

Q&A Vector-Subsetting-Iteration-Function

Question

Subsetting:

How can I subset a vector based on a condition that involves a function, like selecting all values greater than the mean of the vector?

If we have a vector `v`, we can calculate its mean using `mean(v)`. To find values in `v` that are greater than its mean, you can use `v[v > mean(v)]`.

Example:

Consider the vector `v` from 1 to 10. Its mean is 5.5.

```
v <- c(1:10)
v[v > mean(v)]
```

```
[1] 6 7 8 9 10
```

Question

Subsetting

Names are so special, that there are special ways to create them and view them

```
x <- c(a = 1, b = 2, c = 3)
x
```

```
a b c
1 2 3
```

```
names(x)
```

```
[1] "a" "b" "c"
```

Or we can create it as follow

```
y <- 11:13  
names(y) <- c("A", "B", "C")  
y
```

```
  A B C  
11 12 13
```

```
names(y)
```

```
[1] "A" "B" "C"
```

- You can remove names with `unname()` function from `{base}` package.

```
unname(x) -> z  
z
```

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

```
names(z)
```

NULL

- Names stay with single bracket `[]` subsetting

```
x
```

```
a b c  
1 2 3
```

```
names(x[1])
```

```
[1] "a"
```

```
names(x[1:2])
```

```
[1] "a" "b"
```

Not double bracket subsetting [[]]

```
names(x[[1]])
```

NULL

- Names can be used for subsetting (more in Chapter 4)

```
x[["a"]]
```

```
[1] 1
```

Difference between [], [[]] and \$

Sometimes you want just part of an object. In some cases you will use square [] brackets or double square [[]] brackets, and in other cases you will use a dollar sign \$.

Extracting elements from a vector

```
x <- seq(from = 5, to = 50, by = 5)
```

```
x
```

```
[1]  5 10 15 20 25 30 35 40 45 50
```

we can, for example, extract the 2nd element,

```
x[2]
```

```
[1] 10
```

we can also do it by

```
x[[2]]
```

```
[1] 10
```

Elements 3 to 6,

```
x[3:6]
```

```
[1] 15 20 25 30
```

Elements 2, 3, 5, 8

```
x[c(2, 3, 5, 8)]
```

```
[1] 10 15 25 40
```

All elements but 7

```
x[-7]
```

```
[1] 5 10 15 20 25 30 40 45 50
```

All values between 15 and 30, including 15

```
x[x<30 & x>= 15]
```

```
[1] 15 20 25
```

You can extract elements of a vector by using logical.

```
x[c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE, TRUE, FALSE, FALSE)]
```

```
[1] 5 10 15 25 40
```

Extracting elements from a matrix or array

Similarly, we can extract elements from a matrix or array, but now we need *multiple indices* separated by *commas*. For example, given the following 2-dimensional matrix x,

```
x <- matrix(c(10,12,31,14,51,60), nrow = 2, ncol = 3)
x
```

```
      [,1] [,2] [,3]
[1,]   10   31   51
[2,]   12   14   60
```

Extract the element in the 2nd row and 3rd column

```
x[2, 3]
```

```
[1] 60
```

Extract the second row

```
x[2,]
```

```
[1] 12 14 60
```

Extract the third

```
x[, 3]
```

```
[1] 51 60
```

If you leave out the comma, you will get an answer not an error. For example:

```
x[3]
```

```
[1] 31
```

But it is ambiguous to say “*the 3rd element of a matrix*” since you could go *down columns* or *across rows*. R has a default, but rather than try to remember what that is, just **do not forget the comma and then there is no ambiguity**.

