**0.2 R 스튜디오 화면 구성**

처음으로 R 스튜디오를 실행하면 다음과 같은 화면으로 구성되어 있습니다. 이 중 소스 창을 열기 위해 네모 2개가 겹쳐 있는 모양()의 버튼을 클릭합니다.

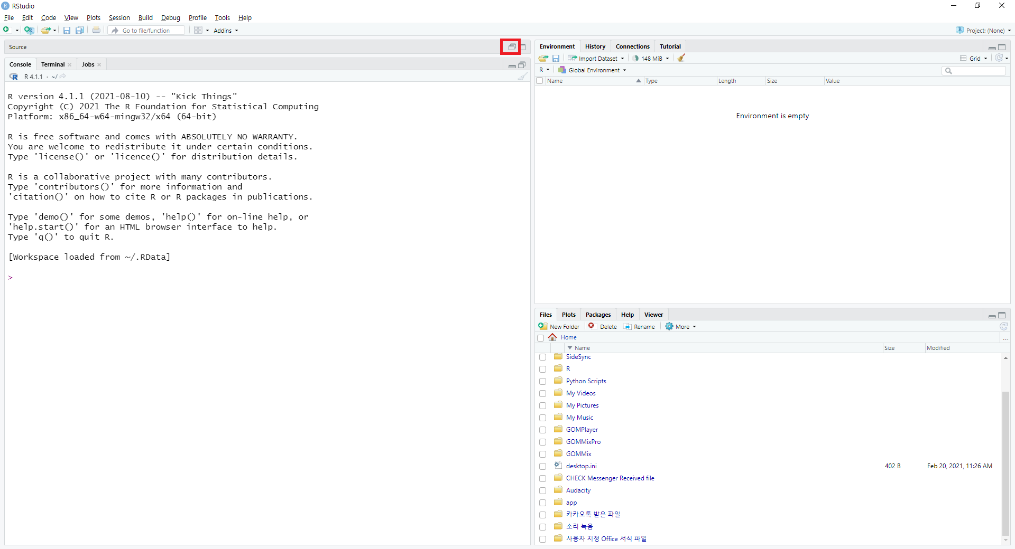


그림 0.5: R 스튜디오 화면구성

소스 창이 활성화되면 총 4개의 창으로 화면이 구성되며, 각 창의 크기는 경계선 부분을 드래그하여 조절할 수 있습니다.

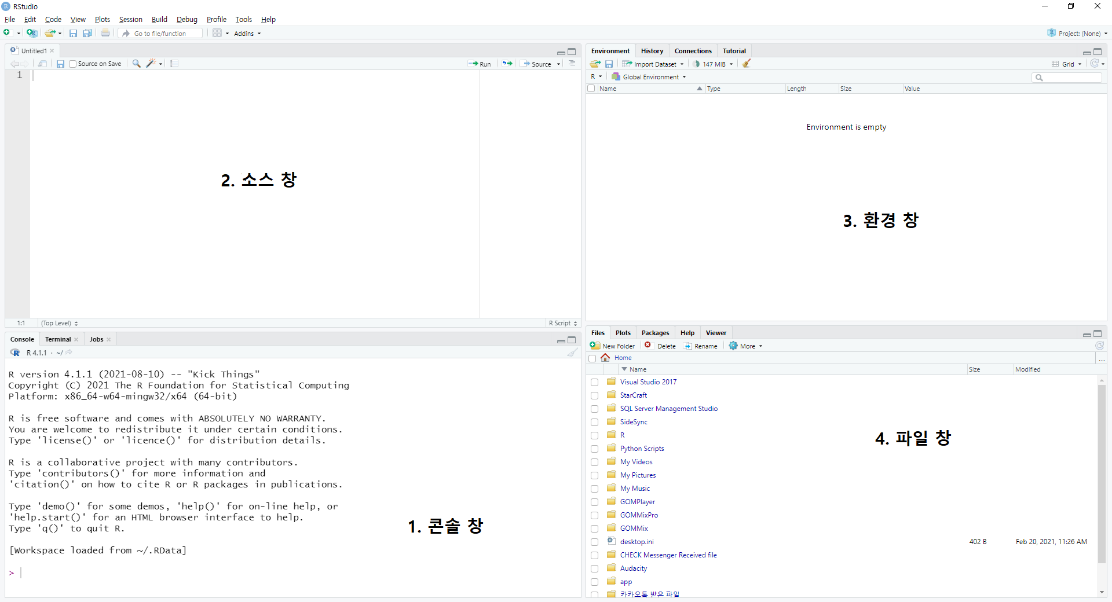


그림 0.6: R 스튜디오 화면구성 (2)

**1. 콘솔 창**

좌측 하단에 있는 콘솔 창은 코드를 입력하고 결과물을 출력하는 곳입니다.

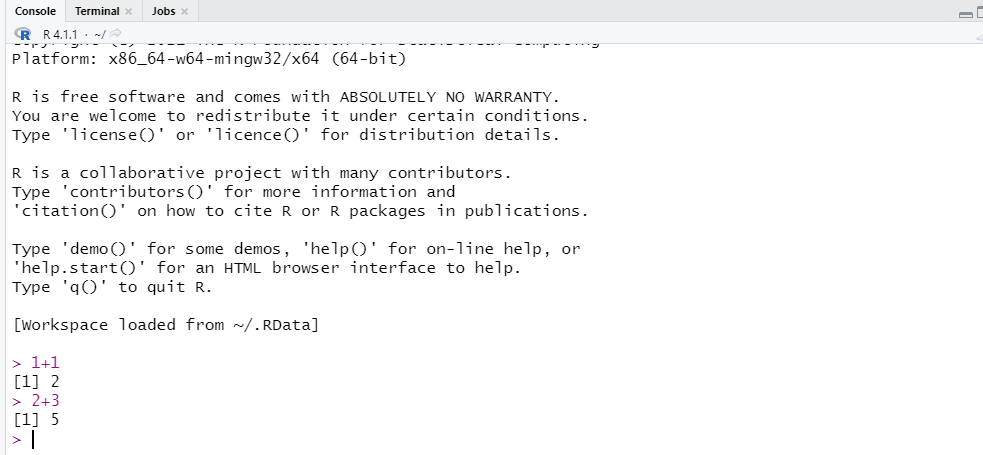


그림 0.7: R 스튜디오 콘솔 창

콘솔 창의 > 기호 뒤에 **1+1**을 입력하면 그 결과값인 **2**가 출력됩니다.

이 외에도 [Terminal] 탭에서는 시스템 쉘을 이용해 운영 체제를 조작할 수 있습니다.

**2. 소스 창**

좌측 상단에 있는 소스 창은 코드를 기록할 수 있는 공간이며, 이를 저장한 파일을 스크립트라고 합니다. 콘솔 창과는 다르게 코드를 입력하여도 바로 실행이 되지 않으며, 엔터를 누르면 행이 바뀝니다. 실행하고자 하는 코드가 있는 행을 선택한 후, [Ctrl + Enter]키를 누르면 해당 코드가 실행됩니다.

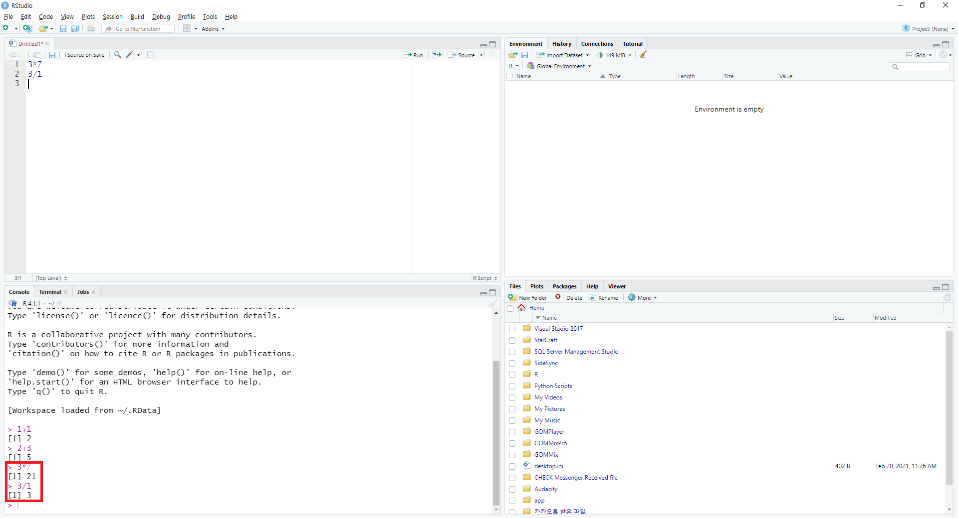


그림 0.8: R 스튜디오 소스 창의 실행

3\*7이란 코드가 있는 곳에 커서를 둔 후 [Ctrl + Enter]키를 누르면 해당 코드가 콘솔 창에서 실행됩니다. 만일 여러줄의 명령어를 한번에 실행하고 할 경우, 원하는 부분의 코드를 마우스로 드래그하여 선택한 후 [Ctrl + Enter]키를 누르면 코드가 순차적으로 콘솔 창에 입력되면서 실행됩니다.

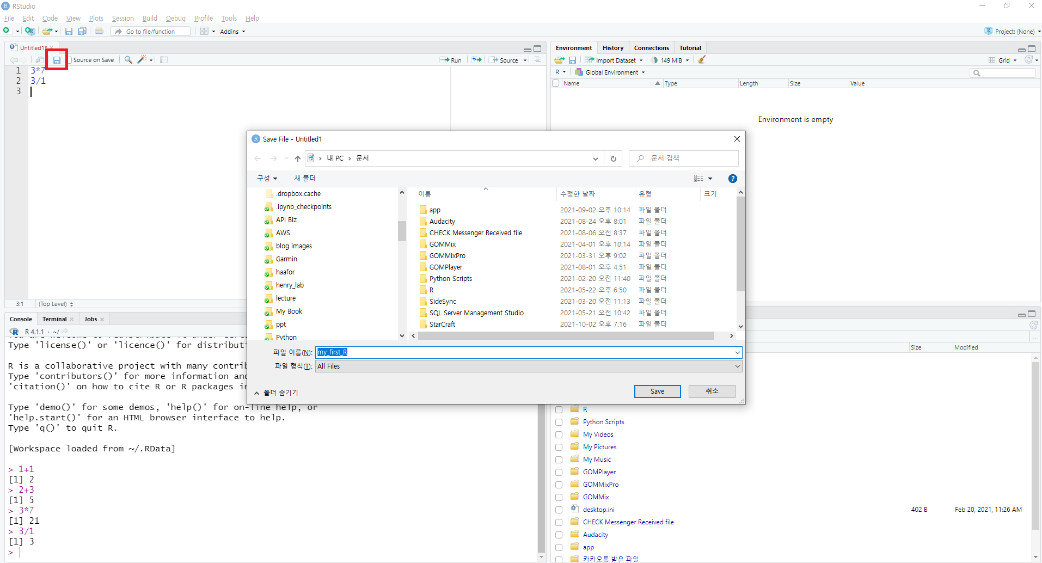


그림 0.9: R 스튜디오 소스 저장하기

위에서 작성한 코드를 저장해보도록 하겠습니다. 저장 버튼()을 클릭한 후 원하는 폴더 및 파일 이름을 입력 한 후 [Save] 버튼을 누릅니다.

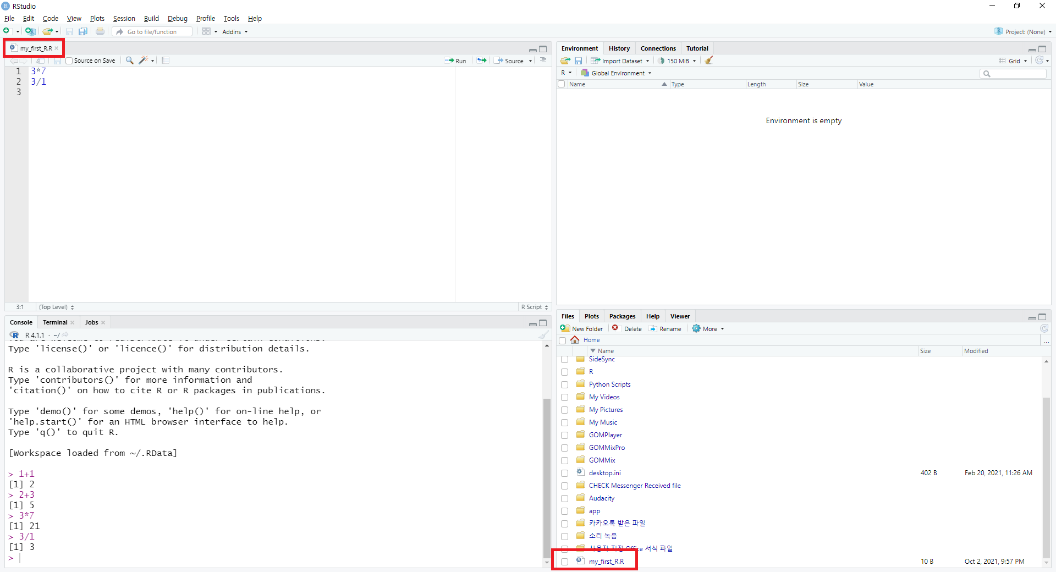


그림 0.10: R 스튜디오 소스 저장하기 (2)

Untitled1로 되어 있던 스크립트의 이름이 저장한 이름으로 바뀌며, 스크립트가 저장되어 있는 것이 확인됩니다. 이처럼 코딩을 한 후 스크립트를 저장할 경우, 나중에 해당 내역을 그대로 불러올 수 있습니다.

**3. 환경 창**

우측 상단에 있는 환경 창은 생성된 데이터를 보여주는 화면입니다.

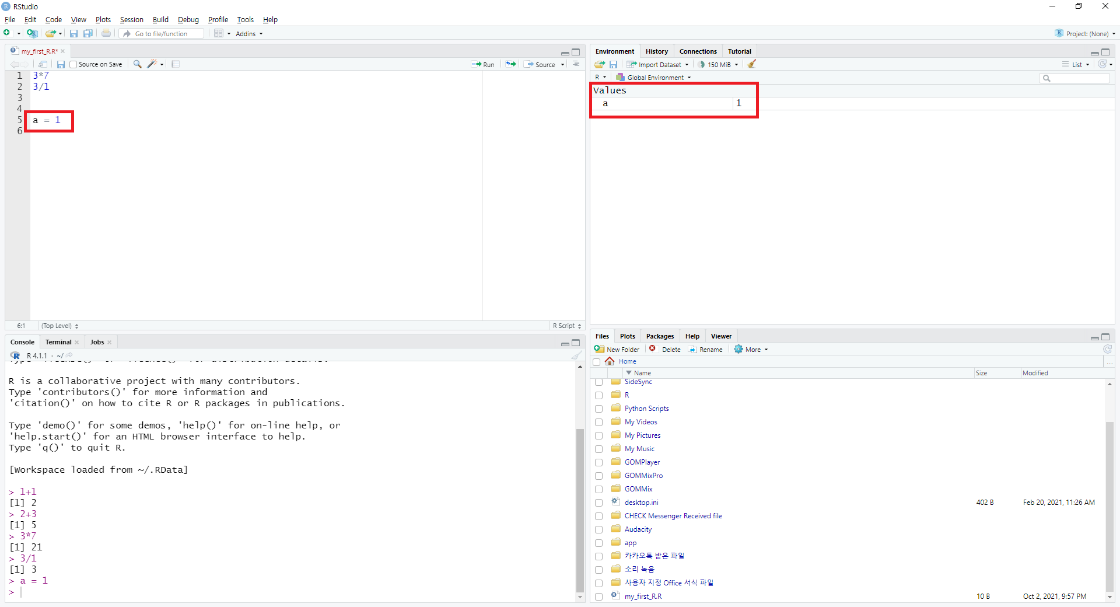


그림 0.11: R 스튜디오 환경 창

스크립트 창에서 **a = 1**을 입력하면, 환경 창의 Values 목록에 a가 생기며 그 값은 1로 표시됩니다.

이 외에도 환경 창의 [History] 탭은 이제까지 실행했던 코드의 내역을 볼 수 있으며, [Connections] 탭은 SQL이나 Spark 등 데이터베이스와의 연결을 도와줍니다.

**4. 파일 창**

우측 하단에 있는 파일 창은 윈도우의 파일 탐색기와 비슷한 역할을 하며, 워킹 디렉터리 내의 파일을 보여줍니다. 이 외에도 [Plots] 탭은 그래프를 보여주며 [Packages] 탭은 설치된 패키지의 목록을 보여줍니다. [Help] 탭은 도움말을 보여주며, [Viewer] 탭은 분석 결과를 HTML 등 웹 문서로 출력한 모습을 보여줍니다.

**0.3 R 스튜디오 설정하기**

R 스튜디오는 기본적으로 흰색 바탕에 검은색 글씨로 설정되어 있습니다. 그러나 흰 화면에서 작업을 하게 되면 눈이 쉽게 피로해지며, 시력에도 좋지 않습니다. 이를 방지하기 위해 어두운 화면으로 설정을 변경해주는 것이 좋습니다.

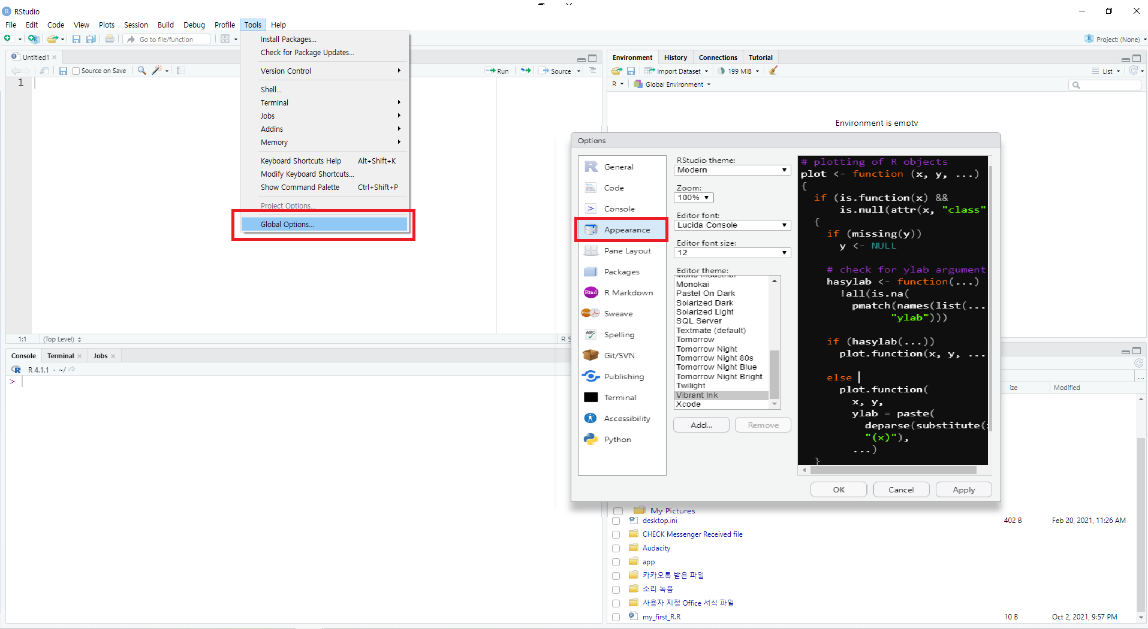


그림 0.12: R 스튜디오의 테마 선택

상단 탭에서 [Tools] → [Global Options]를 선택한 후, Options의 [Appearance] 탭의 [Editor theme]을 통해 각종 테마를 적용할 수 있습니다. 이 중 본인의 마음에 드는 테마를 선택한 후, [OK] 버튼을 누릅니다.

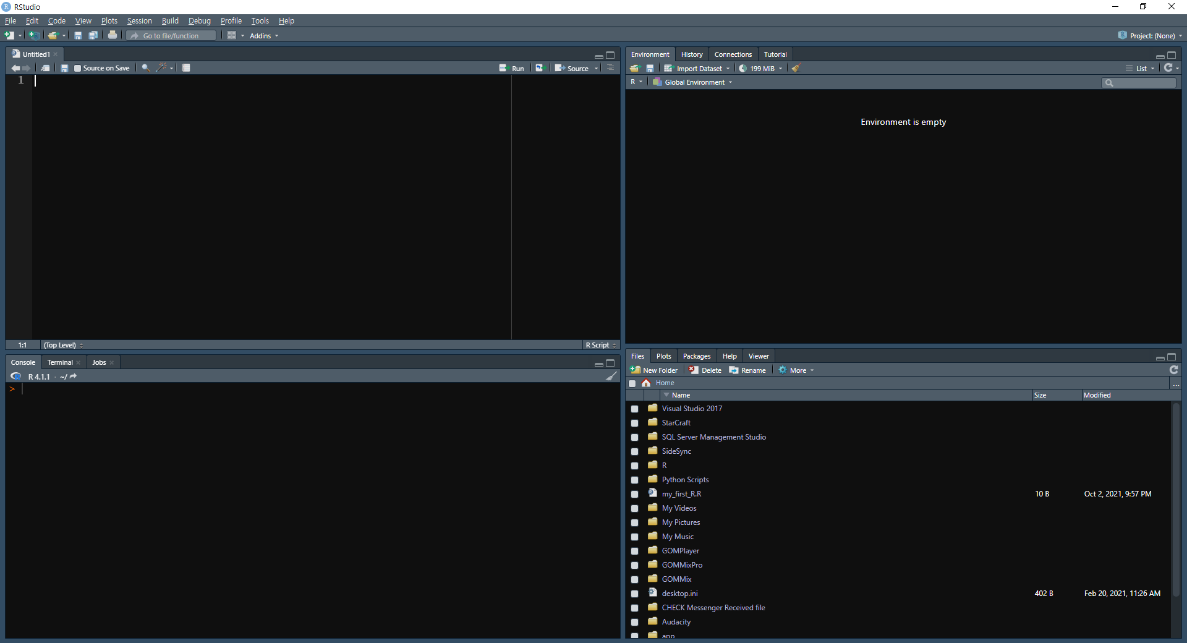


그림 0.13: R 스튜디오의 다크모드 적용

화면의 배경이 어두워져 눈이 한결 편해졌습니다.

