model$to_table(class = c("BuildingSurface:Detailed"), wide =TRUE) %>%
  filter(`Surface Type` == "Wall", `Outside Boundary Condition` == "Outdoors")

unique(ext_wall_surf$`Construction Name`)
## [1] "Steel Frame Non-res Ext Wall"
```

The object `Steel Frame Non-res Ext Wall` comprises of three materials (`Wood Siding`, `Steel Frame NonRes Wall Insulation`, and `1/2IN Gypsum`) that are modeled using the class `Material`. To modify the thermal properties of the external wall, you would need to modify the inputs of either of these three materials.

```
model$object_relation("Steel Frame Non-res Ext Wall")
## -- Refer to Others ----------------------------------------------------------
##  Class: <Construction>
##    Object [ID:37] <Steel Frame Non-res Ext Wall>
##       2: "Wood Siding", !- Outside Layer
##        v~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
##          Class: <Material>
##             Object [ID:50] <Wood Siding>
##                1: "Wood Siding"; !- Name
##
##       3: "Steel Frame NonRes Wall Insulation",  !- Layer 2
....
```

We first extract these three materials from the model.

```
ext_wall_mat <- model$to_table(which = c("Wood Siding",
                                         "Steel Frame NonRes Wall Insulation",
                                         "1/2IN Gypsum"))
ext_wall_mat
##     id                              name    class index              field
##  1: 50                      Wood Siding Material     1               Name
##  2: 50                      Wood Siding Material     2          Roughness
##  3: 50                      Wood Siding Material     3          Thickness
##  4: 50                      Wood Siding Material     4       Conductivity
##  5: 50                      Wood Siding Material     5            Density
##  6: 50                      Wood Siding Material     6      Specific Heat
```

```
##  7: 50                          Wood Siding Material     7 Thermal Absorptance
##  8: 50                          Wood Siding Material     8   Solar Absorptance
##  9: 50                          Wood Siding Material     9 Visible Absorptance
....
```

You can then modify the extracted information in the `data.frame`. Here, we increase the `Conductivity` of the `1/2IN Gypsum` from `0.16` to `0.20`, and double the `Thickness` of the wall insultion (`Steel Frame NonRes Wall Insulation`).

```
ext_wall_mat[name == "1/2IN Gypsum" & field == "Conductivity"]$value <- "0.20"

ext_wall_mat[name == "Steel Frame NonRes Wall Insulation" &
               field == "Thickness"]$value <- "0.174"
```

Subsequently, you can update the model using the modified `data.frame` with the `$update()` method in the `Idf` class.

```
model$update(ext_wall_mat)
## $`Wood Siding`
## <IdfObject: 'Material'> [ID:50] `Wood Siding`
## Class: <Material>
##   1*: "Wood Siding",  !- Name
##   2*: "MediumSmooth", !- Roughness
##   3*: 0.01,           !- Thickness {m}
##   4*: 0.11,           !- Conductivity {W/m-K}
##   5*: 544.62,         !- Density {kg/m3}
##   6*: 1210,           !- Specific Heat {J/kg-K}
##   7 : 0.9,            !- Thermal Absorptance
....

model$Material$`1/2IN Gypsum`
## <IdfObject: 'Material'> [ID:51] `1/2IN Gypsum`
## Class: <Material>
##   1*: "1/2IN Gypsum", !- Name
##   2*: "Smooth",       !- Roughness
##   3*: 0.0127,         !- Thickness {m}
##   4*: 0.2,            !- Conductivity {W/m-K}
##   5*: 784.9,          !- Density {kg/m3}
##   6*: 830,            !- Specific Heat {J/kg-K}
##   7 : 0.9,            !- Thermal Absorptance
##   8 : 0.92,           !- Solar Absorptance
....

model$Material$`Steel Frame NonRes Wall Insulation`
## <IdfObject: 'Material'> [ID:38] `Steel Frame NonRes Wall Insulation`
## Class: <Material>
```