- **ARRAY**

```
z <- array(c(10,12,31,14,51,60, 7, 53), dim = c(2,2,2))
z
```

```
, , 1
```

```
      [,1] [,2]
[1,]   10   31
[2,]   12   14
```

```
, , 2
```

```
      [,1] [,2]
[1,]   51    7
[2,]   60   53
```

we can extract a single element,

```
z[2,1,2]
```

```
[1] 60
```

A sub-vector,

```
z[1,,2]
```

```
[1] 51  7
```

Or a sub-matrix,

```
z[,2,] # Second columns of any row and any of two sub-matrices. Just remember by default it is
```

```
      [,1] [,2]
[1,]   31    7
[2,]   14   53
```

```
z[, ,2] # The Second sub-matrices
```

```
      [,1] [,2]
[1,]   51    7
[2,]   60   53
```

```
z[2,,] # The Second rows of any columns and any of two sub-matrices. Just remember by default
```

```
      [,1] [,2]
[1,]   12   60
[2,]   14   53
```

Extracting elements from a list

For a list, you can use single square [] brackets or double square [[]] brackets, depending on what you want to extract.

```
x <- list("5", c(1,2,3), factor(c("BMW", "FORD", "GM", "FORD", "JEEP", "BMW", "FORD")))
x
```

```
[[1]]
[1] "5"
```

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

```
[[3]]
[1] BMW FORD GM FORD JEEP BMW FORD
Levels: BMW FORD GM JEEP
```

- We can use [] to extract a sub-list containing only, for example, the second element,

```
x[1]
```

```
[[1]]
[1] "5"
```

```
class(x[1])
```

```
[1] "list"
```

```
x[2]
```

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

```
class(x[2])
```

```
[1] "list"
```

```
x[3]
```

```
[[1]]  
[1] BMW FORD GM FORD JEEP BMW FORD  
Levels: BMW FORD GM JEEP
```

```
class(x[3])
```

```
[1] "list"
```

or multiple elements,

```
x[c(1, 3)]
```

```
[[1]]  
[1] "5"  
  
[[2]]  
[1] BMW FORD GM FORD JEEP BMW FORD  
Levels: BMW FORD GM JEEP
```

```
class(x[c(1, 3)])
```

```
[1] "list"
```

- Or we can use `[[]]` to extract a single element, which will have the class of that element.


```
x[[1]]
```

```
[1] "5"
```

```
class(x[[1]])
```

```
[1] "character"
```

```
x[[2]]
```

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

```
class(x[[2]])
```

```
[1] "numeric"
```

```
x[[3]]
```

```
[1] BMW FORD GM FORD JEEP BMW FORD  
Levels: BMW FORD GM JEEP
```

```
class(x[[3]])
```

```
[1] "factor"
```

```
x[[1]][1]
```

```
[1] "5"
```

```
class(x[[1]][1])
```

```
[1] "character"
```

```
x[[1]][2]
```

```
[1] NA
```

```
class(x[[1]][2])
```

```
[1] "character"
```

```
x[[2]][1]
```

```
[1] 1
```

```
class(x[[2]][1])
```

```
[1] "numeric"
```

```
x[[2]][2]
```

```
[1] 2
```

```
x[[2]][3]
```

```
[1] 3
```

```
x[[3]][1]
```

```
[1] BMW  
Levels: BMW FORD GM JEEP
```

```
class(x[[3]][1])
```

```
[1] "factor"
```

```
x[[3]][2]
```

```
[1] FORD  
Levels: BMW FORD GM JEEP
```

```
x[[3]][3]
```

```
[1] GM  
Levels: BMW FORD GM JEEP
```

```
x[[3]][4]
```

```
[1] FORD  
Levels: BMW FORD GM JEEP
```

```
x[[3]][5]
```

```
[1] JEEP  
Levels: BMW FORD GM JEEP
```

```
class(x[[3]][5])
```

```
[1] "factor"
```

Extracting elements from a data frame

Recall that a data.frame is special type of list where each element is one of the columns. You can access the elements of a data.frame in a number of ways, including the \$ method.

```
df <- data.frame(outcome = c(1, 0, 1, 1),  
                 exposure = factor(c("yes", "yes", "no", "no"),  
                                   levels = c("no", "yes"),  
                                   labels = c("No", "Yes")),  
                 age      = c(24, 55, 39, 18))  
df
```

	outcome	exposure	age
1	1	Yes	24
2	0	Yes	55
3	1	No	39
4	1	No	18

- we can extract a data.frame made up of a subset of columns using [],

```
df[1:2]
```