또한 스크립트 내용 중 한글이 깨지는 것을 방지하기 위해 인코딩 방식을 설정할 필요도 있습니다.

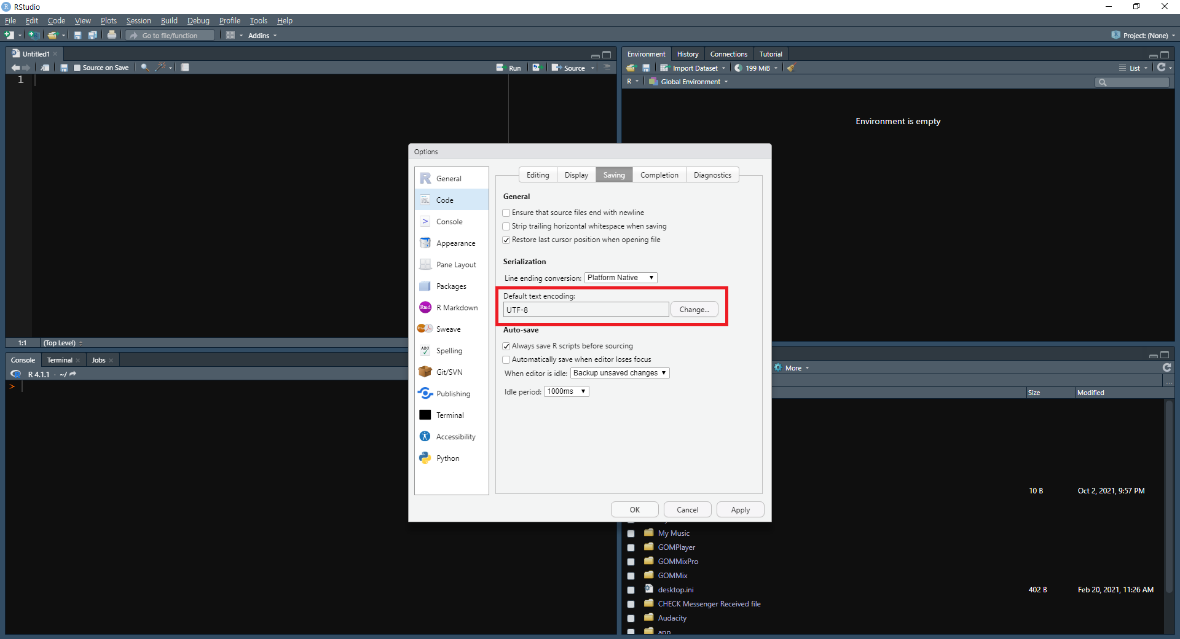


그림 0.14: R 스튜디오의 인코딩 변경

상단 탭에서 [Tools] → [Global Options]를 선택한 후, Options의 [Code] 탭의 [Default text encoding] 부분의 [Change]를 눌러 **UTF-8**로 변경해줍니다.

인코딩은 컴퓨터가 문자를 표현하는 방식을 의미하며, 이에 대해서는 나중 장에서 다시 자세하게 다루도록 합니다.

해당 방법으로도 스크립트의 인코딩이 깨질 경우 [File → Reopen with Encoding] 메뉴에서 [UTF-8] 항목을 선택하고 [Set as default encoding for source files] 항목을 선택한 후 [OK]를 클릭합니다. UTF-8로 인코딩이 설정된 후 파일을 다시 열리게 됩니다.

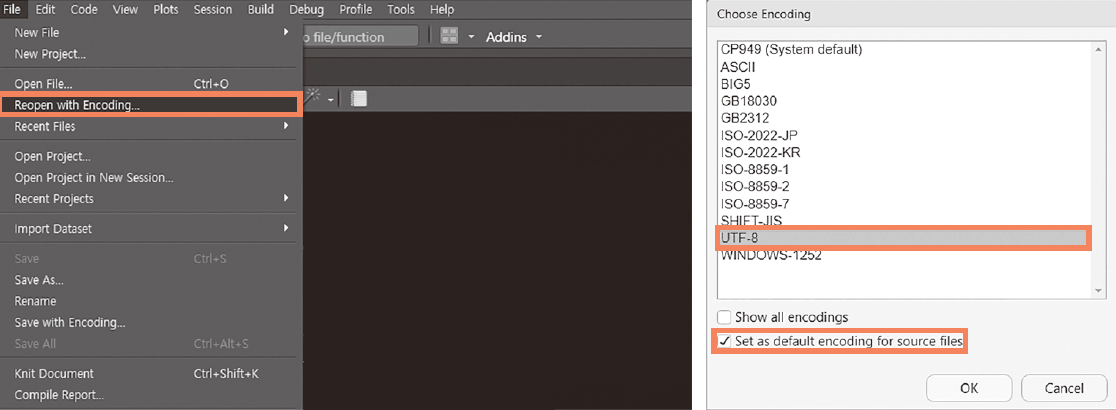


그림 0.15: 인코딩 변경 후 재시작

**0.4 프로젝트 만들기**

R 스튜디오에서 코딩을 하기 전에 프로젝트(Project)를 만들면 하나의 프로젝트에 사용되는 소스 코드, 이미지, 문서 등의 파일을 폴더별로 관리하여 효율적으로 관리할 수 있습니다.

먼저 R 스튜디오 상단의 육각형 모양 버튼()을 클릭하거나 [File → New Project]를 클릭합니다.

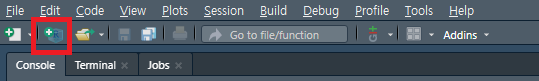


그림 0.16: 프로젝트 생성

[Create Project] 화면에서 가장 상단의 [New Directory]를 클릭합니다. 참고로 Existing Directory는 기존 폴더에 새로운 프로젝트를 만들때, Version Control은 깃허브 등의 버전 관리 시스템을 이용할 때 사용됩니다.

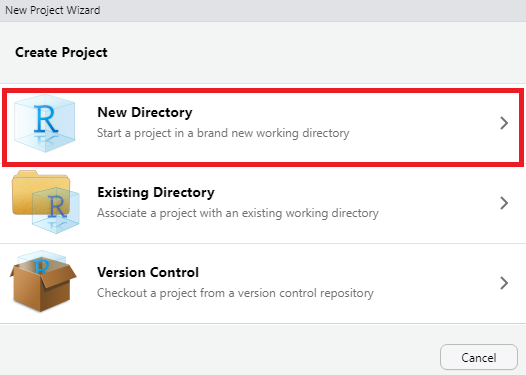


그림 0.17: 새 프로젝트 생성

[Project Type]에서 가장 상단의 [New Project]를 클릭합니다.

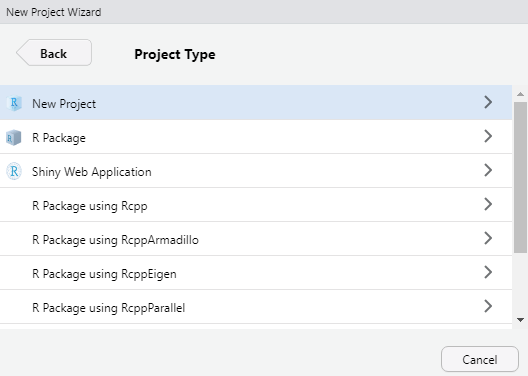


그림 0.18: 새 프로젝트 생성 (2)

[Create New Project] 창에서 [Directory name] 항목에 새로 만들 프로젝트 이름을 입력합니다. [Create project as subdirectory of] 항목에는 프로젝트 폴더를 만들 위치를 선택하며, [Browse]를 클릭해 원하는 위치를 선택합니다. 그 후 하단의 [Creage Project]를 클릭합니다.

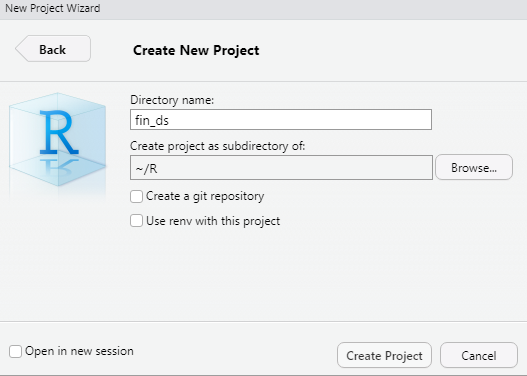


그림 0.19: 새 프로젝트 생성 (3)

R 스튜디오가 재시작되면 우측 상단 부분이 프로젝트 이름으로 바뀌며, 파일 창의 윗부분도 프로젝트 폴더의 위치로 바뀝니다. 또한 폴더 내에 fin\_ds.Rpoj 라는 파일이 생성됩니다. 스크립트 및 각종 파일들을 해당 프로젝트 폴더에 저장하여, 효율적으로 각종 작업을 관리할 수 있습니다.

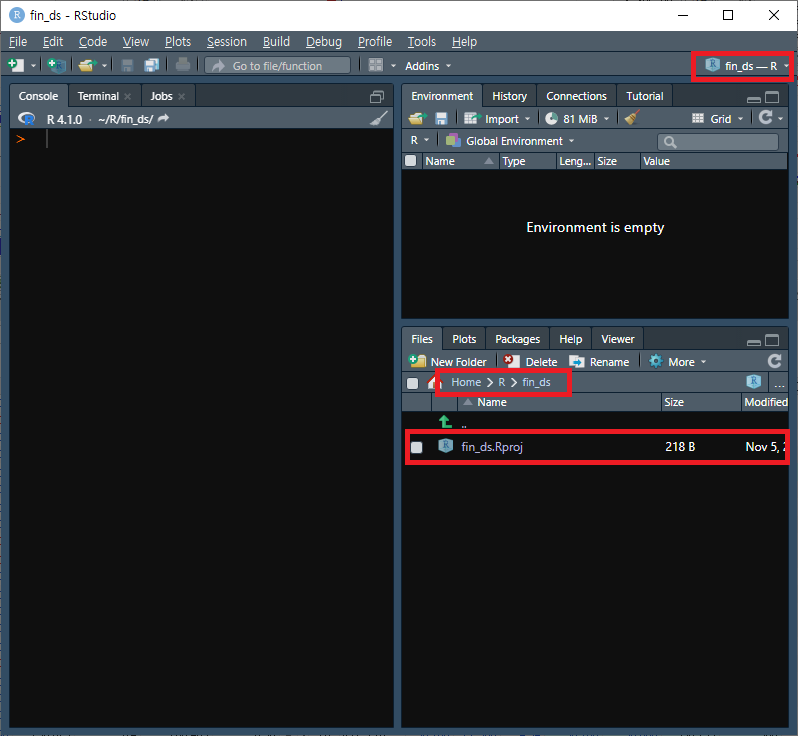


그림 0.20: 새 프로젝트 생성 (4)

프로젝트 이름과 폴더 경로에 한글이 들어가면 오류가 발생할 수 있으니, 영문으로 입력하는 것이 좋습니다.

**0.5 데이터 타입별 다루기**

R과 R 스튜디오 설치가 끝났으면 본격적으로 R의 기본적인 사용법에 대해 배워보겠으며, 먼저 데이터의 타입별로 다루는 법부터 시작하겠습니다.

R 뿐만 아니라 각종 프로그래밍에는 여러가지 데이터 타입이 있으며, 이를 다루는 방법은 각각 다릅니다. 예를 들어 같은 ’3’도 숫자 3인지 문자 3인지에 따라 다루는 방법이 다릅니다. 따라서 데이터 타입의 종류와 이들을 어떻게 다루어야 하는지를 아는 것이 프로그래밍의 기초라고 할 수 있습니다.

**0.5.1 숫자 형태**

R에서 숫자(Numbers) 형태는 크게 integer와 double로 나눌 수 있습니다. 이 중 integer는 정수를 의미하며, double은 부동소수점 실수를 의미합니다.

dbl\_var = c(1, 2.5, 4.5)

print(dbl\_var)

## [1] 1.0 2.5 4.5

위와 같이 입력하면 double, 즉 소수점 형태의 숫자가 만들어 집니다.

int\_var = c(1L, 6L, 5L)

print(int\_var)

## [1] 1 6 5

만일 숫자 뒤에 **L**을 붙이면, integer(정수) 형태의 숫자가 만들어 집니다.

double 형태를 integer 형태로 바꾸려면 할 경우, as.integer() 함수를 사용해 쉽게 변경할 수 있습니다. 이처럼 R에서는 as.\*() 함수의 형태로 각 데이터의 형태를 바꿀 수 있습니다.

as.integer(dbl\_var)

## [1] 1 2 4

[1.0 2.5 4.5] 이던 dbl\_var 값이 as.integer() 함수를 통해 소수점이 사라지고 정수 형태인 [1 2 4]로 변경되었습니다.

**0.5.1.1 숫자 생성하기**

R에서는 콜론(:)과 c() 함수를 통해 순서가 있는 숫자 벡터를 생성할 수 있습니다.

1:10

## [1] 1 2 3 4 5 6 7 8 9 10

**시작숫자:끝숫자**의 형태로 입력하여 1에서 10까지 숫자가 생성됩니다.

c(1, 5, 10)

## [1] 1 5 10

c() 함수 내부에 각각의 숫자를 입력할 경우, 이로 구성된 숫자 벡터가 생성됩니다.

seq() 함수를 이용할 경우 더욱 다양하게 숫자 벡터를 생성할 수 있습니다. seq는 Sequence 즉 ’순서’의 약어입니다. 이처럼 R이나 여타 프로그래밍에서는 함수의 이름을 통해 대략적인 기능을 추론할 수 있습니다.

seq(from = 1, to = 21, by = 2)

## [1] 1 3 5 7 9 11 13 15 17 19 21

seq() 함수 내부에 from에는 시작 숫자, to에는 종료 숫자, by에는 간격을 입력합니다. 즉 1에서 21까지 2 단위로 숫자가 생성됩니다.

seq(0, 21, length.out = 15)

## [1] 0.0 1.5 3.0 4.5 6.0 7.5 9.0 10.5 12.0 13.5 15.0 16.5 18.0 19.5 21.0

만일 입력값에 **by** 대신 **length.out**을 쓸 경우 from에서 to 까지 동일한 증가폭으로 length.out 만큼의 숫자를 생성하며, 해당 예제에서는 총 15개의 숫자가 만들어집니다.

rep() 함수 역시 숫자를 생성해주는 함수입니다.

rep(1:4, times = 2)

## [1] 1 2 3 4 1 2 3 4

rep는 Replicate 즉 ’복제하다’의 약어 입니다. 해당 함수 내에 **times**라는 입력값을 추가해줄 경우, 해당 숫자만큼 반복되어 벡터가 생성됩니다.

rep(1:4, each = 2)

## [1] 1 1 2 2 3 3 4 4

만일 **each**라는 입력값을 추가할 경우, 각각의 숫자를 n번 반복하여 벡터가 생성됩니다.

**0.5.1.2 올림, 내림, 반올림**

함수를 통해 간단하게 숫자의 올림, 내림, 반올림을 할 수도 있습니다. 먼저 다음과 같이 숫자를 입력합니다.

x = c(1, 1.35, 1.7, 2.053, 2.4, 2.758, 3.1, 3.45,

3.8, 4.15, 4.5, 4.855, 5.2, 5.55, 5.9)

round(x)

## [1] 1 1 2 2 2 3 3 3 4 4 4 5 5 6 6

round() 함수는 가장 가까운 정수로 반올림을 합니다.

round(x, digits = 2)

## [1] 1.00 1.35 1.70 2.05 2.40 2.76 3.10 3.45 3.80 4.15 4.50 4.86 5.20 5.55 5.90

함수 내부에 **digits** 입력값을 추가해 줄 경우, 해당 자리수 만큼 반올림을 합니다. 위 예제에서는 소수 둘째자리 만큼 반올림을 하였습니다.

ceiling(x)

## [1] 1 2 2 3 3 3 4 4 4 5 5 5 6 6 6

floor(x)

## [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5

ceiling() 함수는 올림을, floor() 함수는 내림을 실행합니다.

**0.5.2 문자열 형태**

일반적인 글자 혹은 텍스트를 문자열(Character Strings)이라고 합니다.

a = 'learning to create'

b = 'character strings'

paste(a, b)

## [1] "learning to create character strings"

먼저 a와 b 변수에 각각의 문자를 입력한 후, R의 기본함수인 paste() 함수를 이용해 두 문자를 붙일 수 있습니다.

print(pi)

## [1] 3.142

paste('pi is', pi)

## [1] "pi is 3.14159265358979"

원주율을 의미하는 pi는 원래 3.14159 라는 숫자가 입력되어 있습니다. 그러나 paste() 함수를 통해 문자열과 숫자를 합칠 경우, 그 결과값은 문자열이 됩니다.

paste('I', 'love', 'R', sep = ',')

## [1] "I,love,R"

paste() 함수 내부에 sep 인자를 추가할 경우, 각 단어를 구분하는 문자를 입력할 수 있습니다. 기존에는 각 문자가 공백을 기준으로 합쳐졌다면, 이번에는 콤마(,)를 기준으로 합쳐졌습니다.

paste0('I', 'love', 'R')

## [1] "IloveR"

paste0() 함수는 구분 문자가 없이 결합됩니다.

**0.5.2.1 stringr 패키지를 이용한 문자열 다루기**

R의 기본함수를 이용하여도 문자열을 다룰 수 있지만, stringr 패키지를 이용할 경우 더욱 다양한 작업을 수행할 수 있습니다.

library(stringr)

str\_c('Learning', 'to', 'use', 'the', 'stringr', 'package', sep = ' ')

## [1] "Learning to use the stringr package"

str\_c() 함수는 paste() 함수와 기능이 동일하며, sep 인자를 통해 구분자를 추가할 수 있습니다.

text = c('Learning', 'to', NA, 'use', 'the', NA, 'stringr', 'package')

str\_length(text)

## [1] 8 2 NA 3 3 NA 7 7

str\_length() 함수는 문자열 각각의 텍스트 갯수를 세줍니다.

x = 'Learning to use the stringr package'

str\_sub(x, start = 1, end = 15)

## [1] "Learning to use"

str\_sub(x, start = -7, end = -1)

## [1] "package"

str\_sub() 함수는 start부터 end까지의 문자를 출력합니다. 만일 start 혹은 end에 음수를 입력하면, 문장의 뒤에서부터 start/end 지점이 계산됩니다. 즉, start와 end에 각각 -7와 -1을 입력하면 끝에서부터 일곱번째와 첫번째 지점이 시작점과 끝점이 됩니다.

텍스트 데이터를 다룰때는 빈 공백이 따라오는 경우가 많으며, 이는 대부분 제거해주어야 할 대상입니다.

text = c('Text ', ' with', ' whitespace ', ' on', 'both ', 'sides ')

print(text)

## [1] "Text " " with" " whitespace " " on" "both "

## [6] "sides "

각 단어를 자세히 살펴보면 좌/우 혹은 양쪽에 공백이 있습니다. 이를 제거하도록 하겠습니다.

str\_trim(text, side = 'left')

## [1] "Text " "with" "whitespace " "on" "both "

## [6] "sides "

str\_trim(text, side = 'right')

## [1] "Text" " with" " whitespace" " on" "both"

## [6] "sides"

str\_trim(text, side = 'both')

## [1] "Text" "with" "whitespace" "on" "both"

## [6] "sides"

str\_trim() 함수는 공백을 제거해주는 기능을 합니다. side 인자에 **left**를 입력할 경우 각 텍스트 왼쪽의 공백을, **right**를 입력할 경우 오른쪽의 공백을, **both**를 입력할 경우 양쪽의 공백을 제거해줍니다.

마지막으로 원하는 자리수를 채우기 위해 문자열에 공백 혹은 특정 문자를 입력할 수도 있으며, str\_pad() 함수를 통해 손쉽게 작업을 할 수 있습니다.

str\_pad('beer', width = 10, side = 'left')

## [1] " beer"

width에 해당하는 10자리를 맞추기 위해 side의 입력값인 좌측에 공백이 추가되었습니다.

str\_pad('beer', width = 10, side = 'left', pad = '!')

## [1] "!!!!!!beer"

pad 인자를 추가할 경우, 공백이 아닌 입력한 문자가 추가됩니다

아래 페이지에는 stringr 패키지의 자세한 사용법이 나와 있습니다.

https://stringr.tidyverse.org/

**0.5.3 날짜 형태**

시계열 작업을 위해서는 날짜(Date), 혹은 시간(Datetime) 형태를 다루어야 합니다.

Sys.timezone()

## [1] "Asia/Seoul"

Sys.Date()

## [1] "2021-11-20"

Sys.time()

## [1] "2021-11-20 16:43:47 KST"

Sys.timezone() 함수는 현재 타임존을 출력합니다. Sys.Date() 함수는 현재 날짜를, Sys.time() 함수는 날짜와 시간을 출력합니다.

’2018-12-31’과 같이 사용자가 보기에는 날짜 형태이지만 문자열 형태로 데이터가 들어오는 경우, 이를 날짜 형태로 변경해야 할 경우가 있습니다.

x = c('2021-07-01', '2021-08-01', '2021-09-01')

x\_date = as.Date(x)

str(x\_date)

## Date[1:3], format: "2021-07-01" "2021-08-01" "2021-09-01"

as.Date() 함수를 이용하면 문자열을 손쉽게 날짜 형태로 변경할 수 있습니다. str() 함수는 데이터의 형태를 확인하는 함수로써, Date 형태임이 확인됩니다.

y = c('07/01/2015', '08/01/2015', '09/01/2015')

as.Date(y, format = '%m/%d/%Y')

## [1] "2015-07-01" "2015-08-01" "2015-09-01"

YYYY-MM-DD 형태가 아닌 다른 형태(MM/DD/YYYY)로 입력된 경우, format을 직접 입력하여 Date 형태로 변경할 수 있습니다.