```
##   1*: "Steel Frame NonRes Wall Insulation",  !- Name
##   2*: "MediumRough",       !- Roughness
##   3*: 0.174,               !- Thickness {m}
##   4*: 0.049,               !- Conductivity {W/m-K}
##   5*: 265,                 !- Density {kg/m3}
##   6*: 836.8,               !- Specific Heat {J/kg-K}
##   7 : 0.9,                 !- Thermal Absorptance
##   8 : 0.7,                 !- Solar Absorptance
....
```

14.3 Create new objects

14.3.1 add

You can add new objects to the model using the `$add()` method in the `Idf` class. The `$add()` method takes as it's first argument is a nested named list of the form. Note that `<CLASS NAME>` is case-sensitive while `<FIELD NAME>` is not.

```
`list(<CLASS NAME> = list(<FIELD NAME> = <FIELD VALUE>))`
```

Suppose you want to add a new material to your model, which you can define using EnergyPlus's `Material` class. Figure 14.1 can be translated to tell us that the `Material` class consists of the following fields

- `Name` is a required field of type `alpha` (i.e., a string)
- `Roughness` is a required field of type `choice` and the possible choices are `VeryRough`, `Rough`, `MediumRou.gh`, `MediumSmooth`, `Smooth`, and `VerySmooth`.
- `Thickness` is a required field with unit of measurement meters. It is a `real` number and must be $>= 0$
- `Conductivity` is a required field with unit of measurement $W/m.K$. It is a `real` number and must be $>= 0$
- `Density` is a required field with unit of measurement kg/m^3. It is a `real` number and must be $>= 0$
- `Specific Heat` is a required field with unit of measurement $J/kg.K$. It is a `real` number and must be $>= 100$
- `Thermal Absorptance` is *not* a required field. It is a `real` number, has a default value of 0.9 if not specified, and must be between 0 and 0.99999.
- `Solar Absorptance` is *not* a required field. It is a `real` number, has a default value of 0.7 if not specified, and must be between 0 and 0.99999.
- `Visible Absorptance` is *not* a required field. It is a `real` number, has a default value of 0.7 if not specified, and must be between 0 and 0.99999.

Based on the definitions provided in the IDD, you can define and add a new 100mm brick object. By default, all empty fields will be filled with default values specified in the IDD. As mentioned earlier, note that the class name (`Material`

```
Material,
    \memo Regular materials described with full set of thermal properties
    \min-fields 6
  A1 , \field Name
       \required-field
       \type alpha
       \reference MaterialName
  A2 , \field Roughness
       \required-field
       \type choice
       \key VeryRough
       \key Rough
       \key MediumRough
       \key MediumSmooth
       \key Smooth
       \key VerySmooth
  N1 , \field Thickness
       \required-field
       \units m
       \type real
       \minimum> 0
       \ip-units in
  N2 , \field Conductivity
       \required-field
       \units W/m-K
       \type real
       \minimum> 0
  N3 , \field Density
       \required-field
       \units kg/m3
       \type real
       \minimum> 0
  N4 , \field Specific Heat
       \required-field
       \units J/kg-K
       \type real
       \minimum 100
  N5 , \field Thermal Absorptance
       \type real
       \minimum> 0
       \default .9
       \maximum 0.99999
  N6 , \field Solar Absorptance
       \type real
       \default .7
       \minimum 0
       \maximum 1
  N7 ; \field Visible Absorptance
       \type real
       \minimum 0
       \default .7
       \maximum 1
```

Figure 14.1: Definition of EnergyPlus Material class based on the IDD.

in this example) is case-sensitive, while the field names (`Name`, `Roughness`, `Thickness`, etc.) are case-insensitive.