	outcome	exposure
1	1	Yes
2	0	Yes
3	1	No
4	1	No

```
class(df[1:2])
```

```
[1] "data.frame"
```

```
df[c("outcome", "exposure")]
```

	outcome	exposure
1	1	Yes
2	0	Yes
3	1	No
4	1	No

- A single column of the data.frame, returned as the class of that column, using `[[]]` or `$`.

```
df[3]
```

	age
1	24
2	55
3	39
4	18

```
class(df[3])
```

```
[1] "data.frame"
```

```
df[[3]]
```

```
[1] 24 55 39 18
```

```
class(df[[3]])
```

```
[1] "numeric"
```

```
df["age"]
```

```
   age
1   24
2   55
3   39
4   18
```

```
class(df["age"])
```

```
[1] "data.frame"
```

```
df[["age"]]
```

```
[1] 24 55 39 18
```

```
class(df[["age"]])
```

```
[1] "numeric"
```

```
df$age
```

```
[1] 24 55 39 18
```

```
class(df$age)
```

```
[1] "numeric"
```

When using the `$` method, if the variable name has *spaces*, then enclose it in (not regular quotes) when extracting it. To illustrate, let's change the names of this data.frame by assigning a new value to its `names()`.

```
names(df) <- c("Outcome Level", "Exposure", "Age")
```

```
df
```

	Outcome Level	Exposure	Age
1	1	Yes	24
2	0	Yes	55
3	1	No	39
4	1	No	18

```
df$`Outcome Level`
```

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

The double and single quotation still is working. But R studio by default wraps the column names with the backticks.

```
df$"Outcome Level"
```

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

```
df$('Outcome Level')
```

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

- You can also extract elements of a data.frame using matrix indexing.

```
df[,1]
```

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

```
class(df[,1])
```

```
[1] "numeric"
```

the first row of df, returning a data.frame with 1 row,

```
df[1,]
```

	Outcome	Level	Exposure	Age
1		1	Yes	24

```
class(df[1,])
```

```
[1] "data.frame"
```

```
df[2,]
```

	Outcome	Level	Exposure	Age
2		0	Yes	55

or the first column of x, returning a vector of the class of that column,

```
df[,1]
```

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

```
class(df[,1])
```

```
[1] "numeric"
```

or a data.frame if you include `drop=F`.

```
df[,1,drop=F]
```

	Outcome	Level
1		1
2		0
3		1
4		1

```
class(df[,1,drop=F])
```

```
[1] "data.frame"
```

If extracting more than 1 column, `drop=F` is not necessary to return a data.frame.

```
df[,2:3]
```

	Exposure	Age
1	Yes	24
2	Yes	55
3	No	39
4	No	18

```
class(df[,2:3])
```

```
[1] "data.frame"
```

In any of these column extraction via matrix-subsetting examples, you can use the column names.

```
df[, "Outcome Level", drop=F]
```

	Outcome Level
1	1
2	0
3	1
4	1

```
class(df[, "Outcome Level", drop=F])
```

```
[1] "data.frame"
```

- Some data.frame objects have `rownames`. By default, R just assigns numbers.

```
rownames(df)
```

```
[1] "1" "2" "3" "4"
```

But suppose we have a data.frame with, say, participant IDs as the row names.

```
rownames(df) <- c("B239", "B211", "B101", "B439")
df
```


	Outcome	Level	Exposure	Age
B239		1	Yes	24
B211		0	Yes	55
B101		1	No	39
B439		1	No	18

Then you can subset rows using row names.

```
df[c("B211", "B439"),]
```

	Outcome	Level	Exposure	Age
B211		0	Yes	55
B439		1	No	18

You can also subset rows of a data.frame using logical statements about the values in the data.frame.

```
df[df$Exposure == "Yes",]
```

	Outcome	Level	Exposure	Age
B239		1	Yes	24
B211		0	Yes	55

Question:

How to handle missing values?

In R, missing values are typically represented as `NA`. To check for missing values in variables, we use R's `is.na()` function. To find available values, we negate this function.