YYYY는 연, MM은 월, DD는 일을 나타냅니다.

**0.5.3.1 lubridate 패키지를 이용한 날짜 다루기**

lubridate 패키지를 이용할 경우 날짜 형태와 관련된 다양한 작업을 수행할 수 있습니다.

library(lubridate)

x = c('2021-07-01', '2021-08-01', '2021-09-01')

y = c('07/01/2015', '08/01/2015', '09/01/2015')

ymd(x)

## [1] "2021-07-01" "2021-08-01" "2021-09-01"

mdy(y)

## [1] "2015-07-01" "2015-08-01" "2015-09-01"

lubridate 패키지를 이용할 경우 YYYY-MM-DD 형태는 ymd(), MM-DD-YYYY 형태는 mdy() 함수를 사용해 손쉽게 Date 형태로 변경할 수 있습니다. 이 외에도 lubridate에는 Date 형태로 변경하기 위한 다양한 함수가 존재합니다.

| 표 0.1: lubridate 패키지의 Date 형태 변경 함수 | |
| --- | --- |
| **순서** | **함수** |
| 연, 월, 일 | ymd() |
| 연, 일, 월 | ydm() |
| 월, 일, 연 | mdy() |
| 일, 월, 연 | dmy() |
| 시, 분 | hm() |
| 시, 분, 초 | hms() |
| 연, 월, 일, 시, 분, 초 | ymd\_hms() |

lubridate 패키지에는 날짜 관련 정보를 추출할 수 있는 다양한 함수가 존재합니다.

| 표 0.2: lubridate 패키지의 날짜 관련 정보 추출 함수 | |
| --- | --- |
| **정보** | **함수** |
| 연 | year() |
| 월 | month() |
| 주 | week() |
| 연도 내 일수 | yday() |
| 월 내 일수 | mday() |
| 주 내 일수 | wday() |
| 시 | hour() |
| 분 | minute() |
| 초 | second() |
| 타임존 | tz() |

x = c('2021-07-01', '2021-08-01', '2021-09-01')

year(x)

## [1] 2021 2021 2021

month(x)

## [1] 7 8 9

week(x)

## [1] 26 31 35

year(), month(), week() 함수를 통해 년도, 월, 주 정보를 확인할 수 있습니다.

z = '2021-09-15'

yday(z)

## [1] 258

mday(z)

## [1] 15

wday(z)

## [1] 4

yday(), mday(), wday() 함수는 각각 해당 년도에서 몇번째 일인지, 해당 월에서 몇번째 일인지, 해당 주에서 몇번째 일인지를 계산합니다.

x = ymd('2021-07-01', '2021-08-01', '2021-09-01')

x + years(1) - days(c(2, 9, 21))

## [1] "2022-06-29" "2022-07-23" "2022-08-11"

날짜에서 연과 월, 일자를 더하거나 빼는 계산 역시 가능합니다. 먼저 year() 함수를 통해 1년씩을 더 하였으며, days() 함수를 통해 각각의 일자 만큼을 뺍니다.

**0.5.3.2 날짜 순서 생성하기**

숫자와 마찬가지로 seq() 함수를 이용할 경우 날짜 벡터를 생성할 수 있습니다.

seq(ymd('2015-01-01'), ymd('2021-01-01'), by ='years')

## [1] "2015-01-01" "2016-01-01" "2017-01-01" "2018-01-01" "2019-01-01"

## [6] "2020-01-01" "2021-01-01"

2015년 1월 1일부터 2021년 1월 1일까지 1년을 기준으로 벡터가 생성됩니다.

seq(ymd('2021-09-01'), ymd('2021-09-30'), by ='2 days')

## [1] "2021-09-01" "2021-09-03" "2021-09-05" "2021-09-07" "2021-09-09"

## [6] "2021-09-11" "2021-09-13" "2021-09-15" "2021-09-17" "2021-09-19"

## [11] "2021-09-21" "2021-09-23" "2021-09-25" "2021-09-27" "2021-09-29"

지정한 일수인 2일 단위로 날짜 벡터를 생성할 수도 있습니다. 이 외에도 by 인자를 통해 원하는 기간 단위의 벡터를 생성할 수 있습니다.

아래 페이지에는 lubridate 패키지의 자세한 사용법이 나와 있습니다.

https://lubridate.tidyverse.org/

**0.6 데이터 구조 다루기**

R에서 자주 사용되는 데이터구조는 벡터(Vector), 리스트(List), 데이터프레임(Dataframe) 입니다.

**0.6.1 벡터 다루기**

벡터는 R의 가장 기본적인 데이터 구조로써 integer, double, logical, character로 이루어져 있습니다. 벡터를 만드는 방법에 대해서는 앞서 다루었습니다.

vec\_integer = 8:17

vec\_integer

## [1] 8 9 10 11 12 13 14 15 16 17

vec\_double = c(0.5, 0.6, 0.2)

vec\_double

## [1] 0.5 0.6 0.2

vec\_char = c('a', 'b', 'c')

vec\_char

## [1] "a" "b" "c"

integer의 경우 start:end 형태를 통해서, 그 외에는 c() 함수를 통해 벡터를 만들 수 있습니다.

c('a', 'b', 'c', 1, 2, 3)

## [1] "a" "b" "c" "1" "2" "3"

숫자와 문자가 같이 벡터로 묶일 경우, 숫자는 모두 문자 형태로 변경됩니다.

c(1, 2, 3, TRUE, FALSE)

## [1] 1 2 3 1 0

TRUE와 FALSE는 참 혹은 거짓을 나타내는 논리값(logical) 입니다. 숫자와 논리값이 같이 묶일 경우 TRUE는 1, FALSE는 0으로 치환된 후 숫자 형태로 변경됩니다.

c('a', 'b', 'c', TRUE, FALSE)

## [1] "a" "b" "c" "TRUE" "FALSE"

문자와 논리값이 같이 묶일 경우 모두 문자 형태로 변경됩니다. 이처럼 문자와 다른 형태가 묶일 경우엔 모든 데이터가 문자로 변경됩니다.

이번에는 기존의 벡터에 새로운 값을 추가해보겠습니다.

v1 = 8:17

c(v1, 18:22)

## [1] 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

기존 8부터 17까지의 숫자로 이루어진 v1 벡터에, c() 함수를 이용하여 새로운 값을 추가할 수 있습니다.

만약 벡터에서 원하는 부분의 데이터를 추출하려면 대괄호([])를 이용하면 됩니다.

v1[2]

## [1] 9

v1[2:4]

## [1] 9 10 11

v1[c(2, 4, 6)]

## [1] 9 11 13

대괄호 안에 숫자를 입력하면, 벡터에서 해당 순서의 데이터가 추출됩니다. **c(2,4,6)**과 같이 특정 위치를 지정하여 데이터를 추출할 수도 있습니다.

v1[-1]

## [1] 9 10 11 12 13 14 15 16 17

v1[-c(2, 4, 6, 8)]

## [1] 8 10 12 14 16 17

마이너스 기호를 입력하면, 해당 순서를 제외한 데이터가 추출됩니다.

v1 < 12

## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

v1[v1 < 12]

## [1] 8 9 10 11

v1[v1 < 12 | v1 > 15]

## [1] 8 9 10 11 16 17

먼저 **v1 < 12**를 입력하면 해당 조건에 해당하는 부분은 TRUE, 그렇지 않은 부분은 FALSE를 반환합니다. 그 후 대괄호 안에 다시 결과를 입력하면 TRUE에 해당하는 순서의 데이터만 반환합니다. 이처럼 대괄호 내부에 조건을 설정하여 원하는 데이터를 추출할 수도 있습니다.

**0.6.2 리스트 다루기**

먼저 리스트를 생성합니다.

l = list(1:3, 'a', c(TRUE, FALSE, TRUE), c(2.5, 4.2))

str(l)

## List of 4

## $ : int [1:3] 1 2 3

## $ : chr "a"

## $ : logi [1:3] TRUE FALSE TRUE

## $ : num [1:2] 2.5 4.2

첫번째 원소는 정수(int), 두번째 원소는 문자(chr), 세번째 원소는 논리값(logi), 네번째 원손,ㄴ 숫자(num)으로 이루어져 있습니다. 이처럼 리스트는 각 원소간 타입이나 길이가 달라도 데이터가 결합할 수 특징이 있습니다.

l2 = list(1:3, list(letters[1:5], c(TRUE, FALSE, TRUE)))

str(l2)

## List of 2

## $ : int [1:3] 1 2 3

## $ :List of 2

## ..$ : chr [1:5] "a" "b" "c" "d" ...

## ..$ : logi [1:3] TRUE FALSE TRUE

위 예제에서 두번째 원소는 리스트로 구성되어 있습니다. 이처럼 리스트 내에 또 다른 리스트를 생성하는 것 역시 가능합니다.

이번에는 기존 리스트에 새로운 원소를 추가하도록 하겠습니다.

l3 = list(1:3, 'a', c(TRUE, FALSE, TRUE))

l4 = append(l3, list(c(2.5, 4.2)))

print(l4)

## [[1]]

## [1] 1 2 3

##

## [[2]]

## [1] "a"

##

## [[3]]

## [1] TRUE FALSE TRUE

##

## [[4]]

## [1] 2.5 4.2

append() 함수를 이용하면 기존 리스트에 추가로 원소를 붙일 수 있습니다.

l4$item5 = 'new list item'

print(l4)

## [[1]]

## [1] 1 2 3

##

## [[2]]

## [1] "a"

##

## [[3]]

## [1] TRUE FALSE TRUE

##

## [[4]]

## [1] 2.5 4.2

##

## $item5

## [1] "new list item"

또한 기존 리스트에 달러 사인($)을 입력할 경우, 원소에 이름이 생성되며 데이터가 추가됩니다.

리스트에서 원하는 데이터를 추출할 때는, 벡터와 동일하게 대괄호를 이용하면 됩니다.

l4[1]

## [[1]]

## [1] 1 2 3

l4[c(1,3)]

## [[1]]

## [1] 1 2 3

##

## [[2]]

## [1] TRUE FALSE TRUE

원소에 이름이 있을 경우, 이를 이용해 추출도 가능합니다.

l4['item5']

## $item5

## [1] "new list item"

원소의 이름인 **item5**를 입력하면 해당 원소만 반환합니다.

l4[[1]]

## [1] 1 2 3

l4$item5

## [1] "new list item"

대괄호를 두번, 혹은 달러 사인($)을 이용해 데이터를 추출할 경우 원소 내의 형태가 반환되며, 위의 예제들과는 다르게 벡터 형태가 반환되었습니다.

l4[[1]]

## [1] 1 2 3

l4[[1]][3]

## [1] 3

특정 원소의 항목을 추출하기 위해서는 [[와 [를 함께 사용합니다. 위 예제는 l4 리스트의 첫번째 원소에서 3번째 항목을 추출하게 됩니다.

**0.6.3 데이터프레임 다루기**

데이터프레임은 R에서 가장 널리 사용되는 형식으로써, 각 컬럼이 다른 형태를 가질 수 있습니다.

df = data.frame (col1 = 1:3,

col2 = c ("this", "is", "text"),

col3 = c (TRUE, FALSE, TRUE),

col4 = c (2.5, 4.2, pi))

str(df)

## 'data.frame': 3 obs. of 4 variables:

## $ col1: int 1 2 3

## $ col2: chr "this" "is" "text"

## $ col3: logi TRUE FALSE TRUE

## $ col4: num 2.5 4.2 3.14

col1은 숫자(int), col2는 문자(chr), col3는 논리연산자(logi), col4는 숫자(num)로 구성되어 있습니다. 또한 벡터 혹은 리스트를 이용해 데이터프레임을 생성할 수도 있습니다.

v1 = 1:3

v2 = c ("this", "is", "text")

v3 = c (TRUE, FALSE, TRUE)

data.frame(col1 = v1, col2 = v2, col3 = v3)

## col1 col2 col3

## 1 1 this TRUE

## 2 2 is FALSE

## 3 3 text TRUE

v1, v2, v3는 각각 숫자, 문자, 논리연산로 구성된 벡터입니다. 이를 data.frame() 함수 내에 입력할 경우 col1, col2, col3 열 이름에 해당 데이터들이 입력됩니다. 주의해야 할 점은 각 벡터의 길이가 동일해야 데이터프레임 형태를 만들 수 있습니다.

l = list (item1 = 1:3,

item2 = c ("this", "is", "text"),

item3 = c (2.5, 4.2, 5.1))

l

## $item1

## [1] 1 2 3

##

## $item2

## [1] "this" "is" "text"

##

## $item3

## [1] 2.5 4.2 5.1

data.frame(l)

## item1 item2 item3

## 1 1 this 2.5

## 2 2 is 4.2

## 3 3 text 5.1

리스트 역시 data.frame() 함수를 이용할 경우, 각 원소명을 열이름으로 한 데이터프레임이 생성됩니다. 이 경우 역시 각 원소의 데이터 길이가 동일해야 합니다.

기존 데이터프레임에 행방향 혹은 열방향으로 데이터를 추가할 수 있습니다.

df

## col1 col2 col3 col4

## 1 1 this TRUE 2.500

## 2 2 is FALSE 4.200

## 3 3 text TRUE 3.142

v4 = c ("A", "B", "C")

cbind(df, v4)

## col1 col2 col3 col4 v4

## 1 1 this TRUE 2.500 A

## 2 2 is FALSE 4.200 B

## 3 3 text TRUE 3.142 C

cbind() 함수는 ’column bind’의 약어로써, 기존 데이터프레임에 새로운 열을 추가합니다.

v5 = c (4, "R", F, 1.1)

rbind(df, v5)

## col1 col2 col3 col4

## 1 1 this TRUE 2.5

## 2 2 is FALSE 4.2

## 3 3 text TRUE 3.14159265358979

## 4 4 R FALSE 1.1

rbind() 함수는 ’row bind’의 약어로써, 기존 데이터프레임에 새로운 행을 추가합니다. 주의할 점은 각 행의 데이터 형태가 기존 데이터의 형태와 일치해야 합니다.

데이터프레임 역시 대괄호를 이용해 데이터를 추출할 수 있으며, 공백으로 두면 모든 행(열)을 선택하게 됩니다.

df

## col1 col2 col3 col4

## 1 1 this TRUE 2.500

## 2 2 is FALSE 4.200

## 3 3 text TRUE 3.142

df[2:3, ]

## col1 col2 col3 col4

## 2 2 is FALSE 4.200

## 3 3 text TRUE 3.142

데이터프레임중 2:3, 즉 두번째부터 세번째까지의 행과 모든 열을 선택하게 됩니다.

df[ , c('col2', 'col4')]

## col2 col4

## 1 this 2.500

## 2 is 4.200

## 3 text 3.142

행이름 혹은 열이름 직접 입력하여 해당값을 추출할 수도 있습니다. 위 예제에서는 열이름이 col2와 col4인 열을 추출합니다.

df[, 2]

## [1] "this" "is" "text"

df[, 2, drop = FALSE]

## col2

## 1 this

## 2 is

## 3 text

만일 하나의 열만 선택시 결과가 벡터 형태로 추출되며, drop = FALSE 인자를 추가해주면 데이터프레임의 형태가 유지되어 추출됩니다.

**0.6.4 결측치 처리하기**

결측치란 누락된 데이터를 의미하며, 데이터 분석에서 결측치를 처리하는 것은 매우 중요합니다. R에서 결측치는 NA로 표시되며, is.na() 함수를 통해 결측치 여부를 확인할 수 있습니다.

x = c(1:4, NA, 6:7, NA)

x

## [1] 1 2 3 4 NA 6 7 NA

is.na(x)

## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE

데이터가 NA인 경우에는 TRUE, 그렇지 않을 경우 FALSE를 반환합니다. 데이터프레임에도 해당 함수를 적용할 수 있습니다.

df = data.frame (col1 = c (1:3, NA),

col2 = c ("this", NA,"is", "text"),

col3 = c (TRUE, FALSE, TRUE, TRUE),

col4 = c (2.5, 4.2, 3.2, NA),

stringsAsFactors = FALSE)

df

## col1 col2 col3 col4

## 1 1 this TRUE 2.5

## 2 2 <NA> FALSE 4.2

## 3 3 is TRUE 3.2

## 4 NA text TRUE NA

is.na(df)

## col1 col2 col3 col4

## [1,] FALSE FALSE FALSE FALSE

## [2,] FALSE TRUE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

## [4,] TRUE FALSE FALSE TRUE

데이터에 결측치가 있는 경우 계산이 불가능하다는 문제가 발생합니다.

y = c(1, 3, NA, 4)

mean(y)

## [1] NA

데이터의 중간에 결측치인 NA가 존재하여 평균을 계산할 수 없으며, 이 외에도 모든 연산이 불가능하게 됩니다.

mean(y, na.rm = TRUE)

## [1] 2.667

na.rm에서 rm은 ’remove’의 약어입니다. 즉 해당 인자를 TRUE로 설정할 경우 NA를 제외하고 연산을 수행합니다. 따라서 1,3,4의 평균인 1+3+43=2.66671+3+43=2.6667이 계산됩니다.

일반적으로 결측치가 있는 경우 해당 데이터는 문제가 있다고 판단하여 제거하거나, 다른 데이터로 대체하여 채워넣기도 합니다. 먼저 결측치가 있는 데이터를 제거하는 법을 알아보겠습니다.

df = data.frame (col1 = c (1:4),

col2 = c ("this", NA,"is", "text"),

col3 = c (TRUE, FALSE, TRUE, TRUE),

col4 = c (2.5, 4.2, 3.2, 5.0)

)

df

## col1 col2 col3 col4

## 1 1 this TRUE 2.5

## 2 2 <NA> FALSE 4.2

## 3 3 is TRUE 3.2

## 4 4 text TRUE 5.0

두번째 행 col2 열에 결측치가 있으므로, 해당 부분을 제거해주도록 합니다.

na.omit(df)

## col1 col2 col3 col4

## 1 1 this TRUE 2.5

## 3 3 is TRUE 3.2

## 4 4 text TRUE 5.0

na.omit() 함수를 이용하면 NA가 위치한 부분의 데이터가 제거됩니다.

x = c(1:4, NA, 6:7, NA)

x

## [1] 1 2 3 4 NA 6 7 NA

이번에는 위와 같이 결측치가 있는 경우, 다른 데이터로 대체하도록 하겠습니다.

x[is.na(x)] = mean(x, na.rm = TRUE)

x

## [1] 1.000 2.000 3.000 4.000 3.833 6.000 7.000 3.833

mean() 함수 내부를 통해 값들의 평균을 구한 후, is.na() 함수를 통해 결측치가 위치한 부분의 데이터를 평균값인 3.833 으로 대체하였습니다. 이 외에도 결측치를 대체하는데는 다양한 방법이 존재합니다.

**0.7 데이터 불러오기 및 내보내기**

일반적으로 사람들은 csv 혹은 엑셀 파일로 저장된 데이터를 주고 받으며 데이터 분석을 하는 경우가 많습니다. 해당 형식의 파일을 R로 불러오는 법 그리고 저장하는 법에 대해 알아보겠습니다.

csv와 엑셀 샘플 파일의 주소는 다음과 같습니다.

https://github.com/hyunyulhenry/fin\_ds/blob/master/data/kospi.csv

https://github.com/hyunyulhenry/fin\_ds/blob/master/data/kospi.xlsx

아래의 코드를 실행하면 해당 파일을 PC에 다운로드 받을 수 있습니다.

download.file('https://raw.githubusercontent.com/hyunyulhenry/fin\_ds/master/data/kospi.csv', 'kospi.csv')

download.file('https://github.com/hyunyulhenry/fin\_ds/raw/master/data/kospi.xlsx', 'kospi.xlsx', mode = 'wb')

**0.7.1 워킹 디렉터리**

데이터를 불러오거나 내보낼 때 초보자들이 가장 많이 하는 실수가 워킹 디렉터리를 제대로 설정하지 않는 것입니다. 워킹 디렉터리(Working Directory)란 현재 사용중인 폴더를 의미하며, 현재 워킹 디렉터리는 콘솔창 상단 getwd() 함수를 통해 확인할 수 있습니다.

getwd()

*# [1] "C:/Users/henry/Documents/R/fin\_ds"*

파일들이 A 폴더에 있는데 워킹 디렉터리가 B 폴더인 상태에서는 파일을 불러올 수 없으므로, 해당 파일들이 현재 워킹 디렉터리에 있어야 합니다.

워킹 디렉터리를 변경할때는 setwd() 함수를 통해 위치를 직접 지정해주어도 되지만, R 스튜디오의 파일 창을 이용하면 손쉽게 변경할 수 있습니다. 먼저 파일 창 상단의 […] 부분을 클릭합니다.

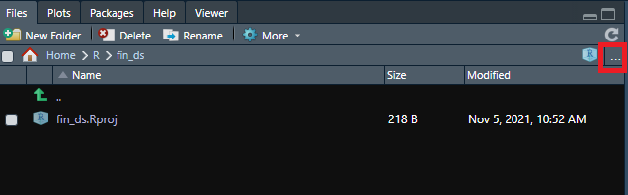


그림 0.21: 워킹 디렉터리 변경

탐색기 화면에서 원하는 폴더를 선택한 후 하단의 [Open]을 클릭합니다.

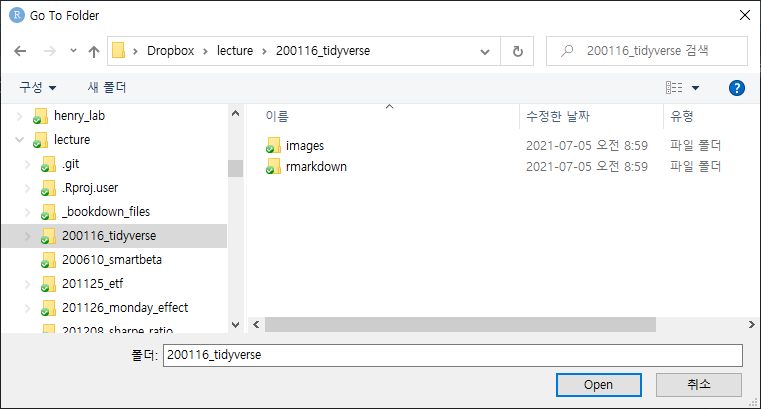


그림 0.22: 워킹 디렉터리 변경 (2)

탐색기 화면에 선택한 폴더의 파일들이 보입니다. 이제 [Move → Set As Working Directory]를 클릭하면 콘솔창에 해당 폴더를 워킹 디렉터리로 변경하는 코드가 입력되면서, 워킹 디렉터리 위치가 해당 폴더로 변경됩니다.