```
new_mat <- list(
  Material = list(
    Name = "100mm brick",
    ROUGHness = "Rough",
    thickness = 0.10,
    conducTIVITY = 0.89,
    density = 1920,
    `Specific Heat` = 790
  ),
  Material = list(
    name = "spandrel glass",
    roughness = "smooth",
    thickness = 0.006,
    conductivity = 0.99,
    density = 2528,
    `Specific Heat` = 880
  ))

model$add(new_mat)
## $`100mm brick`
## <IdfObject: 'Material'> [ID:682] `100mm brick`
## Class: <Material>
##   1*: "100mm brick", !- Name
##   2*: "Rough",       !- Roughness
##   3*: 0.1,           !- Thickness {m}
##   4*: 0.89,          !- Conductivity {W/m-K}
##   5*: 1920,          !- Density {kg/m3}
##   6*: 790;           !- Specific Heat {J/kg-K}
##
....

model$objects(c("100mm brick", "spandrel glass"))
## $`100mm brick`
## <IdfObject: 'Material'> [ID:682] `100mm brick`
## Class: <Material>
##   1*: "100mm brick", !- Name
##   2*: "Rough",       !- Roughness
##   3*: 0.1,           !- Thickness {m}
##   4*: 0.89,          !- Conductivity {W/m-K}
##   5*: 1920,          !- Density {kg/m3}
##   6*: 790;           !- Specific Heat {J/kg-K}
##
....
```

By default, all empty fields will be filled with default values specified in the IDD and the object would be defined using only the minimum number of fields. You change change this by setting with `.default = FALSE` (no default values used) and `.all = TRUE` (all fields are added).

14.3.2 duplicate

Chapter 15

Detailed output

15.1 Prerequisites

In this chapter we will focus on how to set and get EnergyPlus simulation outputs using the eplusr package. We will illustrate the manipulation of simulation data using tidyverse, and use ggplot2 to visualize the simulation data

```
library(eplusr)
library(tidyverse)
library(ggplot2)
library(here)
```

We will be working with the IDF and EPW file that pertains to the U.S. Department of Energy (DOE) Commercial Reference Building and Chicago's TMY3 respectively.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)

path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)
```

15.2 Variable dictionary reports

The variable dictionary reports are one of the most important outputs when working with EnergyPlus simulations. These reports inform EnergyPlus users the outputs that are available for a specific EnergyPlus simulation. Knowing the outputs that are available in your model would allow users to identify and subsequently specify relevant outputs for further analysis. Two data dictionary reports are produced and they are the **m**eter **d**ata **d**ictionary (`.mdd`) file and

the **r**eport **d**ata **d**ictionary (`.rdd`) file. The `.mdd` file lists the names of the output meters while the `.rdd` file lists the names of the output variables that are available for the simulation. However, you must first run the simulation before the available variables and meters can be known. By setting `weather = NULL`, a design day simulation will be run, allowing us to obtain the `.rdd` and `.mdd` file without running an annual simulation that can be time consuming for complex models.

```
job <- model$run(weather = NULL, dir = tempdir())
## Replace the existing IDF located at /private/var/folders/6l/8dz74knn0yd02c_bt8fl1pk
## EnergyPlus Starting
## EnergyPlus, Version 9.4.0-998c4b761e, YMD=2021.10.13 23:03
##
## Could not find platform independent libraries <prefix>
## Could not find platform dependent libraries <exec_prefix>
## Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
## Initializing Response Factors
## Calculating CTFs for "STEEL FRAME NON-RES EXT WALL"
## Calculating CTFs for "IEAD NON-RES ROOF"
## Calculating CTFs for "EXT-SLAB"
....
```

You can then retrieve the list of available variables and meters with the function `read_rdd()` and the `read_mdd()` function respectively. Both functions return a five column `data.table`.

```
mdd <- job$read_mdd()
mdd
## == EnergyPlus Meter Data Dictionary File ========================================
##   * EnergyPlus version: 9.4.0 (998c4b761e)
##   * Simulation started: 2021-10-13 23:03:00
##
## -- Details ---------------------------------------------------------------------
##      index reported_time_step report_type
##   1:     1               Zone       Meter
##   2:     2               Zone       Meter
##   3:     3               Zone       Meter
##   4:     4               Zone       Meter
....
```

```
rdd <- job$read_rdd()
rdd
## == EnergyPlus Report Data Dictionary File =======================================
##   * EnergyPlus version: 9.4.0 (998c4b761e)
##   * Simulation started: 2021-10-13 23:03:00
##
## -- Details ---------------------------------------------------------------------
```

```
##      index reported_time_step report_type
##   1:     1               Zone     Average
##   2:     2               Zone     Average
##   3:     3               Zone     Average
##   4:     4               Zone     Average
....
```