- **Question:**

Determine the number of missing values and the available items.

```
df <- tibble(
  C1 = c(1, 2, NA, 4, 5, 6),
  C2 = c(NA, 2, 3, NA, 5, 6),
  C3 = c(1, NA, 3, 4, NA, 6)
)
missing_values <- is.na(df)
sum(missing_values) -> n_miss
cat("The number of missing values are ", n_miss)
```

The number of missing values are 5

```
n_available_items <- nrow(df)*ncol(df) - n_miss  
cat("\n \n The number of available items is determined by multiplying the observations by the number of columns")
```

The number of available items is determined by multiplying the observations by the number of columns

Handling missing values depends on the context and the nature of your data. Let's explore two common options:

1. Removing Missing Values (Deletion):

- **Pros:**
 - Simple and straightforward.
 - Avoids imputing potentially incorrect values.
- **Cons:**
 - Reduces the sample size.
 - May lead to biased results if missingness is not random.
- **When to Use:**
 - If the proportion of missing values is small and randomly distributed.
 - If you can afford to lose some data.

- **I. Removes the rows that contain NA.**

```
dfn <- drop_na(df)  
dfn
```

```
# A tibble: 1 x 3  
  C1    C2    C3  
<dbl> <dbl> <dbl>  
1     6     6     6
```

Using `drop_na()` Function of `{tidyr}` Package.

- **II. This will remove rows only if they have NA in C1**

```
drop_na(df, C1)
```

```
# A tibble: 5 x 3
      C1     C2     C3
  <dbl> <dbl> <dbl>
1     1     NA     1
2     2     2     NA
3     4     NA     4
4     5     5     NA
5     6     6     6
```

- III. This will remove rows only if they have NA in either C1 or C3. Rows with NA in C2 will be retained.

```
drop_na(df, C1, C3)
```

```
# A tibble: 3 x 3
      C1     C2     C3
  <dbl> <dbl> <dbl>
1     1     NA     1
2     4     NA     4
3     6     6     6
```

2. Imputing with Mean (or Other Measures):

- **Pros:**
 - Retains the entire dataset.
 - Preserves statistical power.
- **Cons:**
 - Assumes that missing values are missing at random.
 - May introduce bias if the mean is not representative.
- **When to Use:**
 - If the proportion of missing values is significant.

```
df_1 <- tibble(A = c(2, 2, NA, 10, 20, NA, 3),
               B = c(1, NA, 5, NA, 8, 9, NA),
               C = c(NA, 4, 6, 7, NA, 2, 3)
)
missing_values <- is.na(df_1)
sum(missing_values) -> n_miss
cat("The number of missing values are ", n_miss)
```

The number of missing values are 7

```
n_available_items <- nrow(df)*ncol(df) - n_miss
cat("\n \n The number of available items is determined by multiplying the observations by the
```

The number of available items is determined by multiplying the observations by the number of

Replacing missing value with mean of each column

```
df_1$A[is.na(df_1$A)] <- mean(df_1$A, na.rm = TRUE)
df_1$B[is.na(df_1$B)] <- mean(df_1$B, na.rm = TRUE)
df_1$C[is.na(df_1$C)] <- mean(df_1$C, na.rm = TRUE)

df_1
```

```
# A tibble: 7 x 3
      A      B      C
  <dbl> <dbl> <dbl>
1     2     1   4.4
2     2   5.75    4
3    7.4     5     6
4    10   5.75     7
5    20     8   4.4
6    7.4     9     2
7     3   5.75     3
```

Question:

How to handle missing values when using functions like `map()` and `apply()` from `{purrr}` package?

In R, the `apply()` function belongs to the `{base}` package, but here we're utilizing functions from the `{purrr}` package. As you know, there are numerous approaches to writing code. In this course, we've opted to prioritize the `tidyverse` package. This choice is often more efficient and straightforward.