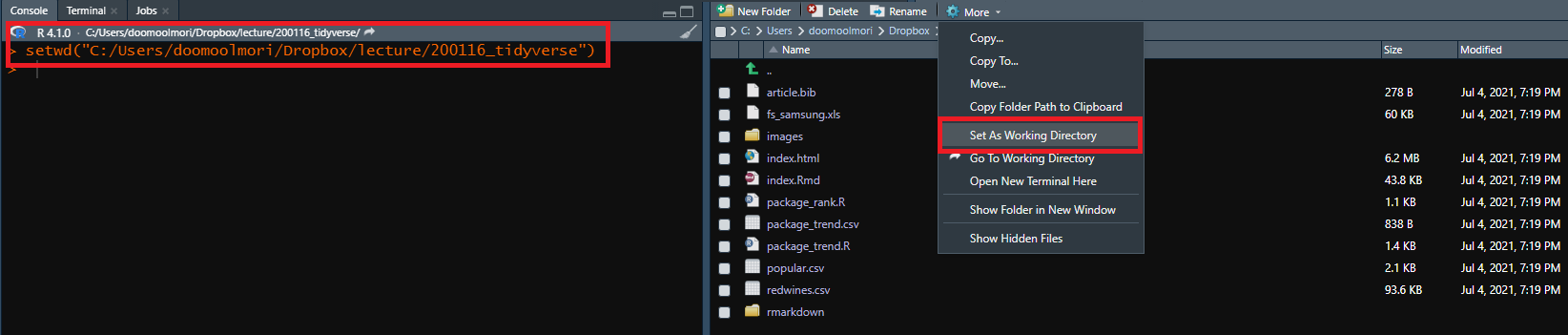


그림 0.23: 워킹 디렉터리 변경 (3)

**0.7.2 csv 파일 불러오기 및 저장하기**

먼저 R의 기본함수인 read.csv() 함수를 이용하면 매우 손쉽게 csv 파일을 불러올 수 있습니다.

kospi = read.csv('kospi.csv')

head(kospi)

## 癤풡ate Close Ret

## 1 2020-01-02 2175 -1.02

## 2 2020-01-03 2176 0.06

## 3 2020-01-06 2155 -0.98

## 4 2020-01-07 2176 0.95

## 5 2020-01-08 2151 -1.11

## 6 2020-01-09 2186 1.63

readr 패키지의 read.\_csv() 함수를 이용하면 기본 함수 대비 10배 정도 빠르게 데이터를 불러올 수 있으며, 훨씬 다양한 조건을 입력할 수도 있습니다.

library(readr)

kospi2 = read\_csv('kospi.csv')

head(kospi2)

## # A tibble: 6 x 3

## Date Close Ret

## <date> <dbl> <dbl>

## 1 2020-01-02 2175. -1.02

## 2 2020-01-03 2176. 0.06

## 3 2020-01-06 2155. -0.98

## 4 2020-01-07 2176. 0.95

## 5 2020-01-08 2151. -1.11

## 6 2020-01-09 2186. 1.63

R의 데이터를 csv로 저장하는 법은 기본함수의 write.csv() 혹은 readr 패키지의 write\_csv() 함수를 이용하면 됩니다.

*# using write.csv*

write.csv(kospi, 'kospi2.csv', row.names = FALSE)

*# using write\_csv*

write\_csv(kospi2, 'kospi2.csv')

기본함수인 write.csv() 의 경우 행이름이 자동으로 새로운 열로 추가되어 저장되므로, 이를 원하지 않을 경우 row.names = FALSE 를 추가로 입력해주어야 합니다.

**0.7.3 엑셀 파일 불러오기 및 저장하기**

R의 기본함수 중에는 엑셀 파일을 불러오는 함수가 없지만, 해당 작업을 수행하는 다양한 패키지가 존재합니다. 그 중에서 readr 패키지와 쌍둥이 격인 readxl 패키지를 이용해보겠습니다.

먼저 해당 패키지의 read\_excel() 함수를 이용해 엑셀 파일을 불러올 수 있습니다.

library(readxl)

kospi\_excel = read\_excel('kospi.xlsx', sheet = 'kospi')

head(kospi\_excel)

## # A tibble: 6 x 3

## Date Close Ret

## <dttm> <dbl> <dbl>

## 1 2020-01-02 00:00:00 2175. -1.02

## 2 2020-01-03 00:00:00 2176. 0.06

## 3 2020-01-06 00:00:00 2155. -0.98

## 4 2020-01-07 00:00:00 2176. 0.95

## 5 2020-01-08 00:00:00 2151. -1.11

## 6 2020-01-09 00:00:00 2186. 1.63

엑셀은 여러 시트로 구성된 경우가 많으며, sheet에 특정 시트명을 입력하면 해당 시트의 내용을 불러오게 됩니다. 만일 아무값도 입력하지 않을 경우 가장 첫번째 시트의 데이터를 불러옵니다.

반대로 R의 데이터를 엑셀로 저장하는 방법은 writexl 패키지의 write\_xlsx() 함수를 이용하면 됩니다.

library(writexl)

write\_xlsx(kospi\_excel, 'kospi\_excel.xlsx')

**0.8 효율성과 가독성 높이기**

**0.8.2 루프 구문**

루프 및 각종 구문을 이용하여 휴율적인 작업을 하는것도 가능합니다.

**0.8.2.1 if 구문**

먼저 if 구문은 다음과 같이 구성됩니다.

**if** (test\_expression) {

statement

}

괄호 안의 **test\_expression**이 TRUE일 경우에만 statement 코드가 실행됩니다. 간단한 예제를 살펴보겠습니다.

x = c(8, 3, -2, 5)

**if** (any(x < 0)) {

print('x contains negative number')

}

## [1] "x contains negative number"

any() 함수는 적어도 하나의 값이 참이면 참으로 출력하는 함수입니다. 즉, 위의 코드는 x중 하나라도 0보다 작은 값이 있으면 **x contains negative number**라는 문장을 출력하며, -2가 0보다 작으므로 해당 문장을 출력합니다.

y = c (8, 3, 2, 5)

**if** (any (y < 0)) {

print ("y contains negative numbers")

}

이번에는 y에 0보다 작은 값이 없으므로, 구문이 실행되지 않아 문장을 출력하지 않습니다. 이처럼 if 구문만 존재할 시 이를 만족하지 않는 경우 아무런 구문도 실행되지 않습니다. 만일 조건을 충족하지 않을 때 동작을 추가하고 싶을 경우 if else 구문을 사용하며, 이는 if의 조건을 만족하지 않을 경우 else에 해당하는 구문이 실행됩니다.

**if** (test\_expression) {

statement 1

} **else** {

statement 2

}

만일 test\_expression 구문이 TRUE이면 statement 1이 실행되며, 그렇지 않을 경우 statement 2가 실행됩니다. 실제 예제를 살펴봅시다.

y = c (8, 3, 2, 5)

**if** (any (y < 0)) {

print ("y contains negative numbers")

} **else** {

print ("y contains all positive numbers")

}

## [1] "y contains all positive numbers"

y에 음수가 존재하는 if 구문이 FALSE 이므로, else에 해당하는 메세지가 출력됩니다. ifelse 구문은 ifelse() 함수로 간단히 나타낼 수도 있습니다.

x = c (8, 3, 2, 5)

ifelse(any(x < 0), "x contains negative numbers", "x contains all positive numbers")

## [1] "x contains all positive numbers"

해당 함수는 ifelse(test, yes, no) 형태로 입력되며, test를 충족하면 yes가 그렇지 않으면 no가 실행됩니다. 위의 예에서는 x에 0보다 작은 값이 없으므로, no에 해당하는 내용이 실행됩니다.

또한 if와 else 사이에 else if 조건을 통해, 여러 조건을 추가할 수도 있습니다.

x = 7

**if** (x >= 10) {

print ("x exceeds acceptable tolerance levels")

} **else** **if**(x >= 0 & x < 10) {

print ("x is within acceptable tolerance levels")

} **else** {

print ("x is negative")

}

## [1] "x is within acceptable tolerance levels"

위 조건은 다음과 같습니다.

1. x가 10 이상일 경우 **x exceeds acceptable tolerance levels**을 출력합니다.
2. 만일 x가 10 이상, 10 미만일경우 **x is within acceptable tolerance levels**을 출력합니다.
3. 그렇지 않을 경우 **x is negative**을 출력합니다.

x는 7 이므로 else에 해당하는 내용이 출력됩니다.

**0.8.2.2 for loop 구문**

for loop 구문은 특정한 부분의 코드가 반복적으로 수행될 수 있도록 하며, 다음과 같이 구성됩니다.

**for** (i **in** 1:100) {

<code: do stuff here with i>

}

먼저 i에 1이 들어간 뒤 code에 해당하는 부분이 실행됩니다. 그 후, i에 2가 들어간 뒤 다시 code가 실행되며 이 작업이 100까지 반복됩니다. 실제 예제를 살펴보도록 하겠습니다.

**for** (i **in** 2016:2020) {

output = paste("The year is", i)

print(output)

}

## [1] "The year is 2016"

## [1] "The year is 2017"

## [1] "The year is 2018"

## [1] "The year is 2019"

## [1] "The year is 2020"

i에 2010부터 2016 까지 대입되며, **The year is**라는 문자와 결합해 결과가 출력됩니다.

**0.8.3 파이프 오퍼레이터**

파이프 오퍼레이터는 R에서 동일한 데이터를 대상으로 연속으로 작업하게 해주는 오퍼레이터(연산자)입니다.

흔히 프로그래밍에서 x라는 데이터를 F()라는 함수에 넣어 결괏값을 확인하고 싶으면 F(x)의 방법을 사용합니다. 예를 들어 3과 5라는 데이터 중 큰 값을 찾으려면 max(3,5)를 통해 확인합니다. 이를 통해 나온 결괏값을 또 다시 G()라는 함수에 넣어 결괏값을 확인하려면 비슷한 과정을 거칩니다. max(3,5)를 통해 나온 값의 제곱근을 구하려면 result = max(3,5)를 통해 첫 번째 결괏값을 저장하고, sqrt(result)를 통해 두 번째 결괏값을 계산합니다. 물론 sqrt(max(3,5))와 같은 표현법으로 한 번에 표현할 수 있습니다.

이러한 표현의 단점은 계산하는 함수가 많아질수록 저장하는 변수가 늘어나거나 괄호가 지나치게 길어진다는 것입니다. 그러나 파이프 오퍼레이터인 %>%를 사용하면 함수 간의 관계를 매우 직관적으로 표현하고 이해할 수 있습니다. 이를 정리하면 아래 표와 같습니다.

| 표 0.3: 파이프 오퍼레이터의 표현과 내용 비교 | |
| --- | --- |
| **내용** | **표현 방법** |
| F(x) | x %>% F |
| G(F(x)) | x %>% F %>% G |

간단한 예제를 통해 파이프 오퍼레이터의 사용법을 살펴보겠습니다. 먼저 다음과 같은 10개의 숫자가 있다고 가정합니다.

x = c(0.3078, 0.2577, 0.5523, 0.0564, 0.4685,

0.4838, 0.8124, 0.3703, 0.5466, 0.1703)

우리가 원하는 과정은 다음과 같습니다.

1. 각 값들의 로그값을 구할 것
2. 로그값들의 계차를 구할 것
3. 구해진 계차의 지수값을 구할 것
4. 소수 둘째 자리까지 반올림할 것

입니다. 즉 log(), diff(), exp(), round()에 대한 값을 순차적으로 구하고자 합니다.

x1 = log(x)

x2 = diff(x1)

x3 = exp(x2)

round(x3, 2)

## [1] 0.84 2.14 0.10 8.31 1.03 1.68 0.46 1.48 0.31

첫 번째 방법은 단계별 함수의 결괏값을 변수에 저장하고 저장된 변수를 다시 불러와 함수에 넣고 계산하는 방법입니다. 전반적인 계산 과정을 확인하기에는 좋지만 매번 변수에 저장하고 불러오는 과정이 매우 비효율적이며 코드도 불필요하게 길어집니다.

round(exp(diff(log(x))), 2)

## [1] 0.84 2.14 0.10 8.31 1.03 1.68 0.46 1.48 0.31

두 번째는 괄호를 통해 감싸는 방법입니다. 앞선 방법에 비해 코드는 짧아졌지만, 계산 과정을 알아보기에는 매우 불편한 방법으로 코드가 짜여 있습니다.

library(magrittr)

x %>% log() %>% diff() %>% exp() %>% round(., 2)

## [1] 0.84 2.14 0.10 8.31 1.03 1.68 0.46 1.48 0.31

마지막으로 파이프 오퍼레이터를 사용하는 방법입니다. 코드도 짧으며 계산 과정을 한눈에 파악하기도 좋습니다. 맨 왼쪽에는 원하는 변수를 입력하며, %>% 뒤에는 차례대로 계산하고자 하는 함수를 입력합니다. 변수의 입력값을 ()로 비워둘 경우, 오퍼레이터의 왼쪽에 있는 값이 입력 변수가 됩니다. 반면 round()와 같이 입력값이 두 개 이상 필요하면 마침표(.)가 오퍼레이터의 왼쪽 값으로 입력됩니다.

파이프 오퍼레이터는 크롤링뿐만 아니라 모든 코드에 사용할 수 있습니다. 이를 통해 훨씬 깔끔하면서도 데이터 처리 과정을 직관적으로 이해할 수 있습니다.

**0.9 데이터 구조 변형하기**

기본적인 R 사용법을 익혔다면 데이터의 모양을 바꾸고 가공하여 내가 원하는 결과물을 출력해야 합니다. 해당 작업은 tidyr 패키지와 dplyr 패키지를 이용해 매우 효율적으로 수행할 수 있으며, dplyr 패키지의 함수 중 일부는 SQL 구문과 매우 유사합니다.

**0.9.1 tidyr 패키지를 이용한 데이터 모양 바꾸기**

깔끔한 데이터(tidy data)는 다음과 같이 구성되어 있습니다.

* 각 변수(variable)는 열로 구성됩니다.
* 각 관측값(observation)은 행으로 구성됩니다.
* 각 타입의 관측치는 테이블을 구성합니다.



그림 0.24: tidy 데이터 요건

tidyr 패키지에는 이러한 깔끔한 데이터를 만드는데 필요한 여러 함수가 있습니다.

**0.9.1.1 pivot\_longer(): 세로로 긴 데이터 만들기**

먼저 가로로 긴(Wide) 데이터를 세로로 길게 만드는데는 pivot\_longer() 함수가 사용됩니다. 이 함수는 여러 열을 key-value 페어로 변형해줍니다.

library(tidyr)

table4a

## # A tibble: 3 x 3

## country `1999` `2000`

## \* <chr> <int> <int>

## 1 Afghanistan 745 2666

## 2 Brazil 37737 80488

## 3 China 212258 213766

위 예제에는 세 국가의 1999, 2000년 데이터가 있습니다. 이 중 country를 제외한 연도별 데이터를 세로로 길게 만들도록 하겠습니다.

long = table4a %>% pivot\_longer(names\_to = 'years', values\_to = 'cases', -country)

print(long)

## # A tibble: 6 x 3

## country years cases

## <chr> <chr> <int>

## 1 Afghanistan 1999 745

## 2 Afghanistan 2000 2666

## 3 Brazil 1999 37737

## 4 Brazil 2000 80488

## 5 China 1999 212258

## 6 China 2000 213766

열 이름에 해당하던 1999, 2000 데이터가 names\_to에 입력한 years 열에 입력되었습니다. 또한 각 관측값이 values\_to에 입력한 cases 열에 입력되었습니다. country 앞에는 마이너스(-) 기호를 붙여 해당 열은 그대로 유지됩니다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **country** | **1999** | **2000** | | --- | --- | --- | | Afghanistan | 745 | 2666 | | Brazil | 37737 | 80488 | | China | 212258 | 213766 | | | **country** | **years** | **cases** | | --- | --- | --- | | Afghanistan | 1999 | 745 | | Afghanistan | 2000 | 2666 | | Brazil | 1999 | 37737 | | Brazil | 2000 | 80488 | | China | 1999 | 212258 | | China | 2000 | 213766 | |

**0.9.1.2 pivot\_wider(): 가로로 긴 데이터 만들기**

pivot\_longer() 함수와 반대로, pivot\_wider() 함수를 이용할 경우 세로로 긴 데이터를 가로로 길게 만들 수 있습니다. 위의 데이터에 year 열에 있는 항목들을 열 이름으로, cases 열에 있는 항목들을 가로로 길게 되돌려 보겠습니다.

back2wide = long %>% pivot\_wider(names\_from = 'years', values\_from = 'cases')

print(back2wide)

## # A tibble: 3 x 3

## country `1999` `2000`

## <chr> <int> <int>

## 1 Afghanistan 745 2666

## 2 Brazil 37737 80488

## 3 China 212258 213766

names\_from와 values\_from에 각각 열이름 및 관측값에 해당하는 값을 입력하면, 원래대로 가로로 긴 테이블 형태가 되었습니다.

**0.9.1.3 separate(): 하나의 열을 여러 열로 나누기**

table3

## # A tibble: 6 x 3

## country year rate

## \* <chr> <int> <chr>

## 1 Afghanistan 1999 745/19987071

## 2 Afghanistan 2000 2666/20595360

## 3 Brazil 1999 37737/172006362

## 4 Brazil 2000 80488/174504898

## 5 China 1999 212258/1272915272

## 6 China 2000 213766/1280428583

rate 열에는 데이터가 ###/#### 형태로 들어가 있습니다. / 기호를 기준으로 앞과 뒤로 각각 나누어보도록 하겠습니다.

table3 %>%

separate(rate, into = c("cases", "population"))

## # A tibble: 6 x 4

## country year cases population

## <chr> <int> <chr> <chr>

## 1 Afghanistan 1999 745 19987071

## 2 Afghanistan 2000 2666 20595360

## 3 Brazil 1999 37737 172006362

## 4 Brazil 2000 80488 174504898

## 5 China 1999 212258 1272915272

## 6 China 2000 213766 1280428583

separate() 함수를 사용할 경우 rate 열이 “/”를 기준으로 into에 입력한 cases와 population 열로 분리됩니다.

table3 %>%

separate(rate, into = c("cases", "population"), remove = FALSE)

## # A tibble: 6 x 5

## country year rate cases population

## <chr> <int> <chr> <chr> <chr>

## 1 Afghanistan 1999 745/19987071 745 19987071

## 2 Afghanistan 2000 2666/20595360 2666 20595360

## 3 Brazil 1999 37737/172006362 37737 172006362

## 4 Brazil 2000 80488/174504898 80488 174504898

## 5 China 1999 212258/1272915272 212258 1272915272

## 6 China 2000 213766/1280428583 213766 1280428583

remove = FALSE 인자를 추가해주면 원래의 열이 사라지지 않고 유지됩니다.

**0.9.1.4 unite(): 여러 열을 하나로 합치기**

separate() 함수와는 반대로, unite() 함수를 이용시 여러 열을 하나로 합칠 수 있습니다.

table5

## # A tibble: 6 x 4

## country century year rate

## \* <chr> <chr> <chr> <chr>

## 1 Afghanistan 19 99 745/19987071

## 2 Afghanistan 20 00 2666/20595360

## 3 Brazil 19 99 37737/172006362

## 4 Brazil 20 00 80488/174504898

## 5 China 19 99 212258/1272915272

## 6 China 20 00 213766/1280428583

이번에는 century와 year 열을 합친 후 새로운 열을 만들어보도록 하겠습니다.

table5 %>%

unite(new, century, year, sep = "\_")

## # A tibble: 6 x 3

## country new rate

## <chr> <chr> <chr>

## 1 Afghanistan 19\_99 745/19987071

## 2 Afghanistan 20\_00 2666/20595360

## 3 Brazil 19\_99 37737/172006362

## 4 Brazil 20\_00 80488/174504898

## 5 China 19\_99 212258/1272915272

## 6 China 20\_00 213766/1280428583

century 열과 year열이 합쳐진 후 new 열에 입력되었으며, sep 인자를 통해 구분자는 “\_“로 설정하였습니다.

**0.9.1.5 tidyr 패키지의 기타 함수**

먼저 fill() 함수는 결측치를 채워주는 역할을 합니다.

score = tribble(

~ person, ~ Math, ~ Computer,

"Henry", 1, 7,

NA, 2, 10,

NA, NA, 9,

"David", 1, 4

)

score

## # A tibble: 4 x 3

## person Math Computer

## <chr> <dbl> <dbl>

## 1 Henry 1 7

## 2 <NA> 2 10

## 3 <NA> NA 9

## 4 David 1 4

score의 2번째와 3번째 행에 NA 데이터가 있어 이를 채워줄 필요가 있습니다.

score %>%

fill(person, Math)

## # A tibble: 4 x 3

## person Math Computer

## <chr> <dbl> <dbl>

## 1 Henry 1 7

## 2 Henry 2 10

## 3 Henry 2 9

## 4 David 1 4

fill() 함수는 결측치가 있을 경우, 각 열의 이전 데이터를 이용해 채워줍니다. 반면에 NA 데이터를 특정 값으로 변경할 수도 있습니다.

score %>% replace\_na(replace = list(person = "unknown", Math = 0))

## # A tibble: 4 x 3

## person Math Computer

## <chr> <dbl> <dbl>

## 1 Henry 1 7

## 2 unknown 2 10

## 3 unknown 0 9

## 4 David 1 4

replace\_na() 함수를 이용해 person 열의 NA 데이터를 **unknown**으로, Math열의 NA 데이터를 0으로 변경하였습니다.

