Assignment 1 - Chapter 8-19 Summary LK

Lynnstacy Kegeshi

2024-10-28

Contents

This is a summary of Chapter 8-19 of the book **R for Data Science**. We shall be using be data set *House Sale* to perform our analysis based on the content from Chapter 8 - 19

We shall be using the following libraries

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
library(stringr)
library(forcats) # Factor manipulation
library(lubridate) # Date and time
library(magrittr) #For Pipe operation
```

```
##
## Attaching package: 'magrittr'
##
## The following object is masked from 'package:purrr':
##
```

```
##     set_names
##
## The following object is masked from 'package:tidyr':
##
##     extract
```

Chapter 8

This chapter focuses on importing data into R, specifically using the read r package which is part of tidyverse.

Reading Data into R Studio

Most functions in readr are designed to turn flat files into data frames. Key functions include:

- read_csv(): Reads comma-delimited files.
- read_csv2(): Reads semicolon-separated files.
- read_tsv(): Reads tab-delimited files.
- read_delim(): Reads files with any delimiter.
- read_fwf(): Reads fixed-width files.
- read_log(): Reads Apache-style log files.

Using the read_csv(), we imported the HouseSale Data to illustrate this fucntion.

```r
house_sale<- read.csv("HouseSale.csv")
head(house_sale)
```

```
##   HOUSE_ID HousePrice StoreArea BasementArea LawnArea StreetHouseFront
## 1        1     163000       433          662     9120               76
## 2        2     102000       396          836     8877               67
## 3        3     265979       864            0    11700               65
## 4        4     181900       572          594    14585               NA
## 5        5     252000      1043            0    10574               85
## 6        6     180000       440          570    10335               78
##      Location ConnectivityType    BuildingType ConstructionYear  EstateType
## 1    RK Puram            Byway IndividualHouse             1958       Other
## 2 Jama Masjid            Byway IndividualHouse             1951       Other
## 3      Burari            Byway IndividualHouse             1880       Other
## 4    RK Puram            Byway IndividualHouse             1960       Other
## 5      Bawana            Byway IndividualHouse             2005       Other
## 6    Timarpur            Byway IndividualHouse             1968 SemiPrivate
##   SellingYear Rating SaleType
## 1        2008      6 NewHouse
## 2        2006      4 NewHouse
## 3        2009      7 NewHouse
## 4        2007      6 NewHouse
## 5        2009      8 NewHouse
## 6        2006      5 NewHouse
```

You can also customize the import of data through the following options.

1. Skipping Metadata: Use skip = n to skip the first n lines or comment = "#".

2. No Column Names: Use col_names = FALSE to treat the first row as data.
3. Custom Column Names: Pass a character vector to col_names.
4. Handling Missing Values: Use the na argument to specify missing value representations.

Writing into a file

‘readr also provides functions for writing data back to disk:

- write_csv(): Writes CSV files.
- write_tsv(): Writes tab-separated files.
- write_excel_csv(): Writes CSV files for Excel.
- write_rds() and read_rds(): Store data in R’s custom binary format.
- write_feather() and read_feather(): Use the feather package for fast binary file format.

Parsing Vectors

The parse_*() functions are designed to convert character vectors into more specialized types. Here are some of the most commonly used functions

- parse_logical(): Parses logical values.
- parse_integer(): Parses integer values.
- parse_double(): Parses double (numeric) values.
- parse_number(): Parses numbers, ignoring non-numeric characters.
- parse_character(): Parses character strings.
- parse_factor(): Parses factors.
- parse_datetime(): Parses date-time values.
- parse_date(): Parses date values.
- parse_time(): Parses time values.