- The square root of NA comes as NA

```
v <- c( 1:3, NA, 2:5, NA)
map(v, sqrt)
```

```
[[1]]
[1] 1
```

```
[[2]]
[1] 1.414214
```

```
[[3]]
[1] 1.732051
```

```
[[4]]
[1] NA
```

```
[[5]]
[1] 1.414214
```

```
[[6]]
[1] 1.732051
```

```
[[7]]
[1] 2
```

```
[[8]]
[1] 2.236068
```

```
[[9]]
[1] NA
```

- Remove NA then compute square root

```
v[!is.na(v)] ->
  v_n
```

```
map(v_n, sqrt)
```

```
[[1]]
[1] 1
```

```
[[2]]
```

```
[1] 1.414214
```

```
[[3]]
```

```
[1] 1.732051
```

```
[[4]]
```

```
[1] 1.414214
```

```
[[5]]
```

```
[1] 1.732051
```

```
[[6]]
```

```
[1] 2
```

```
[[7]]
```

```
[1] 2.236068
```

Most functions in R include an `na.rm` argument, which, when set to `TRUE`, removes `NA` values before computation.

```
l1 <- list(c(1, 2, NA, 4), c(6, NA, 3, NA))  
map(l1, sum, na.rm = TRUE)
```

```
[[1]]
```

```
[1] 7
```

```
[[2]]
```

```
[1] 9
```

Find the mean

```
map(l1, mean, na.rm = TRUE)
```

```
[[1]]
```

```
[1] 2.333333
```

```
[[2]]
```

```
[1] 4.5
```

Question

Anonymous functions

In R, functions are like objects themselves. They don't automatically come with a name attached. If you don't give it a name, it becomes an anonymous function.

Anonymous functions are used when you don't find it necessary to name them.

```
df <- tibble(  
  C1 = c(1, 2, 3, 4, 5),  
  C2 = c( 6, 7, 8, 9, 10),  
  C3 = c(11, 12, NA, 14, 15)  
)  
  
df
```

```
# A tibble: 5 x 3  
   C1     C2     C3  
<dbl> <dbl> <dbl>  
1     1     6    11  
2     2     7    12  
3     3     8    NA  
4     4     9    14  
5     5    10    15
```

In the following code we use a function without selecting a name. The function square the values of `df`

```
lapply(df, function(x) sqrt(x))
```

```
$C1  
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
$C2  
[1] 2.449490 2.645751 2.828427 3.000000 3.162278
```

```
$C3  
[1] 3.316625 3.464102      NA 3.741657 3.872983
```

The function read the data and returns the length of each column

```
lapply(df, function(x) length(x))
```

```
$C1  
[1] 5
```

```
$C2  
[1] 5
```

```
$C3  
[1] 5
```

In the next code we have a function that calculate the mean of each column

```
lapply(df, function(x) mean(x, na.rm = TRUE))
```

```
$C1  
[1] 3
```

```
$C2  
[1] 8
```

```
$C3  
[1] 13
```

In the next code we show you we can do both by `map()` function.

```
map(df, sqrt)
```

```
$C1  
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
$C2  
[1] 2.449490 2.645751 2.828427 3.000000 3.162278
```

```
$C3  
[1] 3.316625 3.464102      NA 3.741657 3.872983
```



```
map(df, length)
```

```
$C1  
[1] 5
```

```
$C2  
[1] 5
```

```
$C3  
[1] 5
```

```
map(df, mean, na.rm = TRUE)
```

```
$C1  
[1] 3
```

```
$C2  
[1] 8
```

```
$C3  
[1] 13
```

Question:

Factor and Subsetting

If I have a factor vector and I subset it, the levels are still there even if they are not in the subset. Why does this happen, and how can I avoid it?