To specify an output variable of interest, you can add the name of the variable and the corresponding reporting frequency to the `Output:Variable` object. To avoid a long list of redundant and unnecessary output variables, you should first remove all of them.

```
model$Output_Variable <- NULL
model$Output_Meter <- NULL
```

You can then use the `$add()` method to add your variable output of interest. The first argument to the `$add()` method is a list of EnergyPlus object definitions where each list is named with a valid class name. In this case, the relevant EnergyPlus objects for defining output meters (`.mdd`) and variables (`.rdd`) are the `Output:Meter` and the `Output:Variable` objects respectively.

```
output_list <- list(
    Output_Variable = list(
        key_value = "*",
        Variable_Name = "Site Outdoor Air Drybulb Temperature",
        Reporting_Frequency = "Hourly"),
    Output_Variable = list(
        key_value = "*",
        Variable_Name = "Zone Mean Air Temperature",
        Reporting_Frequency = "Hourly"),
    Output_Meter = list(
        key_name = "Cooling:Electricity",
        Reporting_Frequency = "Hourly"),
    Output_Meter = list(
        key_name = "Heating:NaturalGas",
        Reporting_Frequency = "Hourly")
)

model$add(output_list)
## $<NA>
## <IdfObject: 'Output:Variable'> [ID:682]
## Class: <Output:Variable>
##   1 : "*",                  !- Key Value
##   2*: "Site Outdoor Air Drybulb Temperature",  !- Variable Name
##   3 : "Hourly";             !- Reporting Frequency
##
## $<NA>
```

```
## <IdfObject: 'Output:Variable'> [ID:683]
## Class: <Output:Variable>
....
```

The above code to add meters and variables can become unnecessarily long if you are adding multiple objects of the same class (in the above example, the `Output:Meter` and `Output:Variable` class. You can take advantage of `data.table`'s `:=` operator to assign the object name by reference.

```
variables <- list(
    key_value = "*",
    Variable_Name = c(
        "Site Outdoor Air Drybulb Temperature",
        "Zone Mean Air Temperature"
    ),
    Reporting_Frequency = "Hourly"
)

model$add(Output_Variable := variables)
## $<NA>
## <IdfObject: 'Output:Variable'> [ID:686]
## Class: <Output:Variable>
##   1 : "*",                 !- Key Value
##   2*: "Site Outdoor Air Drybulb Temperature",  !- Variable Name
##   3 : "Hourly";            !- Reporting Frequency
##
## $<NA>
## <IdfObject: 'Output:Variable'> [ID:687]
## Class: <Output:Variable>
....
```

Likewise, you can use the `$add()` method to add your variable meter of interest.

```
meters <- list(
    key_name = c(
        "Cooling:Electricity",
        "Heating:NaturalGas"
    ),
    Reporting_Frequency = "Hourly"
)

# add meter outputs to get hourly time-series energy consumption
model$add(Output_Meter := meters)
## $<NA>
## <IdfObject: 'Output:Meter'> [ID:688]
## Class: <Output:Meter>
##   1*: "Cooling:Electricity",  !- Key Name
```

```
##   2 : "Hourly";           !- Reporting Frequency
##
## $<NA>
## <IdfObject: 'Output:Meter'> [ID:689]
## Class: <Output:Meter>
##   1*: "Heating:NaturalGas",  !- Key Name
....
```