**0.9.2 dplyr 패키지를 이용한 데이터 변형하기**

데이터를 필터링 하거나, 요약하거나, 정렬하거나, 새로운 변수를 만드는 등 데이터 분석을 위해서는 데이터 변형하고 가공해야 하는 경우가 많습니다. R의 기본 함수도 이러한 기능을 제공하지만, dplyr 패키지를 이용할 경우 훨씬 빠르고 효율적으로 업무를 처리할 수 있습니다.

nycflights13 패키지의 flights 데이터셋을 예제로 사용하도록 하겠습니다.

library(dplyr)

library(nycflights13)

flights

## # A tibble: 336,776 x 19

## year month day dep\_time sched\_dep\_time dep\_delay arr\_time sched\_arr\_time

## <int> <int> <int> <int> <int> <dbl> <int> <int>

## 1 2013 1 1 517 515 2 830 819

## 2 2013 1 1 533 529 4 850 830

## 3 2013 1 1 542 540 2 923 850

## 4 2013 1 1 544 545 -1 1004 1022

## 5 2013 1 1 554 600 -6 812 837

## 6 2013 1 1 554 558 -4 740 728

## 7 2013 1 1 555 600 -5 913 854

## 8 2013 1 1 557 600 -3 709 723

## 9 2013 1 1 557 600 -3 838 846

## 10 2013 1 1 558 600 -2 753 745

## # ... with 336,766 more rows, and 11 more variables: arr\_delay <dbl>,

## # carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,

## # air\_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time\_hour <dttm>

**0.9.2.1 select(): 원하는 열 선택하기**

select() 함수를 이용해 특정 열만을 선택할 수 있습니다.

flights %>% select(year, month, day) %>% head()

## # A tibble: 6 x 3

## year month day

## <int> <int> <int>

## 1 2013 1 1

## 2 2013 1 1

## 3 2013 1 1

## 4 2013 1 1

## 5 2013 1 1

## 6 2013 1 1

select() 함수 내에 선택하고자 하는 열을 입력하여 year, month, day 열을 선택했습니다.

flights %>% select(year:day) %>% head()

## # A tibble: 6 x 3

## year month day

## <int> <int> <int>

## 1 2013 1 1

## 2 2013 1 1

## 3 2013 1 1

## 4 2013 1 1

## 5 2013 1 1

## 6 2013 1 1

콜론(:)을 이용해 year부터 day 까지의 열을 한번에 선택할 수도 있습니다.

flights %>% select(-(year:day)) %>% head()

## # A tibble: 6 x 16

## dep\_time sched\_dep\_time dep\_delay arr\_time sched\_arr\_time arr\_delay carrier

## <int> <int> <dbl> <int> <int> <dbl> <chr>

## 1 517 515 2 830 819 11 UA

## 2 533 529 4 850 830 20 UA

## 3 542 540 2 923 850 33 AA

## 4 544 545 -1 1004 1022 -18 B6

## 5 554 600 -6 812 837 -25 DL

## 6 554 558 -4 740 728 12 UA

## # ... with 9 more variables: flight <int>, tailnum <chr>, origin <chr>,

## # dest <chr>, air\_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,

## # time\_hour <dttm>

마이너스(-)를 이용할 경우 해당 열을 제외한 모든 열이 선택됩니다.

flights %>% select(starts\_with("dep")) %>% head()

## # A tibble: 6 x 2

## dep\_time dep\_delay

## <int> <dbl>

## 1 517 2

## 2 533 4

## 3 542 2

## 4 544 -1

## 5 554 -6

## 6 554 -4

select() 함수 내에 starts\_with() 함수를 이용할 경우, 해당 문자로 시작하는 열을 모두 선택할 수 있습니다. 이 외에도 ends\_with() 함수는 해당 문자로 끝나는 열울, contains() 함수는 해당 문자가 포함되는 열을 선택합니다.

**0.9.2.2 rename(): 열이름 바꾸기**

flights %>% rename(tail\_num = tailnum) %>% select(tail\_num) %>% head()

## # A tibble: 6 x 1

## tail\_num

## <chr>

## 1 N14228

## 2 N24211

## 3 N619AA

## 4 N804JB

## 5 N668DN

## 6 N39463

rename() 함수를 이용해 **tailnum** 이던 열 이름을 **tail\_num** 으로 변경하였습니다. 해당 함수는 rename(변경하고자 하는 이름 = 변경전 이름) 형태로 입력해야 합니다.

**0.9.2.3 filter(): 필터링**

특정 열에 원하는 데이터가 있는 부분만 필터링을 해야하는 경우가 많으며, filter() 함수를 사용해 손쉽게 해결할 수 있습니다.

flights %>% filter(month == 3, day == 1) %>% head()

## # A tibble: 6 x 19

## year month day dep\_time sched\_dep\_time dep\_delay arr\_time sched\_arr\_time

## <int> <int> <int> <int> <int> <dbl> <int> <int>

## 1 2013 3 1 4 2159 125 318 56

## 2 2013 3 1 50 2358 52 526 438

## 3 2013 3 1 117 2245 152 223 2354

## 4 2013 3 1 454 500 -6 633 648

## 5 2013 3 1 505 515 -10 746 810

## 6 2013 3 1 521 530 -9 813 827

## # ... with 11 more variables: arr\_delay <dbl>, carrier <chr>, flight <int>,

## # tailnum <chr>, origin <chr>, dest <chr>, air\_time <dbl>, distance <dbl>,

## # hour <dbl>, minute <dbl>, time\_hour <dttm>

filter() 함수 내에 필터링 하고자 하는 조건, 즉 month가 3이고 day가 1인 조건을 입력하면 해당 조건에 부합하는 행만 선택됩니다.

**0.9.2.4 summarize(): 요약값 계산하기**

요약 통계값은 summarize() 함수를 통해 쉽게 계산할 수 있습니다.

flights %>% summarize(max\_dep = max(dep\_time, na.rm = TRUE),

min\_dep = min(dep\_time, na.rm = TRUE))

## # A tibble: 1 x 2

## max\_dep min\_dep

## <int> <int>

## 1 2400 1

summarize() 함수를 통해 max\_dep에는 dep\_time의 최대값을, min\_dep에는 dep\_time의 최소값을 구해줍니다. **na.rm** 인자를 TRUE로 설정하여 NA 데이터는 제거해 줍니다.

**0.9.2.5 group\_by(): 원하는 조건으로 그룹화**

각 그룹별 통계량을 계산할 때는 group\_by() 함수를 통해 그룹을 묶고, 계산하는 것이 편합니다.

by\_day = flights %>% group\_by(year, month, day)

by\_day %>% head()

## # A tibble: 6 x 19

## # Groups: year, month, day [1]

## year month day dep\_time sched\_dep\_time dep\_delay arr\_time sched\_arr\_time

## <int> <int> <int> <int> <int> <dbl> <int> <int>

## 1 2013 1 1 517 515 2 830 819

## 2 2013 1 1 533 529 4 850 830

## 3 2013 1 1 542 540 2 923 850

## 4 2013 1 1 544 545 -1 1004 1022

## 5 2013 1 1 554 600 -6 812 837

## 6 2013 1 1 554 558 -4 740 728

## # ... with 11 more variables: arr\_delay <dbl>, carrier <chr>, flight <int>,

## # tailnum <chr>, origin <chr>, dest <chr>, air\_time <dbl>, distance <dbl>,

## # hour <dbl>, minute <dbl>, time\_hour <dttm>

year, month, day를 기준으로 그룹을 묶었습니다. 아직 계산을 하지 않았으므로 출력되는 데이터프레임 자체는 원래와 동일하며, **Groups**를 통해 어떠한 조건으로 그룹이 묶여있는지 확인됩니다.

by\_day %>%

summarise(delay = mean(dep\_delay, na.rm = TRUE)) %>%

head()

## # A tibble: 6 x 4

## # Groups: year, month [1]

## year month day delay

## <int> <int> <int> <dbl>

## 1 2013 1 1 11.5

## 2 2013 1 2 13.9

## 3 2013 1 3 11.0

## 4 2013 1 4 8.95

## 5 2013 1 5 5.73

## 6 2013 1 6 7.15

해당 데이터는 그룹별로 묶여 있으므로, summarise() 함수를 적용하면 각 그룹(year, month, day) 별로 dep\_delay의 평균을 구합니다.

flights %>% group\_by(dest) %>%

summarize(

count = n(),

dist = mean(distance, na.rm = TRUE),

delay = mean(arr\_delay, na.rm = TRUE)

)

## # A tibble: 105 x 4

## dest count dist delay

## <chr> <int> <dbl> <dbl>

## 1 ABQ 254 1826 4.38

## 2 ACK 265 199 4.85

## 3 ALB 439 143 14.4

## 4 ANC 8 3370 -2.5

## 5 ATL 17215 757. 11.3

## 6 AUS 2439 1514. 6.02

## 7 AVL 275 584. 8.00

## 8 BDL 443 116 7.05

## 9 BGR 375 378 8.03

## 10 BHM 297 866. 16.9

## # ... with 95 more rows

한 번에 여러 통계량을 계산할 수도 있으며, n()은 데이터의 갯수를 의미합니다.

**0.9.2.6 arrange(): 데이터 정렬하기**

arrange() 함수를 통해 원하는 열을 기준으로 데이터를 순서대로 정렬할 수 있으며, 오름차순을 기본으로 합니다.

flights %>% arrange(year, month, day) %>% head()

## # A tibble: 6 x 19

## year month day dep\_time sched\_dep\_time dep\_delay arr\_time sched\_arr\_time

## <int> <int> <int> <int> <int> <dbl> <int> <int>

## 1 2013 1 1 517 515 2 830 819

## 2 2013 1 1 533 529 4 850 830

## 3 2013 1 1 542 540 2 923 850

## 4 2013 1 1 544 545 -1 1004 1022

## 5 2013 1 1 554 600 -6 812 837

## 6 2013 1 1 554 558 -4 740 728

## # ... with 11 more variables: arr\_delay <dbl>, carrier <chr>, flight <int>,

## # tailnum <chr>, origin <chr>, dest <chr>, air\_time <dbl>, distance <dbl>,

## # hour <dbl>, minute <dbl>, time\_hour <dttm>

arrange() 함수 내에 입력한 [year -> month -> day] 순으로 오름차순 정렬이 됩니다.

flights %>% arrange(desc(dep\_delay)) %>% head()

## # A tibble: 6 x 19

## year month day dep\_time sched\_dep\_time dep\_delay arr\_time sched\_arr\_time

## <int> <int> <int> <int> <int> <dbl> <int> <int>

## 1 2013 1 9 641 900 1301 1242 1530

## 2 2013 6 15 1432 1935 1137 1607 2120

## 3 2013 1 10 1121 1635 1126 1239 1810

## 4 2013 9 20 1139 1845 1014 1457 2210

## 5 2013 7 22 845 1600 1005 1044 1815

## 6 2013 4 10 1100 1900 960 1342 2211

## # ... with 11 more variables: arr\_delay <dbl>, carrier <chr>, flight <int>,

## # tailnum <chr>, origin <chr>, dest <chr>, air\_time <dbl>, distance <dbl>,

## # hour <dbl>, minute <dbl>, time\_hour <dttm>

정렬하고자 하는 열에 desc() 함수를 추가할 경우, 오름차순이 아닌 내림차순으로 정렬됩니다.

**0.9.2.7 \*\_join(): 데이터 합치기**

여러 테이블을 하나로 합치기 위해 \*\_join() 함수를 이용합니다. 합치는 방법은 그림 [0.25](https://hyunyulhenry.github.io/quant_cookbook/r-%EA%B8%B0%EC%B4%88-%EB%B0%B0%EC%9A%B0%EA%B8%B0.html#fig:joinimg2)과 표 [0.4](https://hyunyulhenry.github.io/quant_cookbook/r-%EA%B8%B0%EC%B4%88-%EB%B0%B0%EC%9A%B0%EA%B8%B0.html#tab:joindesc2) 같이 크게 네가지 종류가 있습니다.

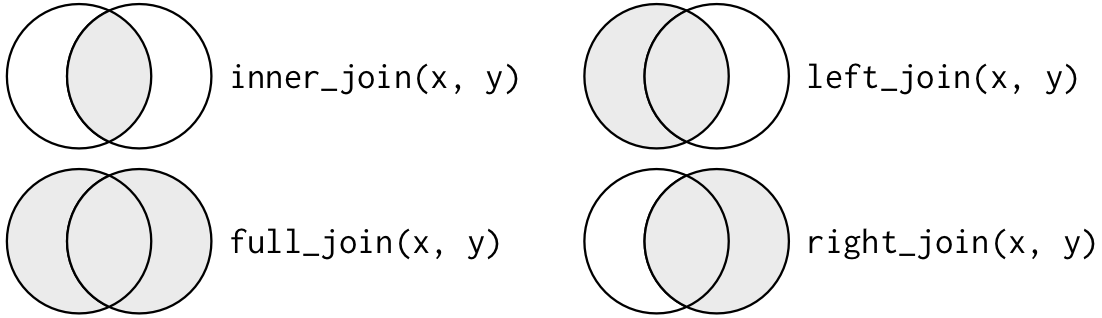


그림 0.25: \*\_ join() 함수의 종류

| 표 0.4: join 함수의 종류 | |
| --- | --- |
| **함수** | **내용** |
| inner\_join() | 교집합 |
| full\_join() | 합집합 |
| left\_join() | 좌측 기준 |
| right\_join() | 우측 기준 |

다음 두개의 데이터 테이블을 이용하도록 합니다.

flights2 = flights %>%

select(year:day, hour, tailnum, carrier)

flights2 %>% head()

## # A tibble: 6 x 6

## year month day hour tailnum carrier

## <int> <int> <int> <dbl> <chr> <chr>

## 1 2013 1 1 5 N14228 UA

## 2 2013 1 1 5 N24211 UA

## 3 2013 1 1 5 N619AA AA

## 4 2013 1 1 5 N804JB B6

## 5 2013 1 1 6 N668DN DL

## 6 2013 1 1 5 N39463 UA

airlines %>% head()

## # A tibble: 6 x 2

## carrier name

## <chr> <chr>

## 1 9E Endeavor Air Inc.

## 2 AA American Airlines Inc.

## 3 AS Alaska Airlines Inc.

## 4 B6 JetBlue Airways

## 5 DL Delta Air Lines Inc.

## 6 EV ExpressJet Airlines Inc.

flights2 데이터에는 항공사 명의 약자인 carrier가 있으며, airlines 데이터는 해당 약자의 풀 네임이 적혀있습니다. left\_join() 함수를 이용해 왼쪽 데이터프레임을 기준으로 데이터를 합치도록 하며, 두 데이터 모두 carrier 열이 있으므로 이를 기준으로 데이터가 합치도록 하겠습니다.

flights2 %>%

left\_join(airlines, by = "carrier") %>%

head()

## # A tibble: 6 x 7

## year month day hour tailnum carrier name

## <int> <int> <int> <dbl> <chr> <chr> <chr>

## 1 2013 1 1 5 N14228 UA United Air Lines Inc.

## 2 2013 1 1 5 N24211 UA United Air Lines Inc.

## 3 2013 1 1 5 N619AA AA American Airlines Inc.

## 4 2013 1 1 5 N804JB B6 JetBlue Airways

## 5 2013 1 1 6 N668DN DL Delta Air Lines Inc.

## 6 2013 1 1 5 N39463 UA United Air Lines Inc.

flights2에서 모든 데이터를 가져오며, airlines의 name 열이 기존 테이블에 추가됩니다. join 구문에 대한 더욱 상세한 예제 및 애니메이션은 다음 주소를 참조하시기 바랍니다.

https://github.com/gadenbuie/tidyexplain

**0.9.2.8 mutate(): 새로운 열 생성하기**

mutate() 함수를 사용해 기존 열끼리 계산을 하여 새로운 열을 생성할 수 있습니다.

flights\_sml = flights %>%

select(

year:day,

ends\_with("delay"),

distance,

air\_time

)

flights\_sml %>%

mutate(

gain = dep\_delay - arr\_delay,

speed = distance / air\_time \* 60

) %>%

head()

## # A tibble: 6 x 9

## year month day dep\_delay arr\_delay distance air\_time gain speed

## <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

## 1 2013 1 1 2 11 1400 227 -9 370.

## 2 2013 1 1 4 20 1416 227 -16 374.

## 3 2013 1 1 2 33 1089 160 -31 408.

## 4 2013 1 1 -1 -18 1576 183 17 517.

## 5 2013 1 1 -6 -25 762 116 19 394.

## 6 2013 1 1 -4 12 719 150 -16 288.

먼저 flights에서 일부 열을 선택하여 flights\_sml에 저장합니다.그 후, mutate() 함수를 이용해 새로운 열을 만들어 줍니다. gain 열에는 dep\_delay와 arr\_delay의 차이가, speed 열에는 distance와 air\_time 비에 60을 곱한 값이 새롭게 생성됩니다.

flights\_sml %>%

mutate(

across(c('dep\_delay', 'arr\_delay'), ~ .x \* 60)

) %>%

head()

## # A tibble: 6 x 7

## year month day dep\_delay arr\_delay distance air\_time

## <int> <int> <int> <dbl> <dbl> <dbl> <dbl>

## 1 2013 1 1 120 660 1400 227

## 2 2013 1 1 240 1200 1416 227

## 3 2013 1 1 120 1980 1089 160

## 4 2013 1 1 -60 -1080 1576 183

## 5 2013 1 1 -360 -1500 762 116

## 6 2013 1 1 -240 720 719 150

동일한 함수를 한번에 여러행에 적용해야 할 때는 mutate() 함수 내에 across() 함수를 입력합니다. 위 예제에서는 dep\_delay과 arr\_delay열의 데이터에 60을 곱해주었습니다. across() 함수의 자세한 사용법은 다소 생소하고 어려울 수 있으므로 아래 페이지의 내용을 따라가며 익히시는게 좋습니다.

https://dplyr.tidyverse.org/reference/across.html

[데이터 시각화]

# 1 그래픽 문법

R에서 기본적으로 제공하는 plot() 함수를 이용해도 시각화가 충분히 가능합니다. 다음 사이트에는 기본 함수를 이용해 표현할 수 있는 다양한 그림이 나와 있습니다.

* <http://www.sthda.com/english/wiki/r-base-graphs>

그러나 기본 함수를 이용하여 시각화를 할 경우 다음과 같은 문제가 있습니다.

* 서로 다른 형태의 그림을 나타내기 위해 각각 다른 함수를 이용해야 함
* 표현할 수 있는 그림에 한계가 있음
* 원하는 형태로 꾸미기가 복잡함

ggplot2 패키지는 데이터 과학자들에게 가장 많이 사랑받는 패키지 중 하나이며, ggplot() 함수를 사용하면 그림을 훨씬 아름답게 표현할 수 있으며 다양한 기능들을 매우 쉽게 사용할 수도 있습니다. 처음에는 함수의 문법이 다소 어색하지만, 해당 패키지의 근본이 되는 철학인 그래픽 문법(The Grammar of Graphics)를 이해하고 조금만 연습해본다면, 충분히 손쉽게 사용이 가능합니다.

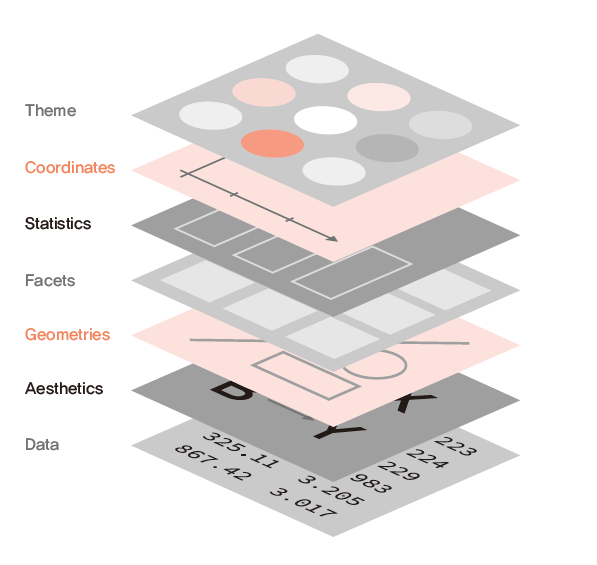
그래픽 문법(Grammar of Graphics)은 릴랜드 윌킨스(Leland Wilkinson)의 책 The Grammar of Graphics(Wilkinson 2012)에서 따온 것으로써, 데이터를 어떻게 표현할 것인지에 대한 내용입니다.

*문법은 언어의 표현을 풍부하게 만든다. 단어만 있고 문법이 없는 언어가 있다면(단어 = 문장), 오직 단어의 갯수만큼만 생각을 표현할 수 있다. 문장 내에서 단어가 어떻게 구성되는 지를 규정함으로써, 문법은 언어의 범위를 확장한다.*

*— Leland Wilkinson, 《The Grammar of Graphics》*

그래픽 문법에서 말하는 요소는 다음과 같습니다.

1. Data: 시각화에 사용될 데이터
2. Aesthetics: 데이터를 나타내는 시각적인 요소(x축, y축, 사이즈, 색깔, 모양 등)
3. Geometrics: 데이터를 나타내는 도형
4. Facets: 하위 집합으로 분할하여 시각화
5. Statistics: 통계값을 표현
6. Coordinates: 데이터를 표현 할 이차원 좌표계
7. Theme: 그래프를 꾸밈



*Figure 1.1: The Grammar of Graphics*

ggplot2 패키지의 앞글자가 gg인 것에서 알 수 있듯이, 해당 패키지는 그래픽 문법을 토대로 시각화를 표현하며, 전반적인 시각화의 순서는 그래픽 문법의 순서와 같습니다. ggplot2 패키지의 특징은 각 요소를 연결할 때 플러스(+) 기호를 사용한다는 점이며, 이는 그래픽 문법의 순서에 따라 요소들을 쌓아나간 후 최종적인 그래픽을 완성하는 패키지의 특성 때문입니다.