In summer 2024, one of my students, Chih-Chen Wang, explained it very well. Here what he wrote;

when you subset a factor vector in R, the underlying levels of the factor remain unchanged even if some of the levels are not present in the subset. This happens because factors are categorical data types with a predefined set of levels. We can remove any levels that are not actually present in the factor by “droplevels()”

Let us create a factor

```
# Create a vector of names
names <- c("John", "Alice", "Bob", "Eve", "Michael")

# Create a factor vector with levels "male" and "female"
f <- factor(c("male", "female", "male", "female", "male"),
            levels = c("male", "female"))

# Assign names to the factor vector
names(f) <- names

# Print the factor vector
print(f)
```

```
      John   Alice    Bob    Eve Michael
      male  female   male  female   male
Levels: male female
```

```
f[-c(1,3,5)] ->
  f1
f1
```

```
      Alice    Eve
      female female
Levels: male female
```

Or we may use the following code to get only females name.

```
f[f=="female"]
```

```
      Alice    Eve
      female female
Levels: male female
```

Now, if you use `droplevels()` function then level of male will be dropped since factor `f1` does not contain any male names.

```
droplevels(f1)
```

```
      Alice    Eve
      female female
Levels: female
```

Question

When to use the `set.seed()` function?

- As you know, the random generation function isn't truly random. It's deterministic based on its input, known as a seed. When we all use the same seed, we get identical results. We often use the seed function to validate our code. If we're debugging, we don't want different outputs each time we run the code. Another reason to use it is to compare our work with others, like team members.

Question:

How to generate a sequence of dates and use them in a for loop?

- There are three main classes for date/time data:
 - `Date` for just the date.
 - `POSIXct` for both the date and the time. “POSIXct” stands for “Portable Operating System Interface Calendar Time”
 - `hms` stands for “hours, minutes, and seconds.”
- `today()` will give you the current date in the `Date` class, `now()` gives you in addition the time.

```
now(tzone = "UTC") # Universal Coordinated Time
```

```
[1] "2024-07-19 02:46:03 UTC"
```

- `Sys.time()` and `Sys.Date()` are from `{base}` package
- current time

```
hms::as_hms(now())
```

```
22:46:03.673952
```

```
class(hms::as_hms(now()))
```

```
[1] "hms"      "difftime"
```

The functions `as_date()`, `as.Date()`, and `ymd()` are all used to work with date data in R, but they come from different packages and have slightly different purposes and behaviors. Here's an overview of each:

1. `as_date()` from `{lubridate}`

- Convert an object to a date or date-time

```
as.Date("2024-07-17")
```

```
[1] "2024-07-17"
```

```
as_date(0)
```

```
[1] "1970-01-01"
```

```
as_date(365)
```

```
[1] "1971-01-01"
```

2. `ymd()` from `{lubridate}`

- **Purpose:** A convenience function to parse dates in the year-month-day format. It automatically recognizes and converts a variety of common date string formats to Date objects.

```
ymd("2024-07-16")
```

```
[1] "2024-07-16"
```

```
ymd("20240716")
```

```
[1] "2024-07-16"
```

3. `as.Date()`: `{base}`

- Convert between character representations and objects of class “Date” representing calendar dates.

```
as.Date("2024-07-17")
```

```
[1] "2024-07-17"
```

```
as.Date("17-07-2024", format = "%d-%m-%Y")
```

```
[1] "2024-07-17"
```

```
as.Date("07-17-24", format = "%m-%d-%y")
```

```
[1] "2024-07-17"
```

We will use `{lubridate}` package

- Only the order of year, month, and day matters

```
ymd(c("2024/07-16", "2024-07/16", "20240716"))
```

```
[1] "2024-07-16" "2024-07-16" "2024-07-16"
```

Note: - Note that `ms()`, `hm()`, and `hms()` won't recognize “-” as a separator because it treats it as negative time. So use `parse_time()` here.

```
ms("10-10")
```

```
[1] "10M -10S"
```

```
ms("10:10")
```

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

— You can order them and it reads only date and time

```
parse_date_time("23, 22, 01 Read only what it needed to read to display the time 07/16/2024", orders = "dHMS")
```

```
[1] "2024-07-16 23:22:01 UTC"
```

- Parsing Dates

```
x <- parse_date("17/07/2024", format = "%d/%m/%Y")
x
```

```
[1] "2024-07-17"
```

```
class(x)
```

```
[1] "Date"
```

```
y <- parse_datetime("07/17/2040 11:59:20", format = "%m/%d/%Y %H:%M:%S")
y
```

```
[1] "2040-07-17 11:59:20 UTC"
```

```
class(y)
```

```
[1] "POSIXct" "POSIXt"
```

```
z <- parse_time("11:59:20", "%H:%M:%S")
z
```

```
11:59:20
```

```
class(z)
```

```
[1] "hms"      "difftime"
```

How to to create dates and date-times?

```
make_date(year = 2024, month = 7, day = 16)
```

```
[1] "2024-07-16"
```

```
make_datetime(year = 2024, month = 8, day = 17, hour = 23, min = 59, sec = 59)
```

```
[1] "2024-08-17 23:59:59 UTC"
```

What happen if we use `as_date()` function to convert a vector of numeral value to date class?