Here is a useful function `preprocess_idf()` that you can use to preprocess your EnergyPlus simulations so that it (1) uses the weather file for the simulation, (2) presents energy consumption outputs in kWh, and (3) remove all existing output meters and variables and add the list of meters and variables provided as arguments to the function.

```
preprocess_idf <- function(idf, meters, variables) {
    # make sure weather file input is respected
    idf$SimulationControl$Run_Simulation_for_Weather_File_Run_Periods <- "Yes"

    # make sure simulation is not run for sizing periods
    idf$SimulationControl$Run_Simulation_for_Sizing_Periods <- "No"

    # make sure energy consumption is presented in kWh
    if (is.null(idf$OutputControl_Table_Style)) {
        idf$add(OutputControl_Table_Style = list("HTML", "JtoKWH"))
    }else{
        idf$OutputControl_Table_Style$Unit_Conversion <- "JtoKWH"
    }

    # remove all existing meter and variable outputs
    if(!is.null(idf$`Output:Meter`)){
        idf$Output_Meter <- NULL
    }

    # remove all existing meter and variable outputs
    if(!is.null(idf$`Output:Table:Monthly`)){
        idf$`Output:Table:Monthly` <- NULL
    }

    if(!is.null(idf$`Output:Variable`)){
        idf$Output_Variable <- NULL
    }

    # add meter outputs to get hourly time-series energy consumption
    idf$add(Output_Meter := meters)

    # add variable outputs to get hourly zone air temperature
```

```
    idf$add(Output_Variable := variables)

    # make sure the modified model is returned
    return(idf)
}
```

The following code demonstrates the use of the function `preprocess_idf()`.

```
meters <- list(
    key_name = c(
        "Cooling:Electricity",
        "Heating:NaturalGas",
        "Heating:Electricity",
        "InteriorLights:Electricity",
        "ExteriorLights:Electricity",
        "InteriorEquipment:Electricity",
        "Fans:Electricity",
        "Pumps:Electricity",
        "WaterSystems:NaturalGas"
    ),
    Reporting_Frequency = "Hourly"
)

variables <- list(
    key_value = "*",
    Variable_Name = c(
        "Site Outdoor Air Drybulb Temperature",
        "Site Outdoor Air Relative Humidity"
    ),
    Reporting_Frequency = "Hourly"
)

model <- preprocess_idf(model, meters, variables)
```

Check if the output variables and meters have been correctly added

```
model$Output_Variable
## $<NA>
## <IdfObject: 'Output:Variable'> [ID:691]
## Class: <Output:Variable>
##   1 : "*",                !- Key Value
##   2*: "Site Outdoor Air Drybulb Temperature",  !- Variable Name
##   3 : "Hourly";           !- Reporting Frequency
##
## $<NA>
## <IdfObject: 'Output:Variable'> [ID:692]
## Class: <Output:Variable>
```

```
....
```

```
model$Output_Meter
## $<NA>
## <IdfObject: 'Output:Meter'> [ID:682]
## Class: <Output:Meter>
##   1*: "Cooling:Electricity",  !- Key Name
##   2 : "Hourly";               !- Reporting Frequency
##
## $<NA>
## <IdfObject: 'Output:Meter'> [ID:683]
## Class: <Output:Meter>
##   1*: "Heating:NaturalGas",  !- Key Name
....
```

Before you can explore the outputs, you have to first save the model and run the simulation.

```
model$save(here("data", "idf", "model_preprocessed.idf"), overwrite = TRUE)
## Replace the existing IDF located at /Users/adrianchong/Documents/GitHub/r4bes/data/idf/model_p
job <- model$run(epw, dir = tempdir())
## EnergyPlus Starting
## EnergyPlus, Version 9.4.0-998c4b761e, YMD=2021.10.13 23:03
##
## Could not find platform independent libraries <prefix>
## Could not find platform dependent libraries <exec_prefix>
## Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
## Initializing Response Factors
## Calculating CTFs for "STEEL FRAME NON-RES EXT WALL"
## Calculating CTFs for "IEAD NON-RES ROOF"
## Calculating CTFs for "EXT-SLAB"
....
```

You will then be able to extract the output variables and meters that have just been specified.

```
report <- job$report_data() %>%
    drop_na() %>%
    select(datetime, name, value) %>%
    filter(name %in% variables$Variable_Name | name %in% meters$key_name) %>%
    pivot_wider(names_from = name, values_from = value)

head(report)
## # A tibble: 6 x 12
##   datetime            `Site Outdoor Air D~ `Site Outdoor Air ~ `InteriorLights:~
##   <dttm>                             <dbl>               <dbl>             <dbl>
## 1 2021-01-01 01:00:00                -7.82                72.6          9649498.
## 2 2021-01-01 02:00:00               -11.9                 73            9649498.
```

```
## 3 2021-01-01 03:00:00               -11.4                  73          9649498.
## 4 2021-01-01 04:00:00               -11.1                  73          9649498.
## 5 2021-01-01 05:00:00               -10.8                  73          9649498.
## 6 2021-01-01 06:00:00               -10.6                  73          9649498.
## # ... with 8 more variables: InteriorEquipment:Electricity <dbl>,
....
```