```r
# Parsing logical values
logical_vector <- parse_logical(c("TRUE", "FALSE", "NA"))
print(logical_vector)
```

```
## [1]  TRUE FALSE    NA
```

```r
#> [1] TRUE FALSE NA

# Parsing integer values
integer_vector <- parse_integer(c("1", "2", "3"))
print(integer_vector)
```

```
## [1] 1 2 3
```

```r
#> [1] 1 2 3

# Parsing double values
double_vector <- parse_double(c("1.23", "4.56", "7.89"))
print(double_vector)
```

```
## [1] 1.23 4.56 7.89
```

```
#> [1] 1.23 4.56 7.89

# Parsing numbers with non-numeric characters
number_vector <- parse_number(c("$100", "20%", "It cost $123.45"))
print(number_vector)
```

```
## [1] 100.00  20.00 123.45
```

```
#> [1] 100  20 123.45

# Parsing date values
date_vector <- parse_date(c("2010-01-01", "1979-10-14"))
print(date_vector)
```

```
## [1] "2010-01-01" "1979-10-14"
```

```
#> [1] "2010-01-01" "1979-10-14"
```

We check the structure of our data to see if we need to parse any data.

```
str(house_sale)
```

```
## 'data.frame':    1300 obs. of  14 variables:
##  $ HOUSE_ID        : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ HousePrice      : int  163000 102000 265979 181900 252000 180000 115000 176000 192000 132500 ...
##  $ StoreArea       : int  433 396 864 572 1043 440 336 486 430 264 ...
##  $ BasementArea    : int  662 836 0 594 0 570 0 552 24 588 ...
##  $ LawnArea        : int  9120 8877 11700 14585 10574 10335 21750 9900 3182 7758 ...
##  $ StreetHouseFront: int  76 67 65 NA 85 78 100 NA 43 NA ...
##  $ Location        : chr  "RK Puram" "Jama Masjid" "Burari" "RK Puram" ...
##  $ ConnectivityType: chr  "Byway" "Byway" "Byway" "Byway" ...
##  $ BuildingType    : chr  "IndividualHouse" "IndividualHouse" "IndividualHouse" "IndividualHouse" ..
##  $ ConstructionYear: int  1958 1951 1880 1960 2005 1968 1960 1968 2004 1962 ...
##  $ EstateType      : chr  "Other" "Other" "Other" "Other" ...
##  $ SellingYear     : int  2008 2006 2009 2007 2009 2006 2009 2008 2010 2007 ...
##  $ Rating          : int  6 4 7 6 8 5 5 7 8 5 ...
##  $ SaleType        : chr  "NewHouse" "NewHouse" "NewHouse" "NewHouse" ...
```

Since our data(House Sale) is already structured with the correct types the parse_* functions are unnecessary here.

Chapter 9

This chapter introduces the concept of tidy data, a consistent way to organize data in R. This is done using tidyr which is a package inside tidyverse.

Tidy data is organized according to three interrelated rules:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Tidy data has two main advantages: **Consistency** and **Efficiency**

Missing Values

Missing values can be explicit (flagged with NA) or implicit (not present in the data). tidyr provides tools to handle missing values:

complete(): Ensures all combinations of variables are present, filling in missing values with NA. Applying this to our data where StreetHouseFront had explicit missing values, we filled missing values with the median on non missing values.

##Applying ggplot and dplyr on tidydata

```
# Fill missing StreetHouseFront values with the median of non-missing values
house_sale <- house_sale %>%
  mutate(StreetHouseFront2 = ifelse(is.na(StreetHouseFront), median(StreetHouseFront, na.rm = TRUE), Str

#Chcking if NAs had been replaced.
head(house_sale$StreetHouseFront)
```

```
## [1] 76 67 65 NA 85 78
```

```
head(house_sale$StreetHouseFront2)
```

```
## [1] 76 67 65 70 85 78
```

fill(): Fills missing values with the most recent non-missing value.