아래 사이트에는 R에서 ggplot2와 기타 패키지를 이용해 표현할 수 있는 그림이 정리되어 있습니다.

* <https://www.r-graph-gallery.com/>

# 2 데이터셋: 다이아몬드

ggplot2 패키지에는 데이터분석 및 시각화 연습을 위한 각종 데이터셋이 있으며, 그 중에서도 diamonds 데이터셋이 널리 사용됩니다. 먼저 해당 데이터를 불러오도록 하겠습니다.

library(ggplot2)

data(diamonds)

head(diamonds)

## # A tibble: 6 x 10

## carat cut color clarity depth table price x y z

## <dbl> <ord> <ord> <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>

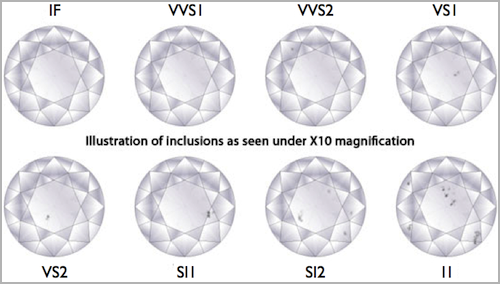
## 1 0.23 Ideal E SI2 61.5 55 326 3.95 3.98 2.43

데이터의 각 변수는 다음과 같습니다.

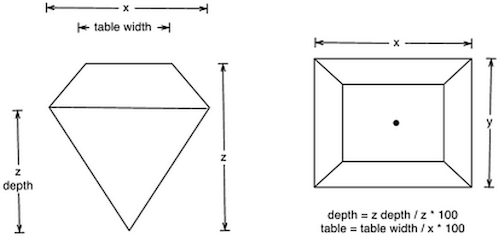
* carat: 다이아몬드 무게
* cut: 컷팅의 가치
* color: 다이아몬드 색상
* clarity: 깨끗한 정도
* depth: 깊이 비율, z / mean(x, y)
* table: 가장 넓은 부분의 너비 대비 다이아몬드 꼭대기의 너비
* price: 가격
* x: 길이
* y: 너비
* z: 깊이



*Figure 2.1: color*



*Figure 2.2: clarity*



*Figure 2.3: table*

이 외에도 R에서 제공하는 다양한 데이터셋은 다음의 함수를 통해 확인할 수 있습니다.

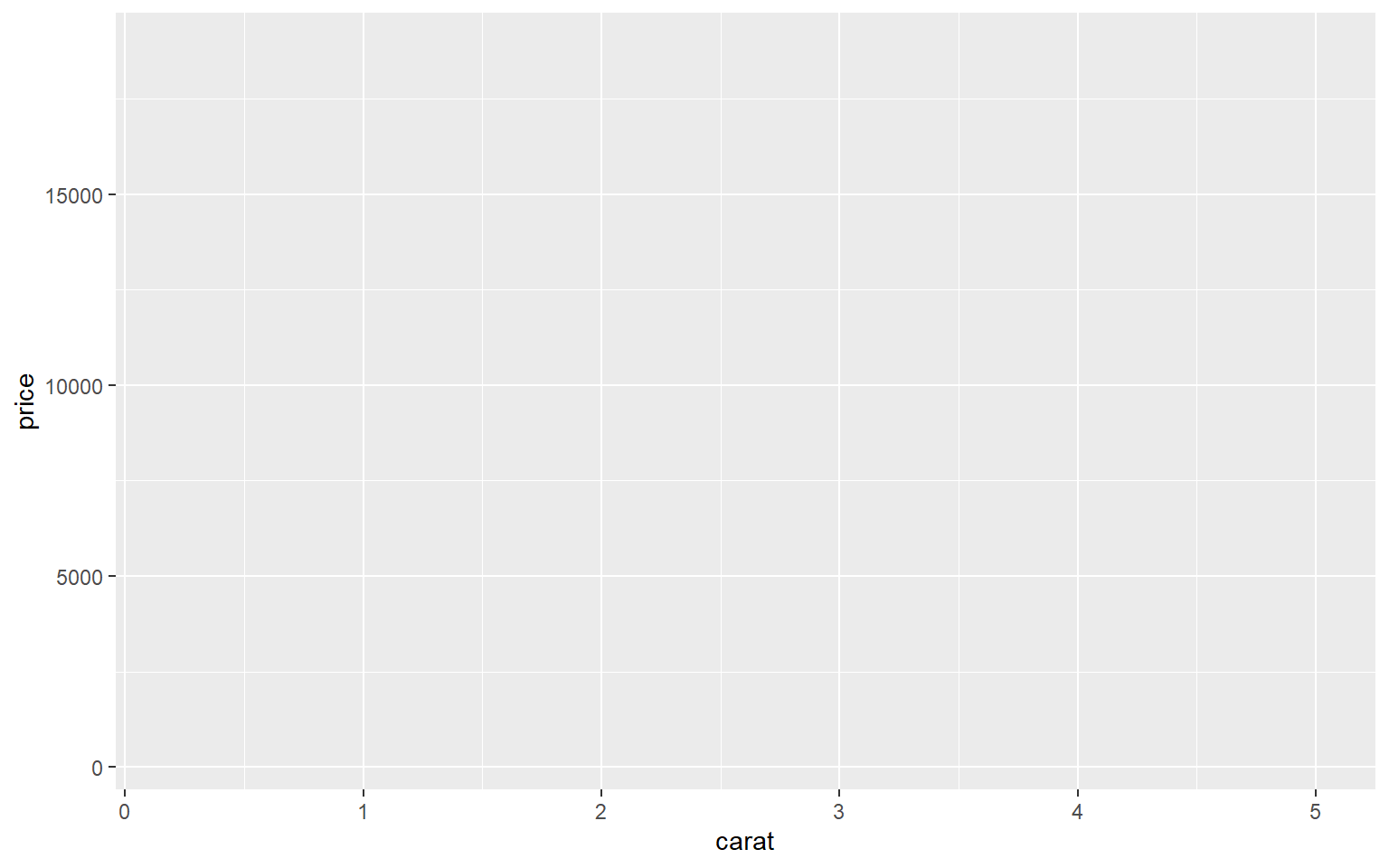
data()

## [1] "diamonds" "economics" "economics\_long"

# 3 Data, Aesthetics, Geometrics

그래픽 문법의 순서에 맞춰 그림을 쌓아나가보도록 하겠습니다. 먼저 Data는 사용될 데이터이며, Aesthetics는 xx축, yy축, 사이즈 등 시각적인 요소를 의미합니다.

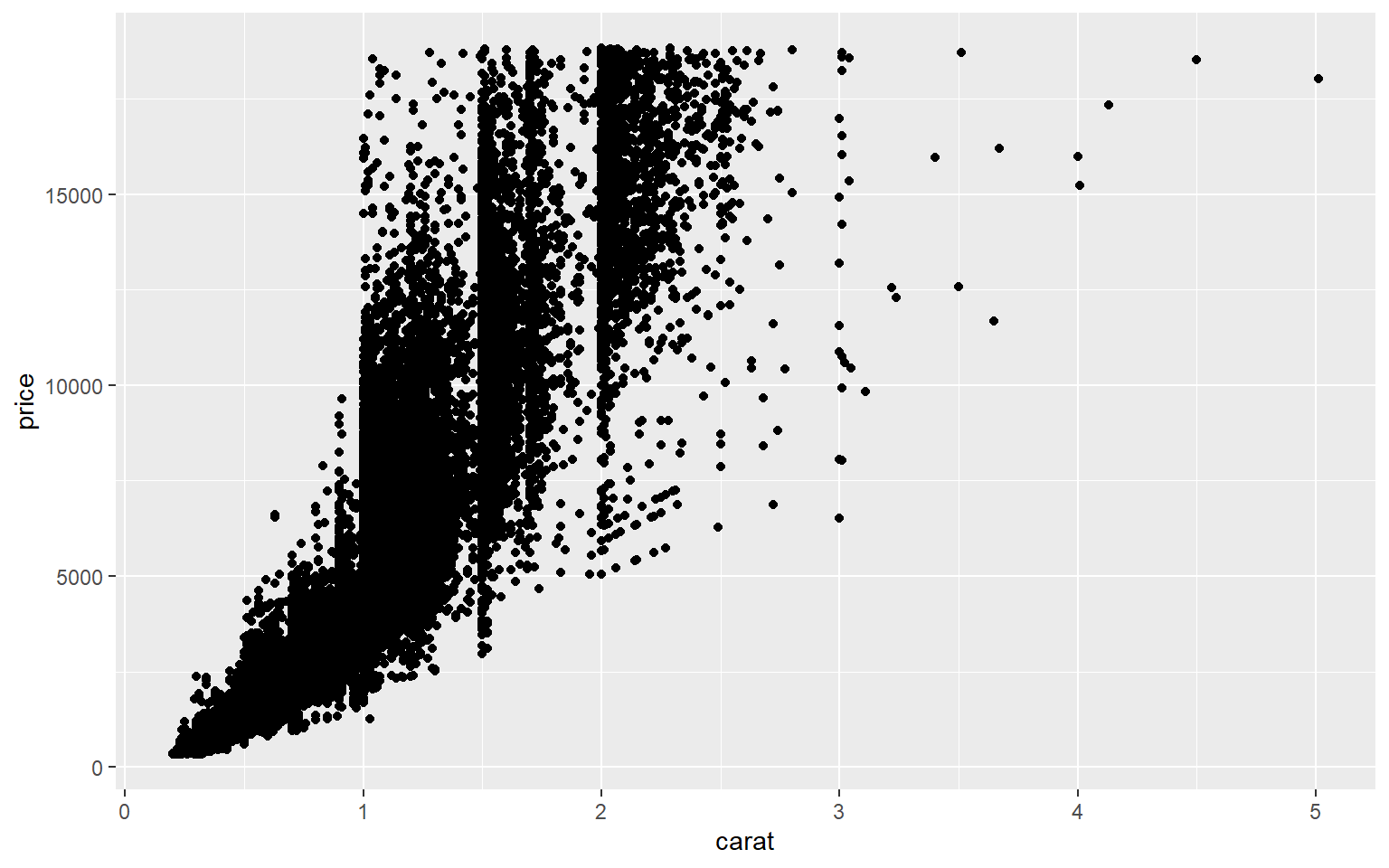
ggplot(data = diamonds, aes(x = carat, y = price))



x축과 y축에 우리가 매핑한 carat과 price가 표현되었지만, 어떠한 모양(Geometrics)으로 시각화를 할지 정의하지 않았으므로 빈 그림이 생성됩니다. 다음으로 Geometrics을 통해 데이터를 그림으로 표현해주도록 하겠습니다.

ggplot(data = diamonds, aes(x = carat, y = price)) +

geom\_point()



사전에 정의된 Data와 Aesthetics 위에, 플러스(+) 기호를 통해 geom\_point() 함수를 입력하여 산점도가 표현되었습니다. geom은 Geometrics의 약자이며, 이처럼 geom\_\*() 함수를 통해 원하는 형태로 시각화를 할 수 있습니다.

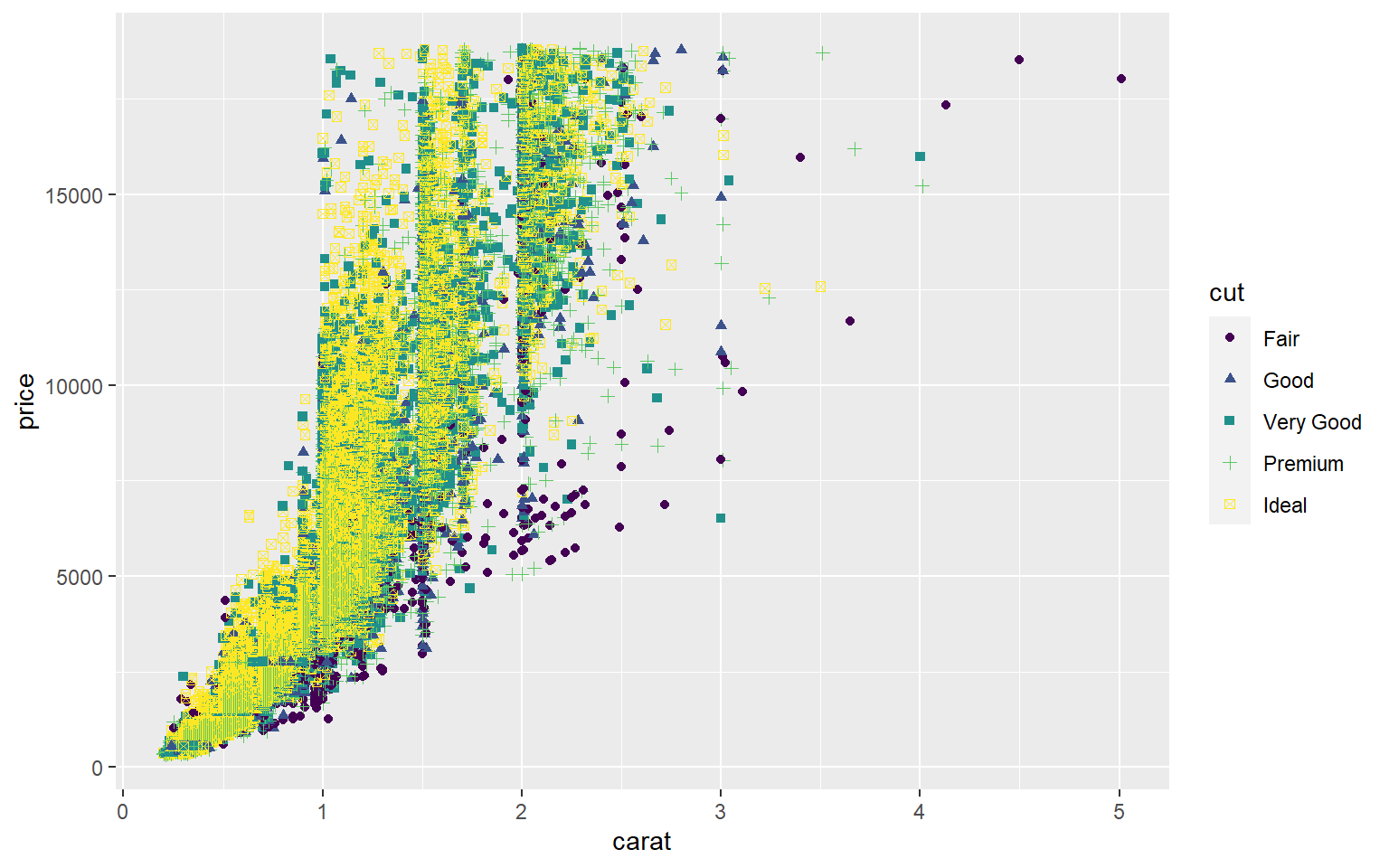
일반적으로 Data는 ggplot() 함수 내에서 정의하기 보다는, dplyr 패키지의 함수들을 이용하여 데이터를 가공한 후 파이프 오퍼레이터(%>%)를 통해 연결합니다.

library(magrittr)

diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_point(aes(color = cut, shape = cut))



diamonds 데이터를 파이프 오퍼레이터로 이을 경우 그대로 시각화가 가능하며, ggplot() 함수 내에 데이터를 입력하지 않아도 됩니다.

geom\_point() 내부에서 aes()를 통해 점의 색깔을 매핑해줄 수 있습니다. color = cut, shape = cut을 지정하여 cut에 따라 점의 색깔과 형태를 다르게 표현하였습니다. 이 외에도 size 등을 통해 그룹별로 그래프를 각각 다르게 표현할 수 있습니다.

# 4 Facets

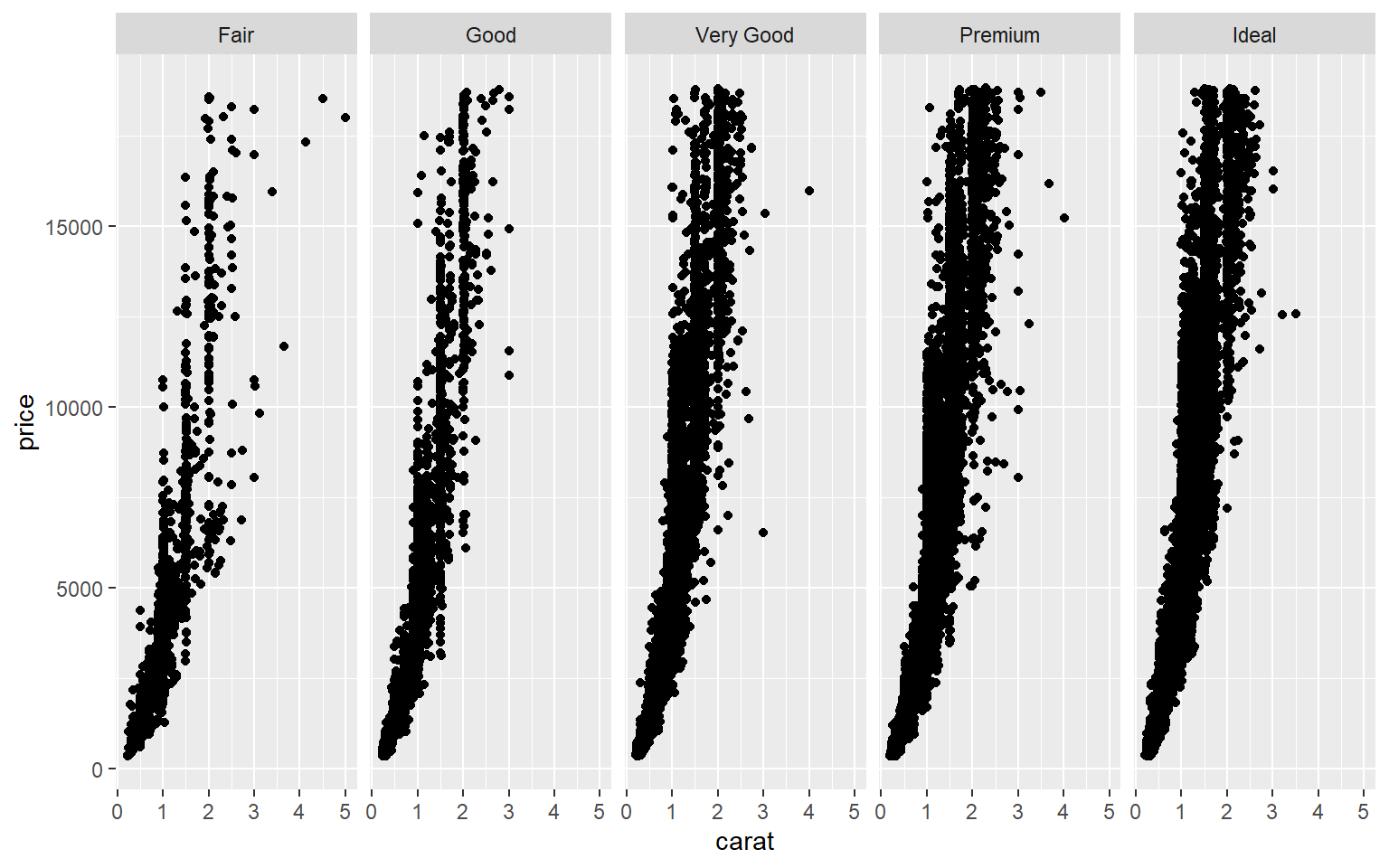
Facets은 여러 집합을 하나의 그림에 표현하기 보다 하위 집합으로 나누어 시각화하는 요소입니다.

diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_point() +

facet\_grid(. ~ cut)



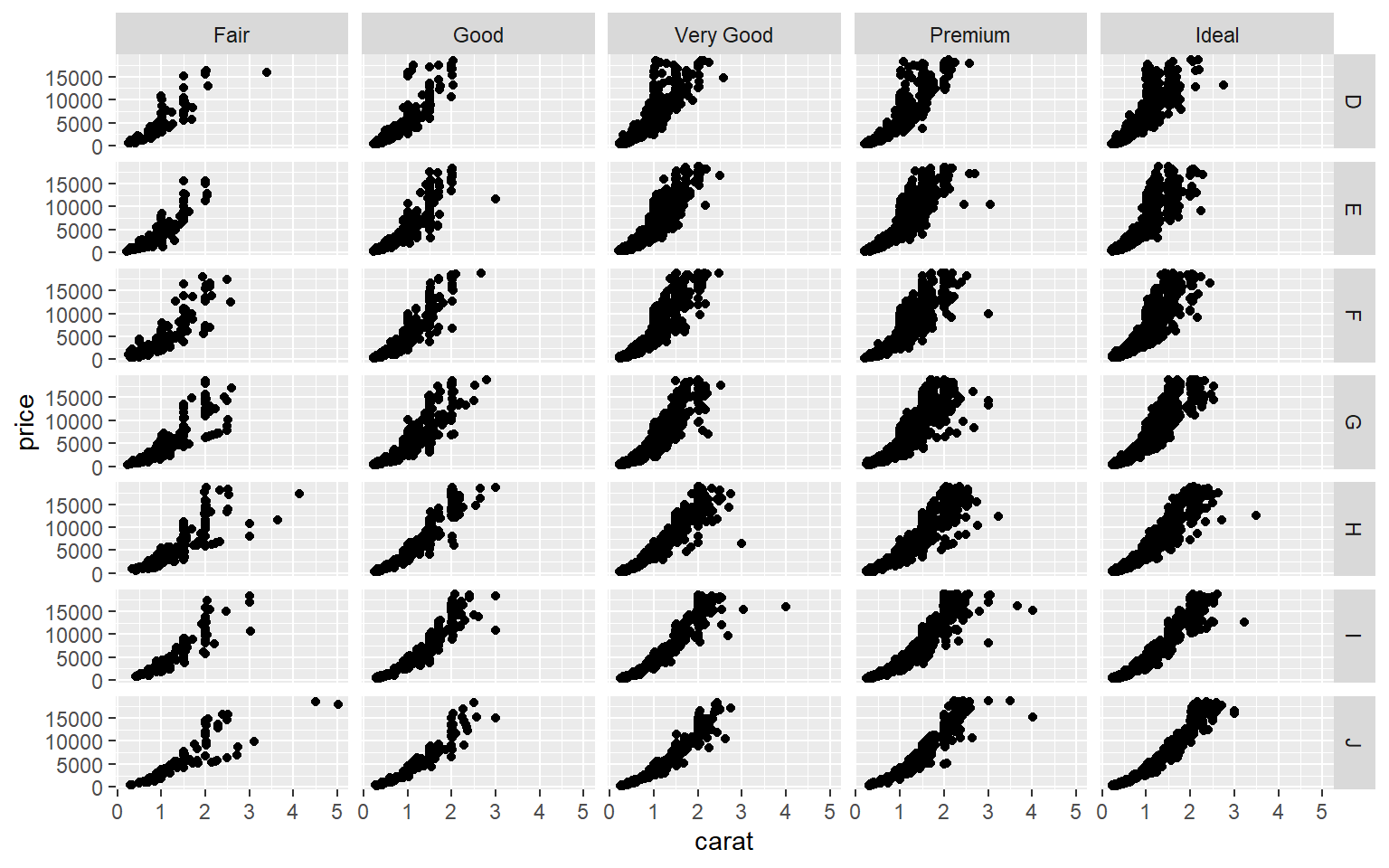
facet\_grid() 혹은 facet\_wrap() 함수를 통해 그림을 분할할 수 있습니다. 물결 표시(~)를 통해 하위 집합으로 나누고자 하는 변수를 선택할 수 있으며, 위 예제에서는 cut에 따라 각기 다른 그림으로 표현되었습니다.

diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_point() +

facet\_grid(color ~ cut)



color를 추가해 facet\_grid(color ~ cut)를 입력하면 가로는 color, 세로는 cut으로 그림이 나누어집니다.