15.3 mtd file

You can also look at the mtd file to know what report variables are on which meters and vice versa

Chapter 16

Model Exploration

16.1 Prerequisites

In this chapter we will explore the model to uncover energy characteristics and create an overview of general trends. We will use eplusr to run the EnergyPlus simulation and extract relevant model inputs/outputs, tidyverse for data transformation, and ggplot2 to visualize the extracted and subsequently transformed data.

```
library(eplusr)
library(tidyverse)
library(lubridate)
library(ggplot2)
library(here)
```

We will be working with the IDF and EPW file that pertains to the U.S. Department of Energy (DOE) Commercial Reference Building and Chicago's TMY3 respectively.

```
path_idf <- here("data", "idf", "RefBldgMediumOfficeNew2004_Chicago.idf")
model <- read_idf(path_idf)

path_epw <- here("data", "epw", "USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw")
epw <- read_epw(path_epw)
```

16.2 Extracting

Before carrying out any model exploration, you need to first specify the outputs of interest. The code below preprocesses the model by adding the list of output meters and variables to the model (See Chapter 15 for details).

```
preprocess_idf <- function(idf, meters, variables) {
    # make sure weather file input is respected
    idf$SimulationControl$Run_Simulation_for_Weather_File_Run_Periods <- "Yes"

    # make sure simulation is not run for sizing periods
    idf$SimulationControl$Run_Simulation_for_Sizing_Periods <- "No"

    # make sure energy consumption is presented in kWh
    if (is.null(idf$OutputControl_Table_Style)) {
        idf$add(OutputControl_Table_Style = list("HTML", "JtoKWH"))
    }else{
        idf$OutputControl_Table_Style$Unit_Conversion <- "JtoKWH"
    }

    # remove all existing meter and variable outputs
    if(!is.null(idf$`Output:Meter`)){
        idf$Output_Meter <- NULL
    }

    # remove all existing meter and variable outputs
    if(!is.null(idf$`Output:Table:Monthly`)){
        idf$`Output:Table:Monthly` <- NULL
    }

    if(!is.null(idf$`Output:Variable`)){
        idf$Output_Variable <- NULL
    }

    # add meter outputs to get hourly time-series energy consumption
    idf$add(Output_Meter := meters)

    # add variable outputs to get hourly zone air temperature
    idf$add(Output_Variable := variables)

    # make sure the modified model is returned
    return(idf)
}

meters <- list(
    key_name = c(
        "Cooling:Electricity",
        "Heating:NaturalGas",
        "Heating:Electricity",
        "InteriorLights:Electricity",
        "ExteriorLights:Electricity",
```

```
        "InteriorEquipment:Electricity",
        "Fans:Electricity",
        "Pumps:Electricity",
        "WaterSystems:NaturalGas"
    ),
    Reporting_Frequency = "Hourly"
)

variables <- list(
    key_value = "*",
    Variable_Name = c(
        "Site Outdoor Air Drybulb Temperature",
        "Site Outdoor Air Relative Humidity"
    ),
    Reporting_Frequency = "Hourly"
)

model <- preprocess_idf(model, meters, variables)
```

You can then run the simulation and it will be possible to extract the output of interest from the model.