```
# Fill NA values in StreetHouseFront with the most recent non-NA value
house_sale3 <- house_sale %>%
  fill(StreetHouseFront)

#Chcking if NAs had been replaced.
head(house_sale$StreetHouseFront)
```

```
## [1] 76 67 65 NA 85 78
```

```
head(house_sale3$StreetHouseFront)
```

```
## [1] 76 67 65 65 85 78
```

The chapter also provides examples of working with tidy data using dplyr and ggplot2

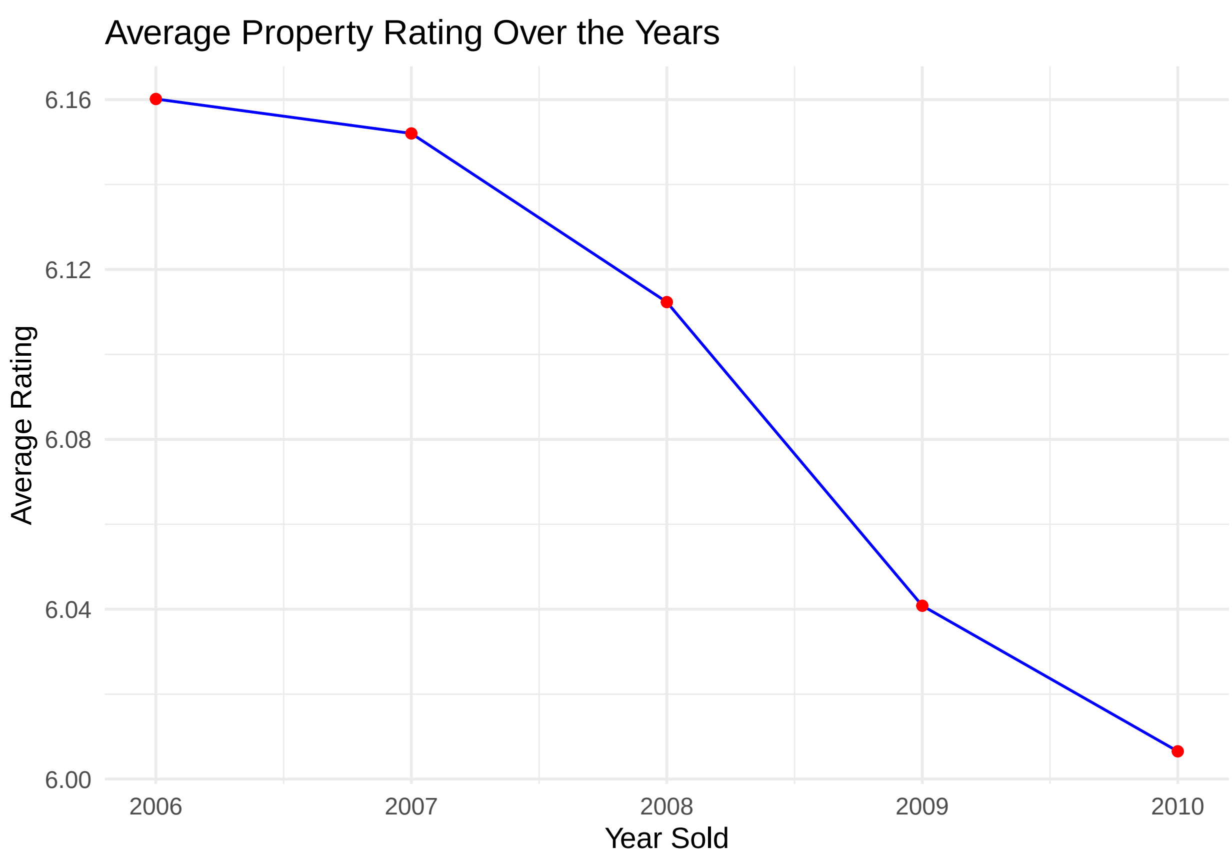
```
# Calculate average house price by Location
house_sale %>%
  group_by(Location) %>%
  summarize(avg_price = mean(HousePrice, na.rm = TRUE))#na.rm ignores missing values
```

```
## # A tibble: 25 x 2
##    Location     avg_price
##    <chr>            <dbl>
##  1 Adarsh Nagar   190859.
##  2 Ajmere Gate    124000
```

