R_dplyr

2023-03-08

dplyr

dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges

loading dplyr

use library(dplyr)

remember the warnings about replacing base R functions

```
#install.packages("tidyverse")
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 4.2.2
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Common Functions (verbs) of data manupulation

```
mutate() adds new variables that are functions of existing variables
select() picks variables based on their names.
filter() picks cases based on their values.
summarise() reduces multiple values down to a single summary.
arrange() changes the ordering of the rows.
```

These all combine naturally with group_by() which allows you to perform any operation "by group".

All verbs work similarly:

```
The first argument is a data frame.

The subsequent arguments describe what to do with the data frame, using the variable names (w
ithout quotes).

The result is a new data frame.
```

load data for manupulation

Load students mathematics performance data

```
# Load students mathematics performance data
stud.data = read.csv("student-mat.csv")
nrow(stud.data) # print number of rows in data
```

```
## [1] 395
```

```
colnames(stud.data) # print all column names from data
```

```
##  [1] "school"     "sex"        "age"        "address"    "famsize"   
##  [6] "Pstatus"    "Medu"       "Fedu"       "Mjob"       "Fjob"      
## [11] "reason"     "guardian"   "traveltime" "studytime"  "failures"  
## [16] "schoolsup"  "famsup"     "paid"       "activities" "nursery"   
## [21] "higher"     "internet"   "romantic"   "famrel"     "freetime"  
## [26] "goout"      "Dalc"       "Walc"       "health"     "absences"  
## [31] "G1"         "G2"         "G3"
```

Filter rows with filter()

filter() allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame.

Simple condition

```
# filter all students staying in Urban area ( address = 'U')
urban.stud = filter(stud.data , address == 'U')
nrow(urban.stud)
```

```
## [1] 307
```

Match multiple values

```
# filter all students whose father's job is teacher or health
f.stud = filter(stud.data , Fjob %in% c('health' , 'teacher'))
nrow(f.stud)
```

```
## [1] 47
```

Multiple conditions (and &)

```
# filter students who are from urban area and whose father's job is teacher or health
comb.stud = filter(stud.data , address == 'U' &
                   Fjob %in% c('health' , 'teacher'))
nrow(comb.stud)
```

```
## [1] 39
```

Multiple conditions (or |)

```
# filter students who are from urban area or whose Mother is at home
comb.stud = filter(stud.data , address == 'U' |
                   Mjob == 'at_home')
nrow(comb.stud)
```

```
## [1] 329
```

NOTE Common Error The easiest mistake to make is to use = instead of == when testing for equality. When this happens you'll get an informative error.

So use == when you want to compare and = when you want to assign a value

Reference : https://r4ds.had.co.nz/transform.html (https://r4ds.had.co.nz/transform.html)

Arrange rows with arrange()

arrange() changes order of rows. It takes a data frame and a set of column names (or more complicated expressions) to order by.

If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

Use desc() to re-order by a column in descending order

Missing values are always sorted at the end

ascending arrange

```
# arange students ascending by their age
age.stud = arrange(stud.data , age)
age.stud$age
```

```
##   [1] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
##  [26] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
##  [51] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
##  [76] 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [101] 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [126] 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [151] 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [176] 16 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [201] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [226] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [251] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [276] 17 17 17 17 17 17 17 17 17 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
## [301] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
## [326] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
## [351] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19 19
## [376] 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 20 20 20 21 22
```

descending arrange

```
# arrange students descending by their age
age.stud = arrange(stud.data , desc(age))
age.stud$age
```

```
##   [1] 22 21 20 20 20 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
##  [26] 19 19 19 19 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
##  [51] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
##  [76] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
## [101] 18 18 18 18 18 18 18 18 18 18 18 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [126] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [151] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [176] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
## [201] 17 17 17 17 17 17 17 17 17 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [226] 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [251] 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [276] 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
## [301] 16 16 16 16 16 16 16 16 16 16 16 16 16 15 15 15 15 15 15 15 15 15 15 15 15
## [326] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
## [351] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
## [376] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
```

arrange by multiple columns

```
# arrange students ascending by their age and descending by father's education
age.stud = arrange(stud.data , age, desc(Fedu) )
head(age.stud[c('age','Fedu')])
```

```
##   age Fedu
## 1  15    4
## 2  15    4
## 3  15    4
## 4  15    4
## 5  15    4
## 6  15    4
```

Select columns with select()

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. select() allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

options in select

starts_with –> match pattern to start of column name

ends_with –> match pattern to end of column name

contains –> match a pattern in complete column name

matches –> find matching columns using regular expressions

num_range("y",1:5) –> y1,y2,y3,y4,y5

one_of –> select one of the columns of array

any_of –> used to check if the column is present or not