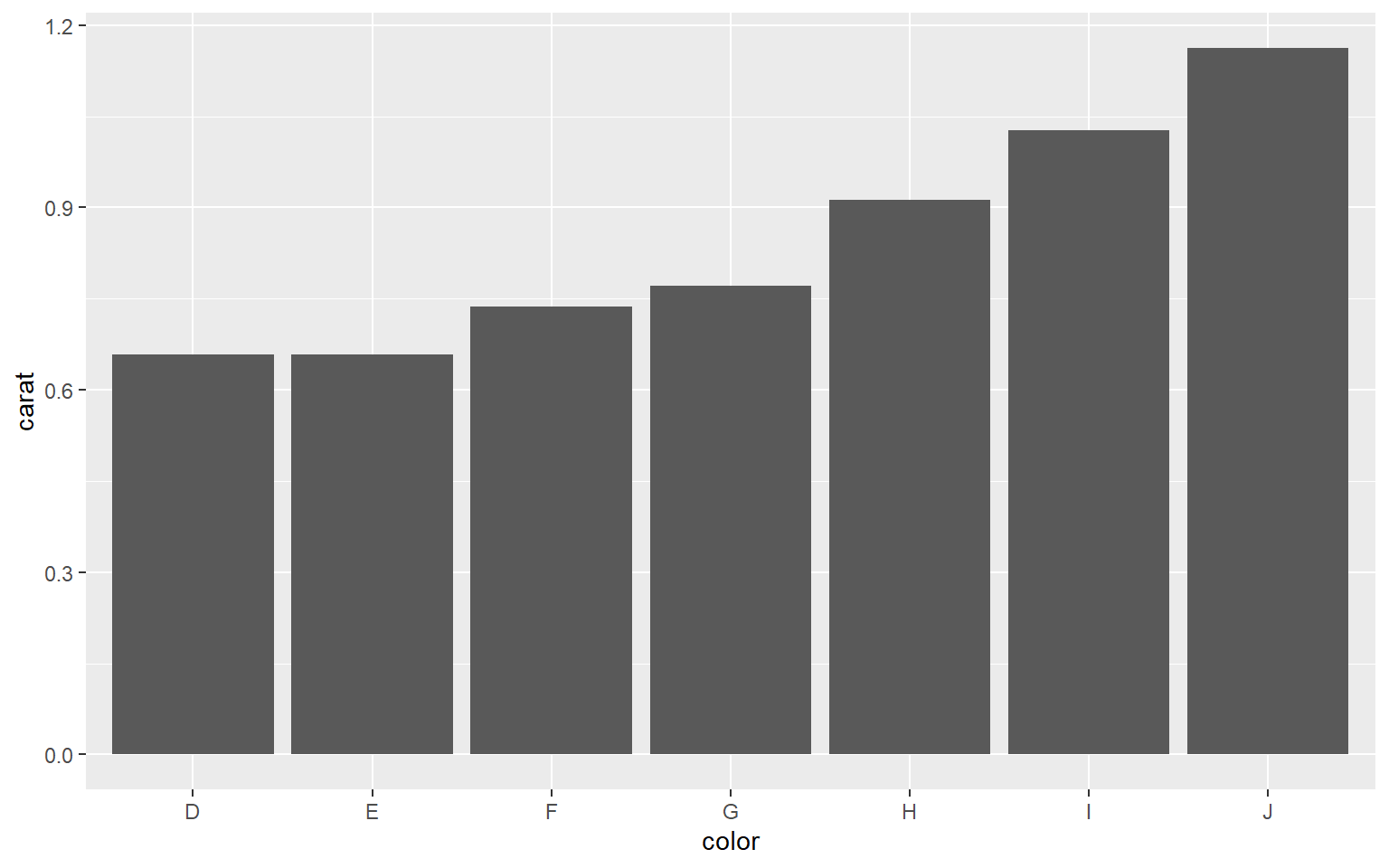
# 5 Statistics

Statistics는 통계값을 나타내는 요소입니다.

diamonds %>%

ggplot(aes(x = color , y = carat)) +

stat\_summary\_bin(fun = "mean", geom = "bar")



결과를 보면 color가 깨끗할 수록(D) 캐럿이 작으며, 더러울 수록(J) 캐럿이 큰 것 처럼 보입니다. 한편, 실무에서는 stat\_\*() 함수를 이용해 통계값을 나타내기 보다는, dplyr 패키지를 이용해 데이터의 통계값을 계산한 후 이를 그림으로 나타냅니다.

library(dplyr)

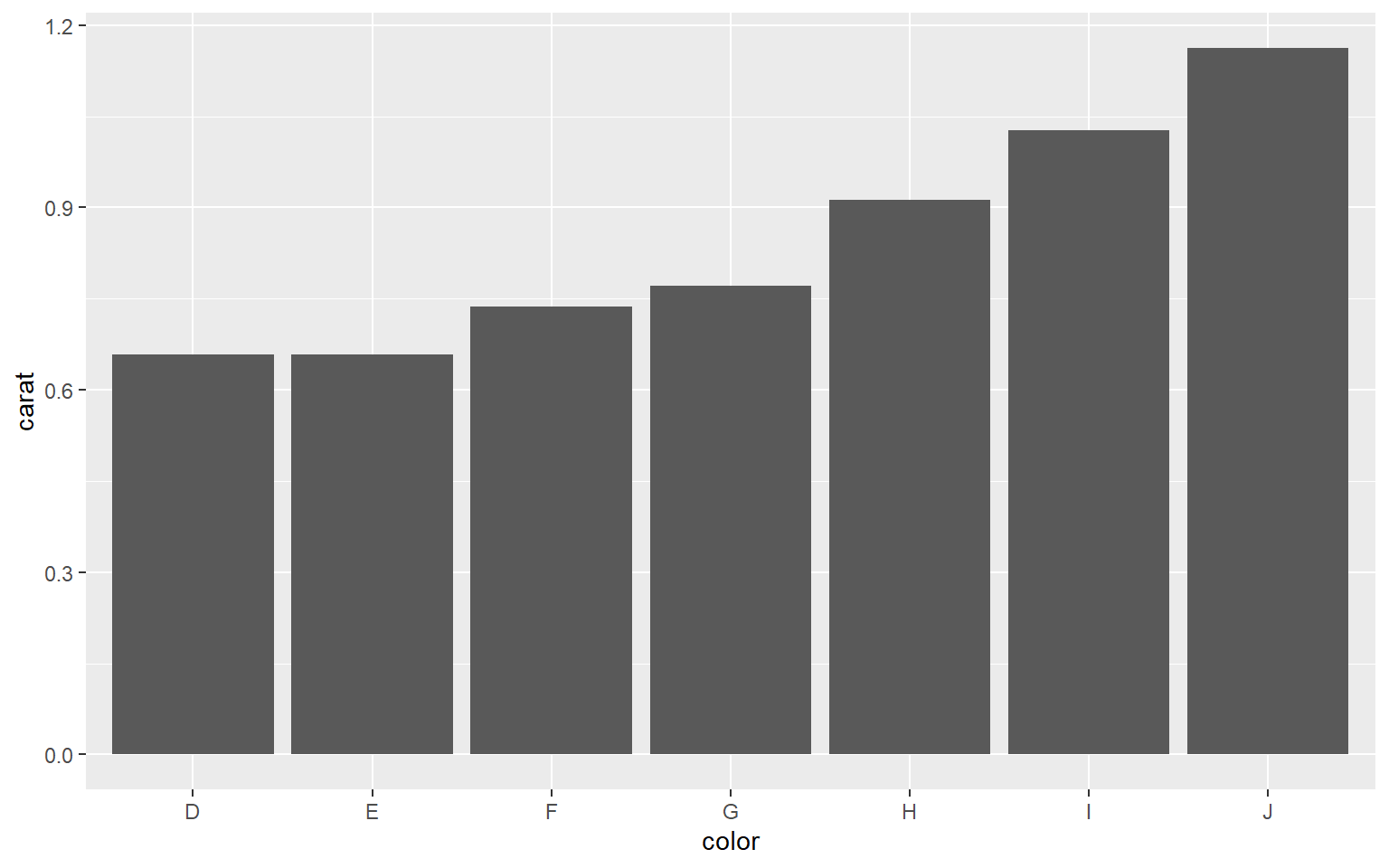
diamonds %>%

group\_by(color) %>%

summarize(carat = mean(carat)) %>%

ggplot(aes(x = color, y = carat)) +

geom\_col()



기존과 그래프가 정확히 일치합니다.

# 6 Coordinates

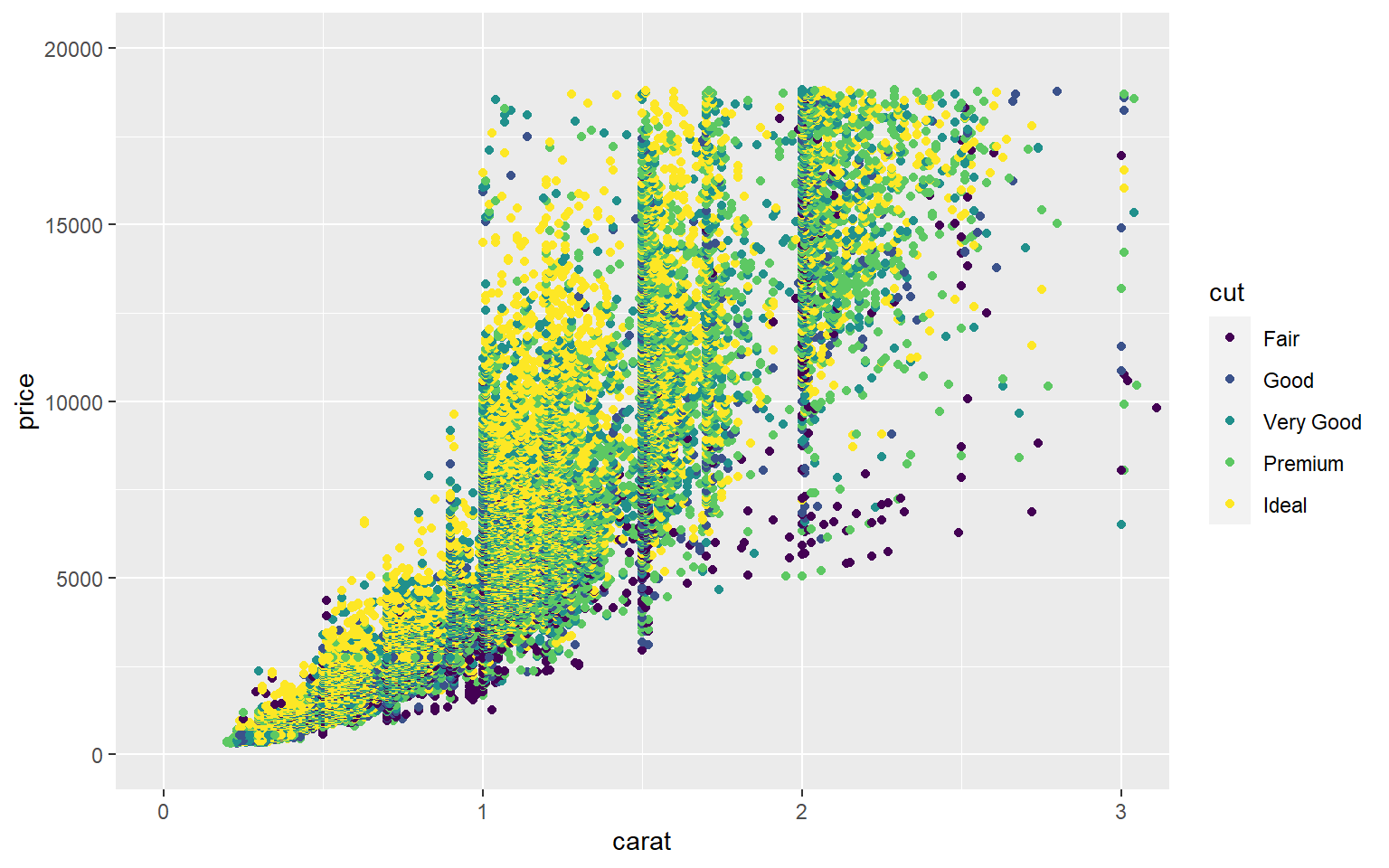
Coordinates는 좌표를 의미합니다. ggplot2에서는 `coord\_\*() 함수를 이용하여 xx축 혹은 yy축 정보를 변형할 수 있습니다.

diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_point(aes(color = cut)) +

coord\_cartesian(xlim = c(0, 3), ylim = c(0, 20000))

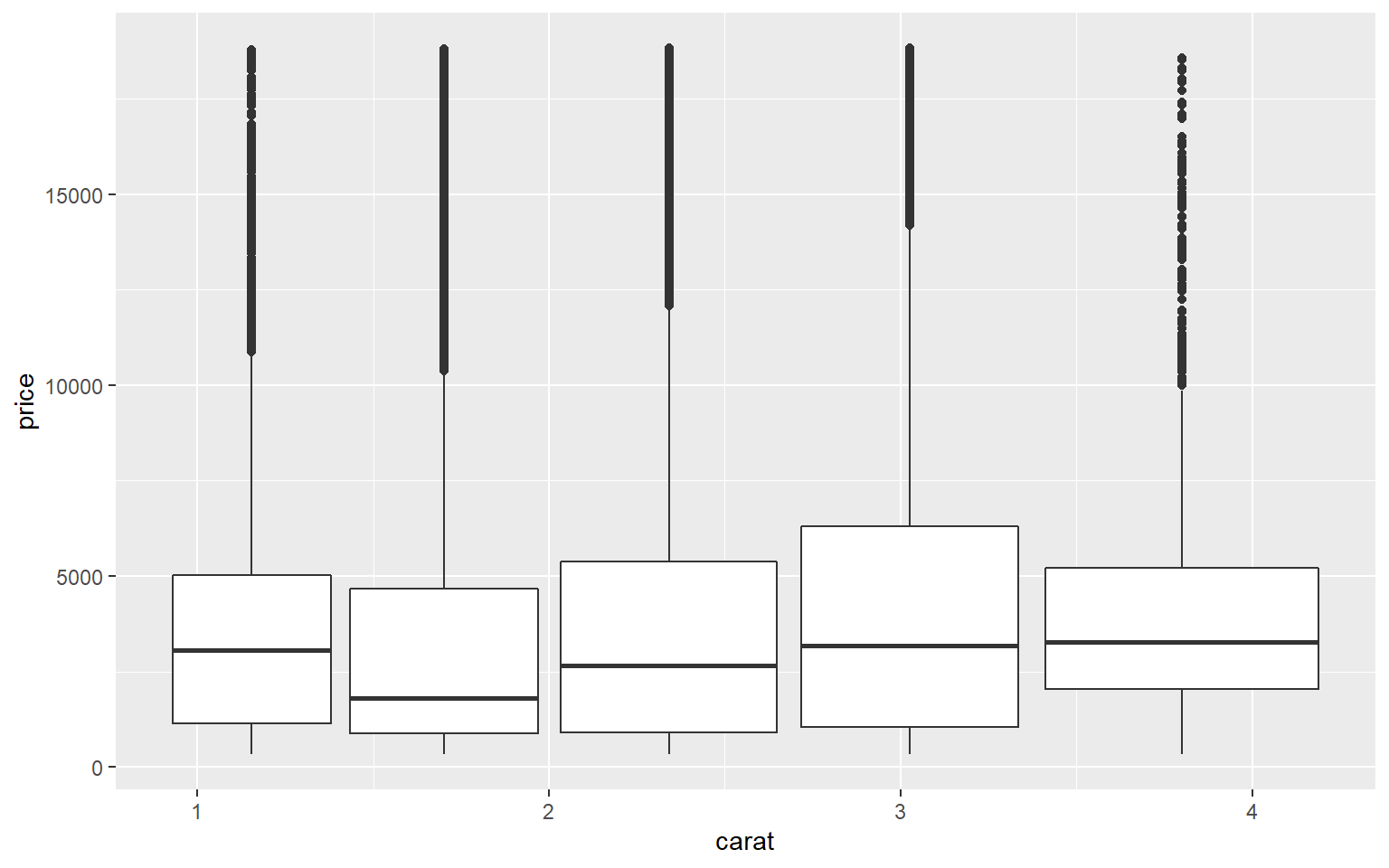


coord\_cartesian() 함수를 통해 xx축과 yy축 범위를 지정해 줄 수 있습니다. xlim과 ylim 내부에 범위의 최소 및 최댓값을 지정해주면, 해당 범위의 데이터만을 보여줍니다.

diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_boxplot(aes(group = cut))



diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_boxplot(aes(group = cut)) +

coord\_flip()



coord\_flip() 함수는 xx축과 yy축을 뒤집어 표현합니다. 위의 그림은 ggplot() 함수의 aes 내부에서 xx축은 carat을, yy축은 price를 지정해 주었지만, 아래 그림에서는 coord\_flip() 함수를 통해 각 축이 서로 바뀌었습니다.

# 7 Theme

Theme은 그림의 제목, 축 제목, 축 단위, 범례, 디자인 등 그림을 꾸며주는 역할을 담당합니다.

diamonds %>%

ggplot(aes(x = carat, y = price)) +

geom\_point(aes(color = cut)) +

theme\_bw() +

labs(title = 'Relation between Carat & Price',

x = 'Carat', y = 'Price') +

theme(legend.position = 'bottom',

panel.grid.major.x = element\_blank(),

panel.grid.minor.x = element\_blank(),

panel.grid.major.y = element\_blank(),

panel.grid.minor.y = element\_blank()

) +

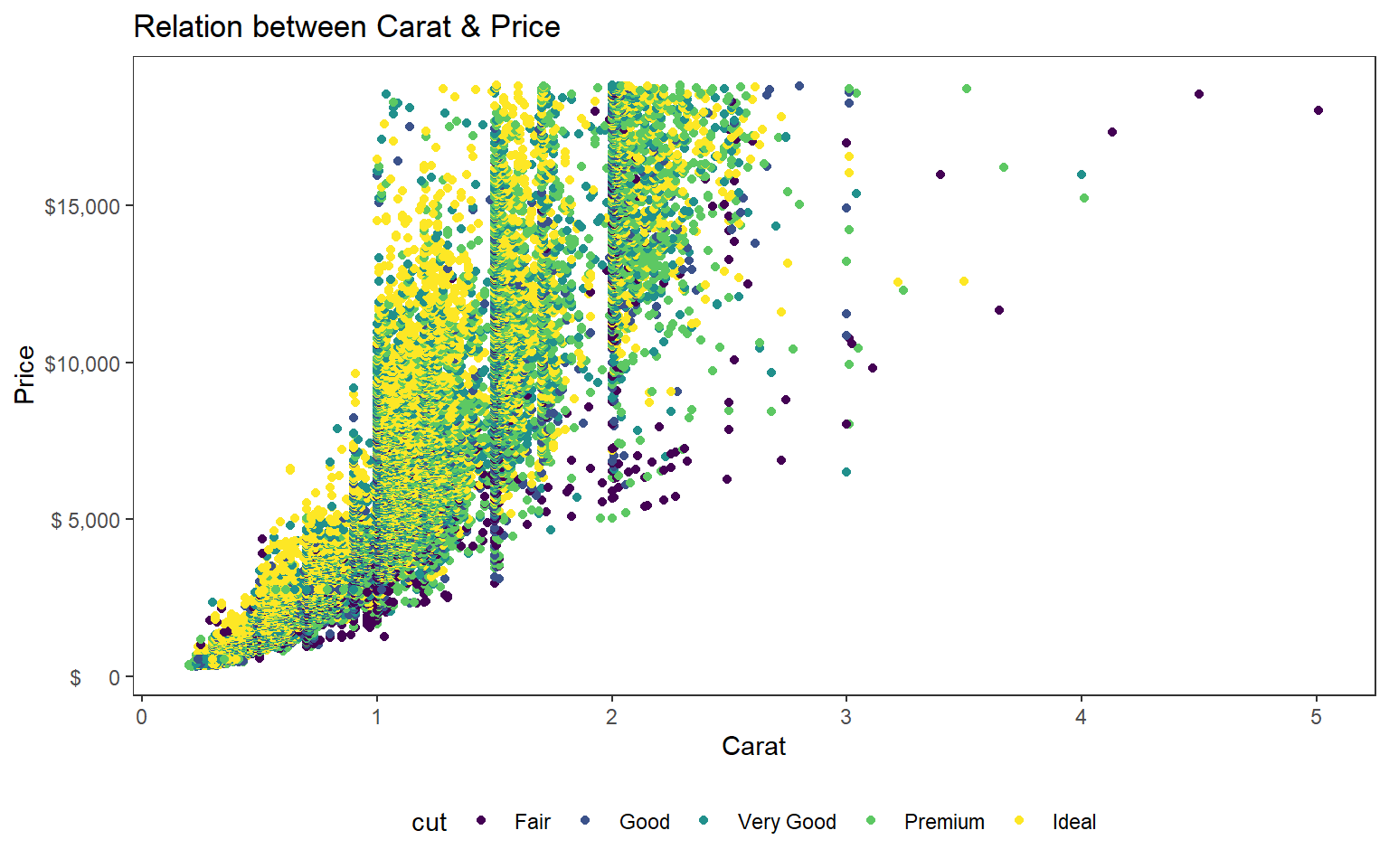
scale\_y\_continuous(

labels = **function**(x) {

paste0('$',

format(x, big.mark = ','))

})



geom\_point() 함수 이후 Theme에 해당하는 부분은 다음과 같습니다.