```
##  3 Bawana         225317.
##  4 Burari         130733.
##  5 Chanakyapuri   103593.
##  6 Chhatarpur     126289.
##  7 Dhaula Kuan    214457.
##  8 Ina Colony     214411.
##  9 India Gate     198040.
## 10 Jama Masjid    128418.
## # i 15 more rows
```

```r
# Calculate average rating by SellingYear
avg_rating_per_year <- house_sale %>%
  group_by(SellingYear) %>%
  summarize(avg_rating = mean(Rating, na.rm = TRUE))

# Plot average rating over time
ggplot(avg_rating_per_year, aes(x = SellingYear, y = avg_rating)) +
  geom_line(color = "blue") +
  geom_point(color = "red") +
  labs(title = "Average Property Rating Over the Years", x = "Year Sold", y = "Average Rating") +
  theme_minimal()
```



Most real-world data is not tidy. Two common problems are:

- One variable spread across multiple columns.
- One observation scattered across multiple rows.

To address these issues, tidyr provides two key functions: gather() and spread().

'separate() and unite() are used to split and combine columns, respectively.

Chapter 10

Keys are variables that uniquely identify an observation. There are two types of keys:

1. Primary Key: Uniquely identifies an observation in its own table.
2. Foreign Key: Uniquely identifies an observation in another table.

Mutating Joins

Mutating joins combine variables from two tables by matching observations based on their keys.

- inner_join(): Matches pairs of observations where keys are equal.
- left_join(): Keeps all observations in the left table.
- right_join(): Keeps all observations in the right table.
- full_join(): Keeps all observations in both tables.

Joins are visualized by matching rows based on key variables.

Filtering Joins

Filtering joins match observations and affect the observations, not the variables:

semi_join(): Keeps all observations in the left table that have a match in the right table. anti_join(): Drops all observations in the left table that have a match in the right table.

Set Operations

You can use set operations to compare data in tables.

Use intersect()if you have two tables with house data and want to find common entries, Use union() to combine two tables without duplicates. Use setdiff() to find rows present in one table but not in the other

When working with real data, identify the primary keys and make sure no primary key variables are missing Using anti_join() to check if all foreign key values in one table match primary key values in another.

Chapter 11: Strings with stringr

It introduces string manipulation in R using stringr which is part of the stringr package.

String Basics

Strings can be created using either single or double quotes. To include special characters like quotes or backslashes, use the escape character `\`. The writeLines() function shows the raw contents of a string.

String Length

The str_length() function from stringr tells you the number of characters in a string:

```r
str_length(c("a", "R for data science", NA))
```

```
## [1]  1 18 NA
```

Combining Strings

Use str_c() to combine two or more strings. The sep argument controls how they are separated, and collapse collapses a vector of strings into a single string:

```r
# Add details column, which is combines Location and BuildingType
house_sale <- house_sale %>%
  mutate(details = str_c(Location, BuildingType, sep = ", "))

head(house_sale$details)
```

```
## [1] "RK Puram, IndividualHouse"    "Jama Masjid, IndividualHouse"
## [3] "Burari, IndividualHouse"      "RK Puram, IndividualHouse"
## [5] "Bawana, IndividualHouse"      "Timarpur, IndividualHouse"
```

Subsetting Strings

The str_sub() function extracts parts of a string based on start and end positions. It can also be used to modify strings:

```r
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
## [1] "App" "Ban" "Pea"
```

Locales

Functions like str_to_lower(), str_to_upper(), and str_to_title() change the case of text. We changed the case of the vectors in Building Type to small letters with the code below:

```r
house_sale <- house_sale %>%
    mutate(BuildingType = str_to_lower(BuildingType))

head(house_sale$BuildingType)
```

```
## [1] "individualhouse" "individualhouse" "individualhouse" "individualhouse"
## [5] "individualhouse" "individualhouse"
```

Anchors

Anchors like ^ and $ match the start and end of a string, respectively. The word boundary matches the boundary between words.

Character Classes and Alternatives

Special patterns match more than one character:

- `\d` matches any digit.
- `\s` matches any whitespace.
- [abc] matches a, b, or c.
- [^abc] matches anything except a, b, or c.

Tools

Various stringr functions apply regexps to real problems:

1. str_detect(): Determines if a character vector matches a pattern.
2. str_extract(): Extracts the actual text of a match.
3. str_replace() and str_replace_all(): Replace matches with new values.
4. str_split(): Splits a string into pieces.
5. str_locate() and str_locate_all(): Give the starting and ending positions of each match.

Chapter 12: Forecasts with Forecat

In R, factors are used to work with categorical variables, which have a fixed and known set of possible values. To work with factors, the forcats package is used, which provides tools for dealing with categorical variables.

Creating Factors

When recording variables like months, using strings can lead to issues such as typos and unhelpful sorting. These problems can be fixed by creating a factor with a predefined list of valid levels. For example:

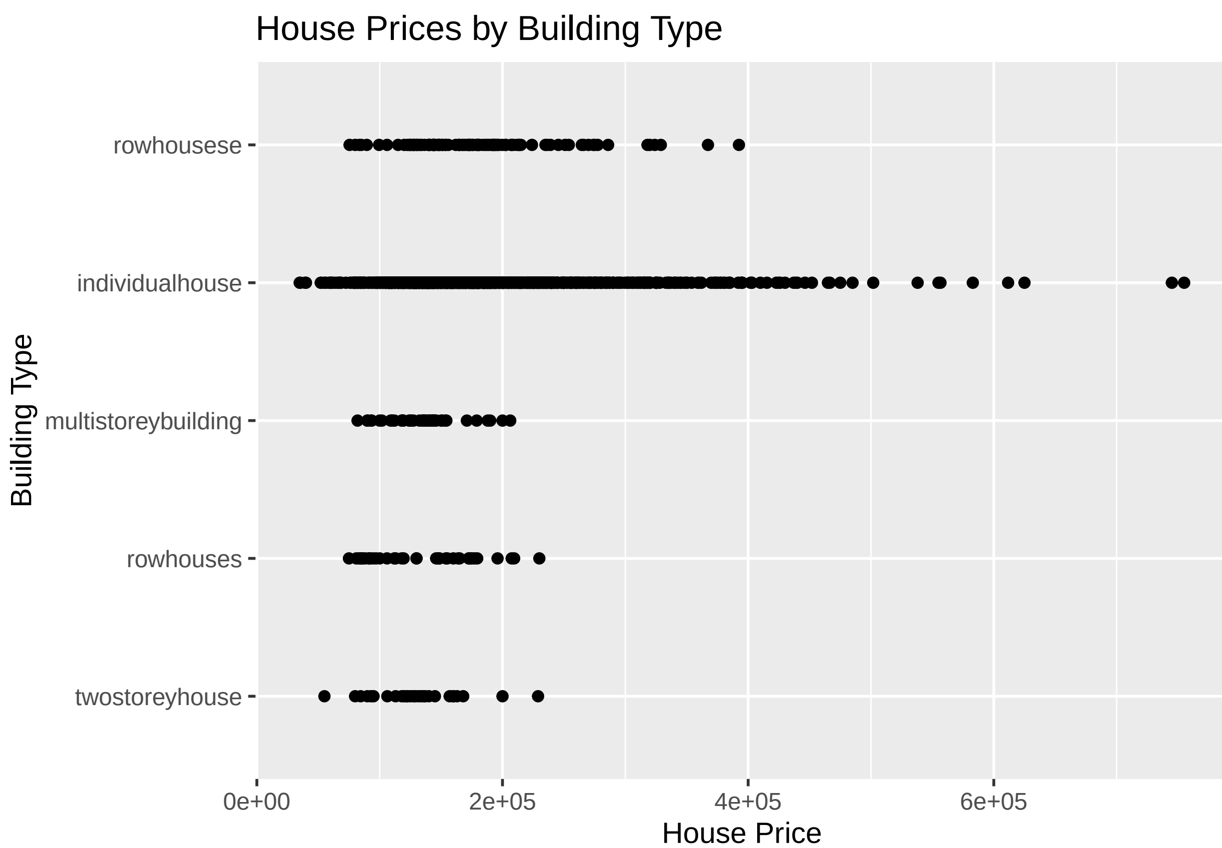
```r
x1 <- c("Dec", "Apr", "Jan", "Mar")
month_levels <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
y1 <- factor(x1, levels = month_levels)
y1
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Modifying Factor Order