- This function will try to coerce an object to a date.
- `as_datetime()` tries to coerce an object to a `POSIXct` object.

```
year <- c(2000, 2001, 2010)
(as_date(year) ->
 year)
```

```
[1] "1975-06-24" "1975-06-25" "1975-07-04"
```

It creates a a vector of dates in the format “YYYY-MM-DD”. The first entry is year 2000 days after year 1970-01-01

nycflights13 example:

```
library(nycflights13)
flights |>
  select(c(year, month, day, hour, minute)) |>
  glimpse()
```

Rows: 336,776

Columns: 5

```
$ year   <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 201~
$ month  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ day    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ hour   <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6, 6, ~
$ minute <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0, 0, 0, ~
```

Create a column that show s the the date and time of the flights

```

data("flights")
flights %>%
  mutate(datetime = make_datetime(year = year,
                                   month = month,
                                   day = day,
                                   hour = hour,
                                   min = minute)) ->

flights
select(flights, datetime)

```

```

# A tibble: 336,776 x 1
  datetime
  <dtm>
1 2013-01-01 05:15:00
2 2013-01-01 05:29:00
3 2013-01-01 05:40:00
4 2013-01-01 05:45:00
5 2013-01-01 06:00:00
6 2013-01-01 05:58:00
7 2013-01-01 06:00:00
8 2013-01-01 06:00:00
9 2013-01-01 06:00:00
10 2013-01-01 06:00:00
# i 336,766 more rows

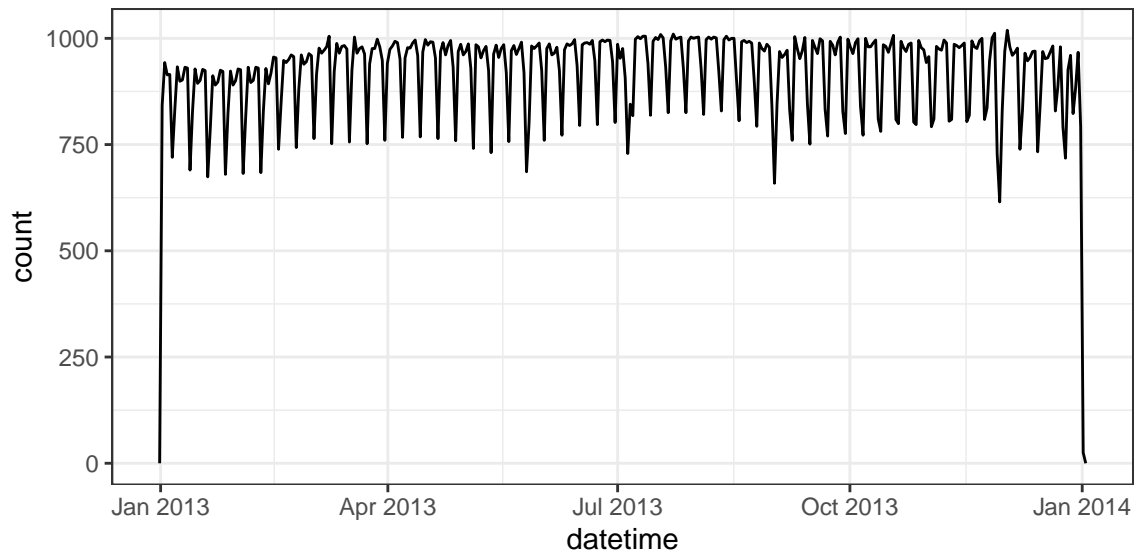
```