```
colnames(stud.data)
```

```
##  [1] "school"     "sex"        "age"        "address"    "famsize"
##  [6] "Pstatus"    "Medu"       "Fedu"       "Mjob"       "Fjob"
## [11] "reason"     "guardian"   "traveltime" "studytime"  "failures"
## [16] "schoolsup"  "famsup"     "paid"       "activities" "nursery"
## [21] "higher"     "internet"   "romantic"   "famrel"     "freetime"
## [26] "goout"      "Dalc"       "Walc"       "health"     "absences"
## [31] "G1"         "G2"         "G3"
```

In following example columns school and all columns starting with "G" are selected

```
col.select = select(stud.data, school, starts_with("G"))
names(col.select)
```

```
## [1] "school"   "guardian" "goout"    "G1"       "G2"       "G3"
```

Add new variables with mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of mutate().

mutate() always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables.

In following example new column total is added at end Total is total grades, sum of G1, G2 and G3.

```
col.select = select(stud.data, school, starts_with("G"))
col.select = mutate(col.select, total = G1 + G2 + G3)
names(col.select)
```

```
## [1] "school"   "guardian" "goout"    "G1"       "G2"       "G3"       "total"
```

Useful creation functions

There are many functions for creating new variables that you can use with mutate(). The key property is that the function must be vectorised. it must take a vector of values as input, return a vector with the same number of values as output. There's no way to list every possible function that you might use, but here's a selection of functions that are frequently useful:

- Arithmetic operators: +, -, *, /, ^.

These are all vectorised, using the so called "recycling rules". If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: air_time / 60, hours * 60 + minute, etc.

Arithmetic operators are also useful in conjunction with the aggregate functions you'll learn about later. For example, x / sum(x) calculates the proportion of a total, and y - mean(y) computes the difference from the mean.

- Logical comparisons, <, <=, >, >=, !=, and ==

- Modular arithmetic: %/% (integer division) and %% (remainder),

where x == y * (x %/% y) + (x %% y).

Modular arithmetic is a handy tool because it allows you to break integers up into pieces. For example, you can compute hour and minute from time variable

- Ranking:

there are a number of ranking functions, but you should start with min_rank(). It does the most usual type of ranking (e.g. 1st, 2nd, 2nd, 4th). The default gives smallest values the small ranks; use desc(x) to give the largest values the smallest ranks.

Grouped summaries with summarise()

The last key verb is summarise(). It collapses a data frame to a single row.

summarise() is not terribly useful unless we pair it with group_by(). This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the dplyr verbs on a grouped data frame they'll be automatically applied "by group".

```
# Find the average grades in G1 for each level of mother's education
medu.groups = group_by(stud.data, Medu)
summarize(medu.groups, avg.G1 = mean(G1))
```

```
## # A tibble: 5 × 2
##    Medu avg.G1
##   <int>  <dbl>
## 1     0  12
## 2     1   9.75
## 3     2  10.6
## 4     3  10.6
## 5     4  11.9
```

Useful summarize functions

min(x) - minimum value of vector x .

max(x) - maximum value of vector x .

mean(x) - mean value of vector x .

median(x) - median value of vector x .

quantile(x, p) - pth quantile of vector x .

sd(x) - standard deviation of vector x .

var(x) - variance of vector x .

IQR(x) - Inter Quartile Range (IQR) of vector x .

diff(range(x)) - total range of vector x .

special functions

first(x) - The first element of vector x .

last(x) - The last element of vector x .

nth(x, n) - The nth element of vector x .

n() - The number of rows in the data.frame or group of observations that summarise() describes.

n_distinct(x) - The number of unique values in vector

Combining multiple operations with the pipe

pipe operator

%>% is pipe operator in dplyr

Here output of first operation is passed as input to next step

for example x %>% f(y) means x is passed to function f(y)

Behind the scenes, x %>% f(y) turns into f(x, y), and x %>% f(y) %>% g(z) turns into g(f(x, y), z) and so on.

Advantages of pipe operator

- avoid complex codes
- avoids use of lot of temporary variables

Example of pipe usage

```
# Find the average grades in G1 for each level of mother's education
stud.data %>%
 group_by(Medu) %>%
  summarize(avg.G1 = mean(G1))
```

```
## # A tibble: 5 × 2
##    Medu avg.G1
##   <int>  <dbl>
## 1     0  12
## 2     1   9.75
## 3     2  10.6
## 4     3  10.6
## 5     4  11.9
```