Changing the order of factor levels in visualizations can make patterns easier to interpret. Functions like fct_reorder() and fct_relevel() are used to reorder levels based on specific criteria. For example, we can reorder Building type based on houseprice.

```r
# Reorder BuildingType based on average HousePrice
house_sale %>%
  ggplot(aes(x = HousePrice, y = fct_reorder(BuildingType, HousePrice))) +
  geom_point() +
  labs(title = "House Prices by Building Type", x = "House Price", y = "Building Type")
```



Modifying Factor Levels

Changing the values of factor levels can clarify labels and collapse levels for high-level displays. The fct_recode() function allows users to recode factor levels. Basically renaming.

```
#Converting EstateType from character to factor

house_sale <- house_sale %>%
  mutate(EstateType = as.factor(EstateType))

# Recode EstateType levels
house_sale <- house_sale %>%
  mutate(EstateTypeNew = fct_recode(EstateType,
                                 "Private-Builder" = "PrivateBuilder",
                                 "Semi-Private" = "SemiPrivate",
                                 "Gvt-Build" = "GovernmentBuild",
                                 "Community" = "Society",
                                 "Other" = "Other"))

head(house_sale$EstateTypeNew)
```

```
## [1] Other         Other         Other         Other         Other
## [6] Semi-Private
## Levels: Gvt-Build Other Private-Builder Semi-Private Community
```

To combine groups, multiple old levels can be assigned to the same new level. For collapsing many levels, fct_collapse() is useful.

```
#Converting EstateType from character to factor
house_sale <- house_sale %>%
  mutate(SaleType = as.factor(SaleType))

# Collapse SaleType levels into broader categories
house_sale <- house_sale %>%
  mutate(SaleType2 = fct_collapse(SaleType,
                                  "New" = c("NewHouse"),
                                  "Old" = c("FifthResale", "FirstResale", "FourthResale", "SecondResale

# Count new sale types
print(head(house_sale$SaleType2))
```

```
## [1] New New New New New New
## Levels: Old New
```

Chapter 13: Date and times wit lubridate

This chapter explored working with dates and time in R. This will be done using the lubridate package.

There are three types of date/time data that refer to an instant in time:

1. A date. Tibbles print this as .
2. A time within a day. Tibbles print this as .
3. A datetime is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as . Elsewhere in R these are called POSIXct. To get the current date or datetime you can use today() or now():

```
today()
```

```
## [1] "2024-10-31"
```

```
now()
```

```
## [1] "2024-10-31 15:25:15 EAT"
```

From Strings

Date/time data often comes as strings. lubridate provides helpers that automatically work out the format once you specify the order of the component. For example:

```
ymd("2017-01-31")
```

```
## [1] "2017-01-31"
```

```r
mdy("January 31st, 2017")
```

```
## [1] "2017-01-31"
```

```r
dmy("31-Jan-2017")
```

```
## [1] "2017-01-31"
```

To create a datetime, add an underscore and one or more of "h", "m", and "s" to the name of the parsing function:

```r
ymd_hms("2017-01-31 20:11:59")
```

```
## [1] "2017-01-31 20:11:59 UTC"
```

```r
mdy_hm("01/31/2017 08:01")
```

```
## [1] "2017-01-31 08:01:00 UTC"
```

In some instances, different components of the datetime will be spread across multiple columns. To create a date/time from this sort of input, use make_date() for dates, or make_datetime() for datetimes.