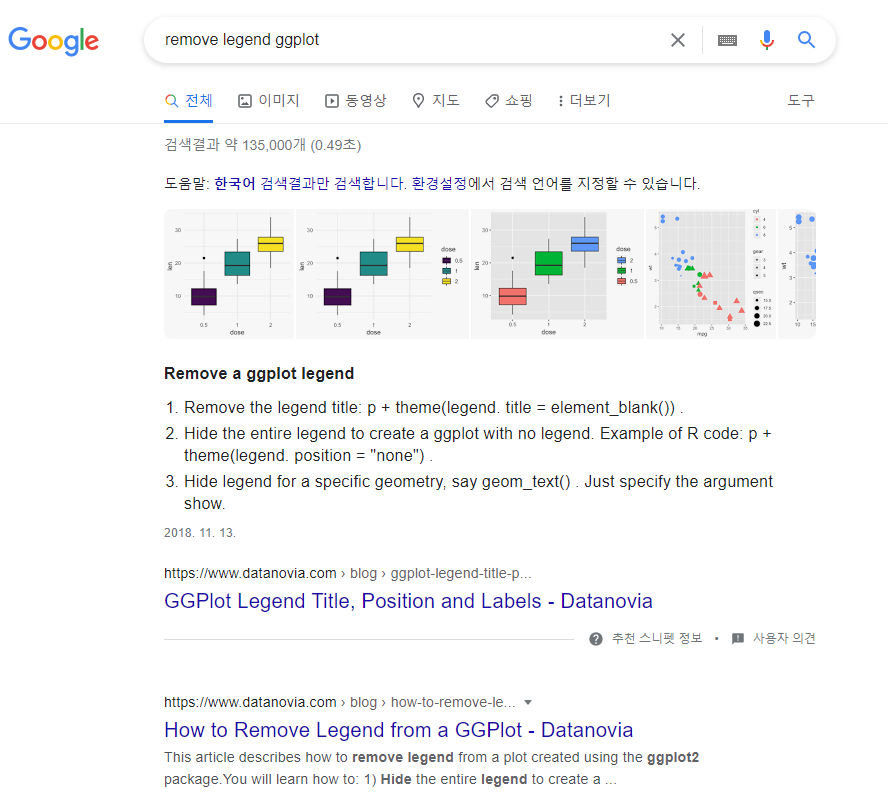
1. theme\_bw() 함수를 통해 배경을 흰색으로 설정합니다.
2. labs() 함수를 통해 그래프의 제목 및 xx축, yy축 제목을 변경합니다.
3. theme() 함수 내 legend.position을 통해 범례를 하단으로 이동합니다.
4. theme() 함수 내 panel.grid를 통해 격자를 제거합니다.
5. scale\_y\_continuous() 함수를 통해 yy축에서 천원 단위로 콤마(,)를 붙여주며, 이를 달러($) 표시와 합쳐줍니다.

이 외에도 각종 테마를 적용해 얼마든지 원하는 그림을 꾸밀 수 있습니다. R에서 적용가능한 그래프의 테마는 다음과 같습니다.

* <https://ggplot2.tidyverse.org/reference/ggtheme.html>

# 8 각종 팁

원하는 형태로 그래프를 가공하고자 할 경우, 구글에 검색을 하면 얼마든지 원하는 답을 얻을 수 있습니다. 만약 범례를 지우고 싶을 경우, 구글에서 ’remove legend ggplot’을 검색하도록 합니다.



*Figure 8.1: 범례 지우기*

이를 통해 우리가 원하는 답을 쉽게 얻을 수 있습니다.

**[데이터 분석 실습]**

R에서는 일반적으로 dplyr 패키지를 이용하여 데이터 분석을 한 후, ggplot2 패키지를 이용하여 시각화로 마무리를 한다.

이번에는 실제 데이터를 이용해 데이터 분석을 하도록 하겠으며, 데이터는 ‘**한국복지패널데이터**’를 이용한다. 해당 데이터는 한국보건사회연구원에서 가구의 경제활동을 연구해 정책지원에 반영할 목적으로 발간하는 조사 자료다. 전국에서 7,000여 가구를 선정해 2006년부터 매년 추적 조사한 자료로, 경제활동, 생활실태, 복지욕구 등 천여 개 변수로 구성되어 있다. 본 데이터는 <https://www.koweps.re.kr:442/data/data/list.do> 에서 다운로드 받을 수 있으며, 강의자료 깃허브에서도 받을 수 있다.

**https://github.com/hyunyulhenry/hyu\_data\_analysis**

**https://github.com/hyunyulhenry/hyu\_data\_analysis/blob/main/welfare/Koweps\_hpda15\_2020\_beta1.1.sav**

해당 데이터는 SPSS 전용 파일로 업로드되어 있으므로, haven 패키지를 통해 불러올 수 있다. 또한 각종 변수명과 조사값이 의미하는 바는 여러 엑셀파일을 통해 확인할 수 있다.

먼저 데이터를 불러오도록 하자.

|  |
| --- |
| library(foreign)  library(dplyr)  library(tidyr)  library(magrittr)  library(ggplot2)  library(readxl)  library(haven)  library(stringr)  library(forcats)  raw = read\_spss('koweps\_hpda15\_2020\_Beta1.1.sav') |

이 중 우리가 사용하려고 하는 **성별, 태어난 연도, 교육 수준, 직종 코드, 지역 코드, 월급** 항목만 선택한다. 각 변수명은 다음 엑셀 파일에 설명되어 있다.

**[15차 머지데이터\_변수명.xlsx]**

|  |
| --- |
| welfare = raw %>% select('h15\_g3',   # 성별                 'h15\_g4',   # 태어난 연도                 'h15\_g6',   # 교육 수준                 'h15\_eco9', # 직종 코드                 'h15\_reg7', # 지역 코드                 'p1502\_8aq1' # 월급                 ) %>%    set\_colnames(c('성별', '연도', '교육', '직종', '지역', '월급')) |

1000여개 변수 중 원하는 변수만을 선택한 후, 변수명을 보기 쉽게 변경하였다.

# 성별에 따른 월급 차이

먼저 남/녀 성별에 따라 월급에 차이가 있는지 살펴보도록 하자.

|  |
| --- |
| welfare %>%    select(성별) %>%    table() |

먼저 남자와 여자 데이터가 몇개가 있는지 살펴본다.

|  |
| --- |
| welfare = welfare %>%    mutate(성별 = if\_else(성별 == 1, '남', '여')) |

남자는 1, 여자는 2로 입력되어 있으며, 이를 이해하기 쉽게 각각 남과 여로 변경한다. 값에 대한 내용은 코드북에 상세하게 나와 있다.

**[(한국복지패널)\_1\_15차\_결합데이터\_코드북\_머지데이터(가구+가구원+부가)(release\_210409).xlsx]**

|  |
| --- |
| welfare %>%    select(성별) %>%    ggplot(aes(x = 성별)) +    geom\_bar() |

이를 간단하게 시각화로 나타낼 수도 있다. 여성의 비율이조금 더 높다.

|  |
| --- |
| welfare %>%    select(월급) %>%    summary() |

이번에는 월급 변수를 확인해보도록 한다. 월급이 0인 경우도 있으며, NA 즉 월급 정보가 없는 데이터도 꽤나 있다. 이는 향후 데이터분석시 제거해야 할 대상이다.

|  |
| --- |
| welfare = welfare %>%    mutate(월급 = ifelse(월급 == 0, NA, 월급)) |

월급이 0인 경우 NA로 변경하도록 한다.

이번에는 성별에 따른 월급의 차이를 분석한다.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(성별) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 성별, y = 평균월급)) +    geom\_col() |

남성의 평균임금이 여성의 평균임금보다 훨씬 높다. (중앙값 선택하기 위해 mean이 아닌 median을 사용)

# 나이에 따른 월급의 관계

이번에는 나이에 따른 월급의 변화를 살펴보자. 데이터는 출생 연도가 들어와 있으므로, 측정시점인 2020년 기준으로 나이를 계산한다.

|  |
| --- |
| welfare = welfare %>%    mutate(나이 = 2020 - 연도 + 1) |

이제 나이에 따른 월급을 살펴보자.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(나이) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 나이, y = 평균월급)) +    geom\_line() +    geom\_vline(xintercept = 45, color = 'red', linetype = 2) +    geom\_vline(xintercept = 60, color = 'red') |

45세 가량 피크를 찍은 후, 점차 감소하는 모습을 보인다. 70세부터는 실질적으로 수입이 없는 모습이다. 이번에는 연령대를 나눠보도록 하겠다. 30세 미만은 초년, 50세 이하는 중년, 그 이상은 노년으로 구분한다.

|  |
| --- |
| welfare = welfare %>%    mutate(연령대 = if\_else(나이 < 30, '초년', if\_else(나이 <= 50, "중년", "노년"))) |

이번에는 연령대 별 월급의 차이를 살펴보겠다.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(연령대) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 연령대, y = 평균월급)) +    geom\_col() +    scale\_x\_discrete(limits = c('초년', '중년', '노년')) |

# 성별, 나이와 월급의 관계

이번에는 성별과 나이에 따른 월급의 관계를 살펴보자.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(성별, 연령대) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 연령대, y = 평균월급, fill = 성별)) +    geom\_col(position = 'dodge') +    scale\_x\_discrete(limits = c('초년', '중년', '노년')) |

전 연령대에서 남성이 여성보다 월급이 많다.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(성별, 나이) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 나이, y = 평균월급, color = 성별)) +    geom\_line() |

각 성별로 구분하여 나이에 따른 월급을 살펴보면, 30세 이후 남성이 여성보다 현격하게 높은 월급을 받는것을 볼 수 있다. 그런데 남성의 경우 90세 가량에서 극단치를 보이는 데이터가 있다. 이를 찾아보도록 하자.

|  |
| --- |
| welfare %>%    filter(성별 == '남' & 나이 >= 85 & 월급 > 400) |

88세 중 한명의 월급이 624로 잡혔다. 샘플이 작은데 극단치가 있으니 평균 통계값이 튀게 나왔던 것이다. 이는 비정상적 데이터로 취급해 제거한 후 다시 살펴보도록 하자.(이는 88세 남성 중 유일하게 월급을 받는 데이터다.)

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    filter(!(성별 == '남' & 나이 >= 85 & 월급 > 400)) %>%    group\_by(성별, 나이) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 나이, y = 평균월급, color = 성별)) +    geom\_line() |

# 학력, 성별에 따른 월급 차이

이번에는 학력과 성별에 따른 월급의 차이를 살펴보겠다. 학력은 1~9개 값으로 나타나 있으며, 각 값이 의미하는 바 역시 코딩북에 나와있다.

|  |
| --- |
| welfare %>%    select(교육) %>%    table() %>%    prop.table() |

먼저 분포를 살펴보도록 하자. **출생년도에 최근으로 올 수록 고학력자가 증가할 것이라 예상해볼 수 있다.**

|  |
| --- |
| welfare = welfare %>%    mutate(교육 = case\_when(      교육 == 1 ~ '1\_미취학(만7세미만)',      교육 == 2 ~ '2\_무학(만7세이상)',      교육 == 3 ~ '3\_초등학교',      교육 == 4 ~ '4\_중학교',      교육 == 5 ~ '5\_고등학교',      교육 == 6 ~ '6\_전문대학',      교육 == 7 ~ '7\_대학교',      교육 == 8 ~ '8\_대학원(석사)',      교육 == 9 ~ '9\_대학원(박사)',      TRUE ~ 'NA'    )    ) |

학력에 따른 월급을 살펴보도록 하자.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(교육) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 교육, y = 평균월급)) +    geom\_col() |

학력이 높으면 높을수록 임금이 높다. 나이대별로 구분하여 살펴보도록 하자.

|  |
| --- |
| welfare %>%    mutate(나이대 = case\_when(      나이 <= 20 ~ '1\_20세이하',      나이 <= 30 ~ '2\_21~30세',      나이 <= 40 ~ '3\_31~40세',      나이 <= 50 ~ '4\_41~50세',      나이 <= 60 ~ '5\_51~60세',      TRUE ~ '6\_60세이상'    )    ) %>%    filter(!is.na(월급)) %>%    group\_by(나이대, 교육) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 교육, y = 평균월급)) +    geom\_col() +    facet\_grid( ~ 나이대) |

대부분의 나이대에서 학력이 높을수록 월급이 높다. 특히 41~60세처럼 임원급이 되면 그 차이가 더욱 벌어진다. 이번에는 성별에 따른 학력과 임금의 차이를 살펴보자.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(성별, 교육) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 교육, y = 평균월급, fill = 성별)) +    geom\_col(position = 'dodge') |

모든 교육수준에서 남성의 임금이 여성보다 높다.

# 성별에 따른 학력 차이

이번에는 성별에 따라 학력에도 차이가 있는지 살펴보도록 하자.

|  |
| --- |
| welfare %>%    group\_by(교육) %>%    summarize(n = n()) %>%    mutate(freq = n / sum(n) \* 100) |

먼저  학력(교육)별 숫자를 구한 후 이를 비율로 환산한다.

|  |
| --- |
| welfare %>%    group\_by(성별, 교육) %>%    summarize(n = n()) %>%    mutate(freq = n / sum(n) \* 100) %>%    select(-n) %>%    pivot\_wider(names\_from = 성별, values\_from = freq) |

이번에는 성별과 교육 별로 비율을 구한 후, 피벗을 통해 표 형태로 나타낸다. 고임금군에 해당하는 대졸 이상의 경우 남성이 여성보다 비율이 높다.

# 직업별 월급차이

이번에는 직업에 따른 월급의 차이를 살펴보자. 직업코드에 따른 직업명 역시 코드북에 나타나 있다.

|  |
| --- |
| job = read\_excel('(한국복지패널)\_1\_15차\_결합데이터\_코드북\_가구데이터(release\_210409).xlsx',                   sheet = '직종코드(14-15차)')  job = job %>% select(1:4) %>%    set\_colnames(c('대분류', '중분류', '소분류', '소분류명')) %>%    fill(대분류, 중분류)  welfare = welfare %>%    mutate(직종 = str\_pad(직종, 4, 'left', 0)) %>%    left\_join(job, by = c('직종' = '소분류')) |

먼저 코드북 엑셀파일 중 직종코드가 담겨있는 엑셀시트를 불러온 후, 필요한 열만 선택한다. 마지막으로 이를 기존의 데이터프레임과 조인한다. (기존 데이터의 경우 직종코드가 4자리로 이루어져 있지 않으므로, 이를 강제로 맞춰준다.)

|  |
| --- |
| income\_by\_job = welfare %>%    filter(!is.na(월급)) %>%    group\_by(소분류명) %>%    summarize(평균월급 = median(월급)) %>%    arrange(desc(평균월급)) %>%    mutate(소분류명 = fct\_reorder(소분류명, 평균월급))  income\_by\_job |

소분류 별로 평균월급을 계산한다. 먼저 상위 30개 직업을 살펴보자.

|  |
| --- |
| income\_by\_job %>%    top\_n(30) %>%    ggplot(aes(x = 소분류명, y = 평균월급)) +    geom\_col() +    coord\_flip() +    xlab('') |

이번에는 하위 30개 직업을 살펴보자.

|  |
| --- |
| income\_by\_job %>%    top\_n(-30) %>%    ggplot(aes(x = 소분류명, y = 평균월급)) +    geom\_col() +    coord\_flip() +    xlab('') |

# 고/저수익 직군의 성별 차이

이번에는 고/저수익 직군에서 성별에 따라 어떠한 차이가 있는지 살펴보도록 하자. 먼저 고수익과 저수익 각각 30개 직군 정보를 저장한다.

|  |
| --- |
| income\_top30 = income\_by\_job %>%    top\_n(30) %>%    select(소분류명) %>%    pull() %>%    as.vector()  income\_bottom30 = income\_by\_job %>%    top\_n(-30) %>%    select(소분류명) %>%    pull() %>%    as.vector() |

먼저 상위 30개 직군의 성별에 따른 임금차이를 살펴보자.

|  |
| --- |
| welfare %>%    filter(소분류명 %in% income\_top30) %>%    filter(!is.na(월급)) %>%    group\_by(소분류명, 성별) %>%    summarize(평균월급 = median(월급)) %>%    arrange(성별, desc(평균월급)) %>%    mutate(소분류명 = fct\_reorder(소분류명, 성별)) %>%    ggplot(aes(x = 소분류명, y = 평균월급, fill = 성별)) +    geom\_col(position = 'dodge') +    coord\_flip() +    xlab('') |

대부분 해당 직업은 남성 근로자만 있으며, 여성의 경우 남성보다 임금이 낮다. 이번에는 하위 30개 직군의 차이를 살펴보자.

|  |
| --- |
| welfare %>%    filter(소분류명 %in% income\_bottom30) %>%    filter(!is.na(월급)) %>%    group\_by(소분류명, 성별) %>%    summarize(평균월급 = median(월급)) %>%    arrange(성별, desc(평균월급)) %>%    mutate(소분류명 = fct\_reorder(소분류명, 성별)) %>%    ggplot(aes(x = 소분류명, y = 평균월급, fill = 성별)) +    geom\_col(position = 'dodge') +    coord\_flip() +    xlab('') |

여성 근로자만 있는 경우가 많으며, 역시나 남성이 여성에 비해 임금이 높다.

# 지역별 차이

이번에는 지역별로 임금과 고령화수준에 어떠한 차이가 있는지 살펴보자. 먼저 고령화수준은 15-64세 생산가능인구 대비 65세 이상 고령인구 비율로 나타나며, 지역에 해당하는 값 역시 코드북에 나타나 있다. 이를 바탕으로 고령화 지표 및 지역명을 만들어준다.

|  |
| --- |
| welfare = welfare %>%    mutate(지표 = if\_else(between(나이, 15, 64), '생산가능인구',                        if\_else(나이 >= 65, '고령', '아동'))) %>%    mutate(지역명 = case\_when(      지역 == 1 ~ '1\_서울',      지역 == 2 ~ '2\_수도권(인천/경기) ',      지역 == 3 ~ '3\_부산/경남/울산',      지역 == 4 ~ '4\_대구/경북',      지역 == 5 ~ '5\_대전/충남',      지역 == 6 ~ '6\_강원/충북',      지역 == 7 ~ '7\_광주/전남/전북/제주도',      TRUE ~ 'NA'    )) |

지역별 임금 차이를 살펴보도록 하자.

|  |
| --- |
| welfare %>%    filter(!is.na(월급)) %>%    group\_by(지역명) %>%    summarize(평균월급 = median(월급)) %>%    ggplot(aes(x = 지역명, y = 평균월급)) +    geom\_col() |

서울, 경기 및 부울경의 임금이 가장 높다. **(대전/충남이 높은 이유는?)** 반면 지방인 대구/경북, 강원/충북, 광주/전남, 전북/제주는 임금이 상대적으로 낮다.

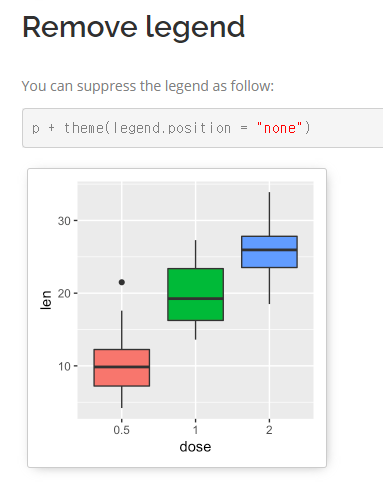
이번에는 지역별 인구 비중을 살펴보자.

|  |
| --- |
| welfare %>%    group\_by(지역명) %>%    summarize(n = n()) %>%    mutate(prop = n / sum(n)) |

서울 경기와 부울경에 절반 가량의 인구가 밀집되어 있다. 마지막으로 고령화수준을 계산하도록 하자.

|  |
| --- |
| welfare %>%    filter(지표 != '아동') %>%    group\_by(지역명, 지표) %>%    summarize(n = n()) %>%    ungroup() %>%    pivot\_wider(names\_from = '지표', values\_from = n) %>%    mutate(고령화수준 = 고령 / 생산가능인구) %>%    ggplot(aes(x = 지역명, y = 고령화수준)) +    geom\_col() +    coord\_flip() +    scale\_x\_discrete(limits=rev) |

인구가 작은 지방의 고령화수준이 상대적으로 높음을 확인할 수 있다.



*Figure 8.2: 범례를 지우는 코드*

또한 ggplot2 패키지 만으로도 나타낼 수 없는 그래프는 다양한 확장 패키지들을 통해 얼마든지 나타낼 수 있습니다.

* <https://exts.ggplot2.tidyverse.org/gallery/>

해당 패키지에 대한 더욱 자세한 설명은 패키지들의 책을 살펴보시기 바랍니다. 해당 책은 온라인에서 무료로 확인할 수 있습니다.

* <https://ggplot2-book.org/>