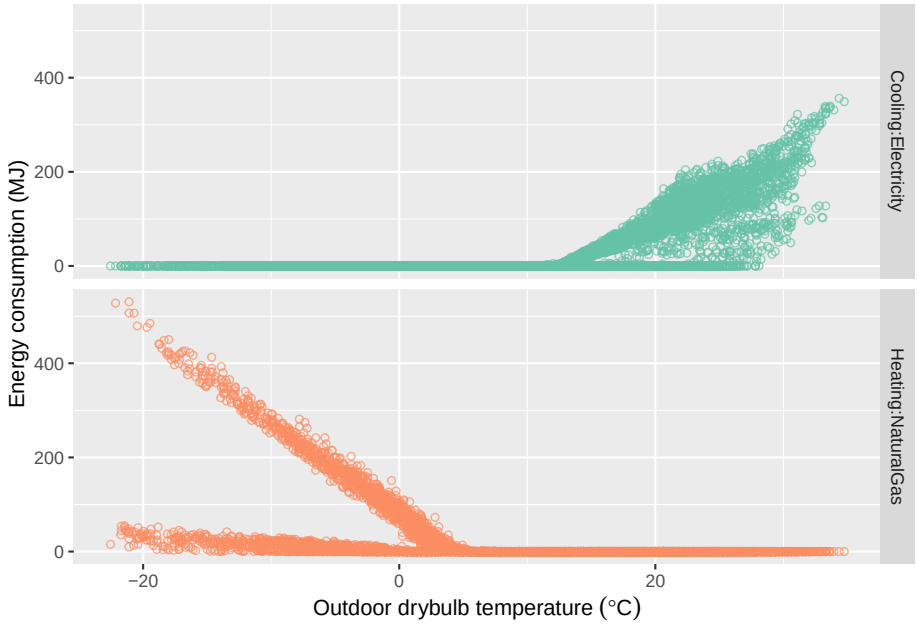
```
model$save(here("data", "idf", "model_preprocessed.idf"), overwrite = TRUE)
## Replace the existing IDF located at /Users/adrianchong/Documents/GitHub/r4bes/data/idf/model_p
job <- model$run(epw, dir = tempdir())
## Replace the existing IDF located at /private/var/folders/6l/8dz74knn0yd02c_bt8fl1pk80000gn/T/R
## EnergyPlus Starting
## EnergyPlus, Version 9.4.0-998c4b761e, YMD=2021.10.13 23:03
##
## Could not find platform independent libraries <prefix>
## Could not find platform dependent libraries <exec_prefix>
## Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
## Initializing Response Factors
## Calculating CTFs for "STEEL FRAME NON-RES EXT WALL"
## Calculating CTFs for "IEAD NON-RES ROOF"
## Calculating CTFs for "EXT-SLAB"
....
```

16.3 Energy signature

```
pt_of_interest <- c("Site Outdoor Air Drybulb Temperature",
                    "Cooling:Electricity",
                    "Heating:NaturalGas")
```