You may want to switch between a datetime and a date. That's the job of as_datetime() and as_date()

Getting Components

You can pull out individual parts of the date with the accessor functions year(), month(), mday() (day of the month), yday() (day of the year), wday() (day of the week), hour(), minute(), and second().

```r
datetime <- ymd_hms("2016-07-08 12:34:56")
year(datetime)
```

```
## [1] 2016
```

```r
month(datetime)
```

```
## [1] 7
```

```r
mday(datetime)
```

```
## [1] 8
```

```r
yday(datetime)
```

```
## [1] 190
```

```r
wday(datetime)
```

```
## [1] 6
```

For month() and wday() you can set label = TRUE to return the abbreviated name of the month or day of the week.

Time Spans

These classes represent time spans.

- Durations, which represent an exact number of seconds.
- Periods, which represent human units like weeks and months.
- Intervals, which represent a starting and ending point.

In R, when you subtract two dates, you get a difftime object.

Now for durations, durations come with a number of convenient constructors.

```r
dseconds(15)
```

```
## [1] "15s"
```

```r
dminutes(10)
```

```
## [1] "600s (~10 minutes)"
```

```r
dhours(c(12, 24))
```

```
## [1] "43200s (~12 hours)" "86400s (~1 days)"
```

```r
ddays(0:5)
```

```
## [1] "0s"                 "86400s (~1 days)"   "172800s (~2 days)"
## [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

```r
dweeks(3)
```

```
## [1] "1814400s (~3 weeks)"
```

```r
dyears(1)
```

```
## [1] "31557600s (~1 years)"
```

Periods are time spans but do not have a fixed length in seconds; instead, they work with "human" times, like days and months.

```r
one_pm <- ymd_hms("2016-03-12 13:00:00", tz = "America/New_York")
one_pm + days(1)
```

```
## [1] "2016-03-13 13:00:00 EDT"
```

Period can also be constructed with friendly constructors.

```r
seconds(15)
```

```
## [1] "15S"
```

```r
minutes(10)
```

```
## [1] "10M 0S"
```

```r
hours(c(12, 24))
```

```
## [1] "12H 0M 0S" "24H 0M 0S"
```

```r
days(7)
```

```
## [1] "7d 0H 0M 0S"
```

```r
months(1:6)
```

```
## [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
## [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

```r
weeks(3)
```

```
## [1] "21d 0H 0M 0S"
```

```r
years(1)
```

```
## [1] "1y 0m 0d 0H 0M 0S"
```

Timezones

One can find out what R thinks the current time zone is with Sys.timezone() And see the complete list of all time zone names with OlsonNames()

```r
Sys.timezone()
```

```
## [1] "Africa/Nairobi"
```

```r
length(OlsonNames())
```

```
## [1] 596
```

```r
head(OlsonNames())
```

```
## [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
## [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

Chapter 14: Pipes with magrittr

Pipes are used to express a sequence of multiple operations. Since the tidyverse library automatically loads dplyr and %>%, we won't need to load magrittr explicitly.Applying the learnings of this chapter to my house_sale data:

Basic Pipe (%>%):

It streamlines multiple operations such as filtering and mutating.

```r
house_location<- house_sale %>%
  filter(Location == "Dhaula Kuan") %>%
  mutate(price_per_sqft = HousePrice / (StoreArea+BasementArea+LawnArea))

head(house_location %>% select(Location, price_per_sqft), 5)
```

```
##      Location price_per_sqft
## 1 Dhaula Kuan       15.88115
## 2 Dhaula Kuan       16.21472
## 3 Dhaula Kuan       14.18628
## 4 Dhaula Kuan        1.73032
## 5 Dhaula Kuan       11.02739
```

Tee Pipe (%T>%)

Use to print interim output while retaining the original data.

```r
house_location<- house_sale %>%
  filter(Location == "Dhaula Kuan") %>%
  head(5)%T>%
  print() %>%
  mutate(price_per_sqft = HousePrice / (StoreArea+BasementArea+LawnArea))
```

```
##   HOUSE_ID HousePrice StoreArea BasementArea LawnArea StreetHouseFront
## 1       23     155000       440          504     8816               80
## 2       35     235000       564          429    13500               NA
## 3      104     187500       444          568    12205               NA
## 4      111     277000       389          697   159000               NA
## 5      138     200500         0          152    18030              138
```

```
##      Location ConnectivityType    BuildingType ConstructionYear  EstateType
## 1 Dhaula Kuan            Byway individualhouse             1971 SemiPrivate
## 2 Dhaula Kuan            Byway individualhouse             1960       Other
## 3 Dhaula Kuan            Byway individualhouse             1966       Other
## 4 Dhaula Kuan            Byway individualhouse             1958       Other
## 5 Dhaula Kuan            Byway individualhouse             1946 SemiPrivate
##   SellingYear Rating SaleType StreetHouseFront2                      details
## 1        2010      6 NewHouse                80 Dhaula Kuan, IndividualHouse
## 2        2008      6 NewHouse                70 Dhaula Kuan, IndividualHouse
## 3        2007      6 NewHouse                70 Dhaula Kuan, IndividualHouse
## 4        2007      6 NewHouse                70 Dhaula Kuan, IndividualHouse
## 5        2007      5 NewHouse               138 Dhaula Kuan, IndividualHouse
##   EstateTypeNew SaleType2
## 1  Semi-Private       New
## 2         Other       New
## 3         Other       New
## 4         Other       New
## 5  Semi-Private       New
```

Explode Pipe (%$%):

Apply calculations directly between specific columns.

```r
house_sale %$%
  cor(HousePrice, BasementArea)  # Finds the correlation between HousePrice and BasementArea
```

```
## [1] 0.3939375
```

Assignment Pipe (%<>%):

Mutate house_sale directly without reassigning each time.

```r
house_location %<>%
  mutate(price_per_sqft2 = HousePrice / BasementArea)

print(head(house_location$price_per_sqft2))
```

```
## [1]  307.5397  547.7855  330.1056  397.4175 1319.0789
```

Chapter 15: Functions

Functions in R help automate repetitive tasks, making your code cleaner and easier to maintain.

Writing functions examples

Below, we are trying to calculate squares of numbers and returning result as a string.

```r
square_number <- function(x, as_string = FALSE) {
  result <- x^2
  if (as_string) {
    return(paste("The square of", x, "is", result))
  }
  return(result)
}
print(square_number(10,TRUE))
```

```
## [1] "The square of 10 is 100"
```

We can also create a function to check if a number is positive or negative.

```r
check_number <- function(x) {
  if (x > 0) {
    return("Positive")
  } else if (x < 0) {
    return("Negative")
  } else {
    return("Zero")
  }
}

check_number(10)
```

```
## [1] "Positive"
```

```r
check_number(0)
```

```
## [1] "Zero"
```

```r
check_number(-50)
```

```
## [1] "Negative"
```

Dot-Dot-Dot (...) for Flexible Functions

```r
combine_strings <- function(...) {
  return(paste(..., collapse = " "))
}

# Using the function
print(combine_strings("We", "can't", "overlook", "the", "effect", "of", "game", "theory"))  #
```

```
## [1] "We can't overlook the effect of game theory"
```

Lazy Evaluation

this is a function that only evaluates its arguments when needed.

```r
lazy_evaluation_example <- function(x) {
  if (x == 0) {
    return("Zero is not allowed")
  }
  return(1/x)
}

# Using the function
print(lazy_evaluation_example(0))
```

```
## [1] "Zero is not allowed"
```

```r
print(lazy_evaluation_example(20))
```

```
## [1] 0.05
```

Returning Values and Early Returns

The below function that sums two numbers but returns early if one of them is zero.

```r
sum_numbers <- function(a, b) {
  if (a == 0 || b == 0) {
    return(0)
  }
  return(a + b)
}

# Using the function
print(sum_numbers(5, 3))
```

```
## [1] 8
```

```r
print(sum_numbers(0, 3))
```

```
## [1] 0
```