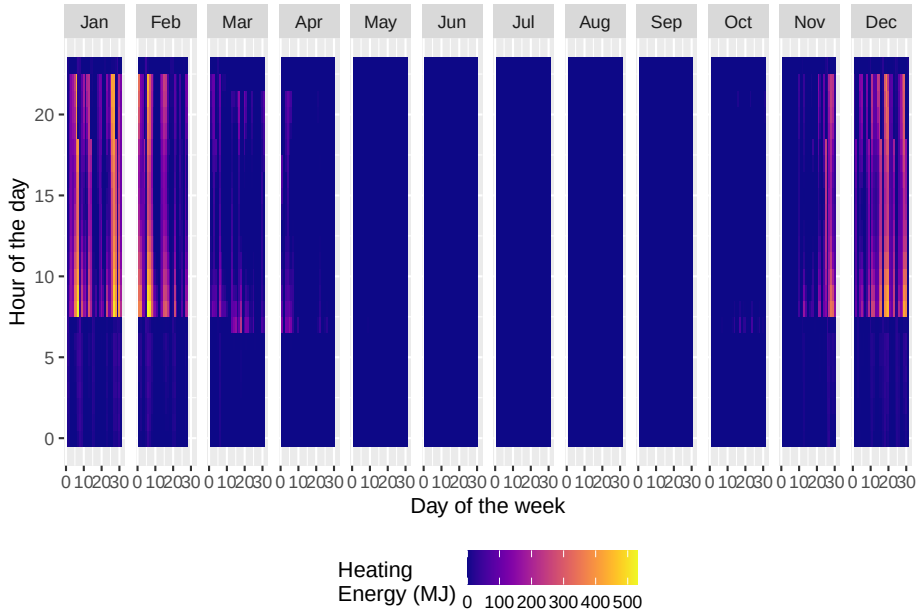
```
report <- job$report_data() %>%
  drop_na() %>%
  select(datetime, name, value) %>%
  filter(name %in% pt_of_interest) %>%
  pivot_wider(names_from = name, values_from = value) %>%
  pivot_longer(cols = c("Cooling:Electricity", "Heating:NaturalGas"),
               names_to = "type", values_to = "value") %>%
  mutate(value = value * 1e-6, # convert J to MJ
         month = month(datetime, label = TRUE), # create a new column containing the m
         day = day(datetime), # create a new column containing the day of the month
         wday = wday(datetime, label = TRUE), # create a new column containing day of
         hour = hour(datetime)) # create a new column containing the hour of the day
```

```
ggplot(report, aes(x = `Site Outdoor Air Drybulb Temperature`,
                   y = value,
                   color = type)) +
  geom_point(shape = 1, alpha = 0.7) +
  facet_grid(rows = vars(type)) +
  scale_color_brewer(palette = "Set2") +
  xlab(expression("Outdoor drybulb temperature"~(degree*C))) +
  ylab("Energy consumption (MJ)") +
  theme(legend.position = "none")
```




```
report_heating <- report %>%
  filter(type == "Heating:NaturalGas")
```

```
ggplot(report_heating, aes(x = day, y = hour,
                           fill = value)) +
  geom_tile() +
  scale_fill_viridis(name = "Heating\nEnergy (MJ)",
                     option = "plasma") +
  facet_grid(cols = vars(month)) +
  ylab("Hour of the day") +
  xlab("Day of the week") +
  theme(legend.position = "bottom")
```



Part IV

Program

Chapter 17

Introduction

Chapter 18

Energy Efficient Measures

Chapter 19

Parametric simulations

Part V

Advanced

Chapter 20

Introduction

Chapter 21

Sensitivity analysis

William of Occam to seek an economical description of natural phenomena and avoid excessive models that are overparameterized. Principal of Parisimony.

Chapter 22

Optimization

Chapter 23

Calibration

Taking reference to the quotation of statistician George Box that “All models are wrong, but some are useful”

Part VI

Reproduce

Chapter 24

Introduction

Chapter 25

R Markdown

Chapter 26

Containers

Bibliography

Best Directory | Building Energy Software Tools, a. URL https://www.buildingenergysoftwaretools.com/.

EnergyPlus, b. URL https://www.energy.gov/eere/buildings/downloads/energyplus-0.

ASHRAE. ASHRAE Guideline 14-2014 Measurement of energy, demand, and water savings, 2014.

Winston Chang. *R graphics cookbook: practical recipes for visualizing data*. O'Reilly, Beijing ; Boston, second edition edition, 2018. ISBN 978-1-4919-7860-3. OCLC: on1076544092.

Michael Deru, Kristin Field, Daniel Studer, Kyle Benne, Brent Griffith, Paul Torcellini, Bing Liu, Mark Halverson, Dave Winiarski, and Michael Rosenberg. US Department of Energy commercial reference building models of the national building stock. 2011. Publisher: National Renewable Energy Laboratory.

Kieran Healy. *Data visualization: a practical introduction*. Princeton University Press, 2018.

Hongyuan Jia and Adrian Chong. eplusr: A framework for integrating building energy simulation and data-driven analytics. *Energy and Buildings*, 237: 110757, April 2021. ISSN 03787788. doi: 10.1016/j.enbuild.2021.110757. URL https://linkinghub.elsevier.com/retrieve/pii/S0378778821000414.

Hadley Wickham and Garrett Grolemund. *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly, Sebastopol, CA, first edition edition, 2016. ISBN 978-1-4919-1039-9 978-1-4919-1036-8. OCLC: ocn968213225.

Leland Wilkinson. The grammar of graphics. In *Handbook of computational statistics*, pages 375–414. Springer, 2012.