- Having it in the date-time format makes it easier to plot.

```

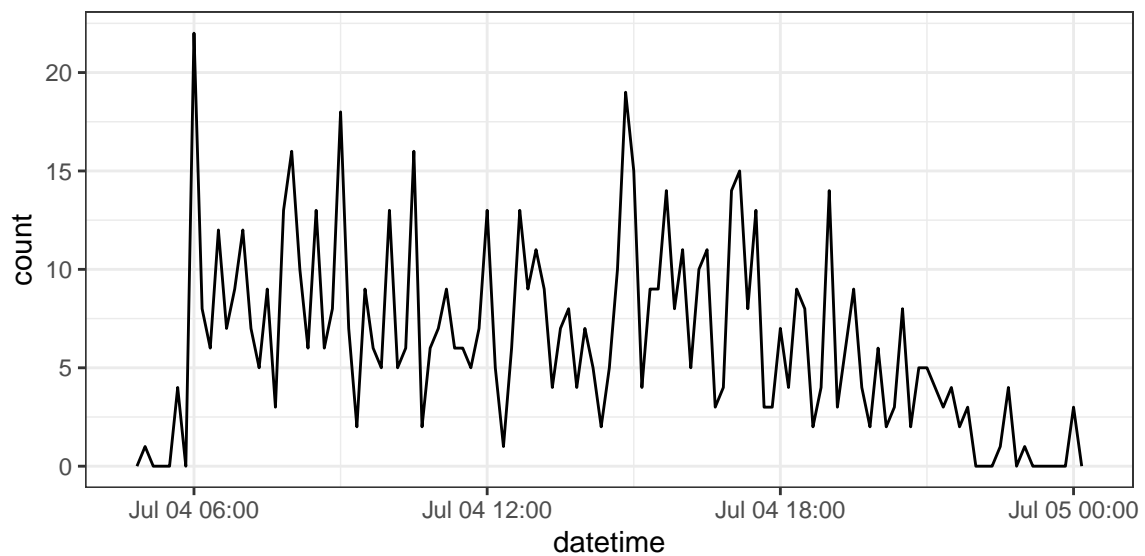
ggplot(flights, aes(x = datetime)) +
  geom_freqpoly(bins = 365)

```

- It makes it easier to filter by date

```
flights %>%
  filter(as_date(datetime) == ymd(20130704)) %>%
  ggplot(aes(x = datetime)) +
  geom_freqpoly(binwidth = 600)
```



Extracting Components

```
ddat <- mdy_hms("07/16/2024 03:51:44")  
ddat
```

```
[1] "2024-07-16 03:51:44 UTC"
```

- `year()` extracts the year.
- `month()` extracts the month.
- `week()` extracts the week.
- `mday()` extracts the day of the month (1, 2, 3, ...).
- `wday()` extracts the day of the week (Saturday, Sunday, Monday ...).
- `yday()` extracts the day of the year (1, 2, 3, ...)
- `hour()` extracts the hour.
- `minute()` extract the minute.
- `second()` extracts the second.

```
year(ddat)
```

```
[1] 2024
```

```
month(ddat, label = TRUE)
```

```
[1] Jul
```

```
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```
week(ddat)
```

```
[1] 29
```

```
mday(ddat)
```

```
[1] 16
```

```
wday(ddat, label = TRUE)
```

```
[1] Tue
```

```
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```
yday(ddat)
```

```
[1] 198
```

```
hour(ddat)
```

```
[1] 3
```

```
minute(ddat)
```

```
[1] 51
```

```
second(ddat)
```

```
[1] 44
```

- Let us generate a sequence of dates starting from today to the end of semester

```
l_day <- ymd("2024-08-17")
t_day <- today()

# Create sequence of dates from today until last day
sequence_day <- seq(t_day, l_day, by = "day")

# Create the dataframe
df <- tibble(
  date = sequence_day,
  days_left = as.numeric(l_day - sequence_day)
)

print(df)
```

```
# A tibble: 31 x 2
  date      days_left
<date>      <dbl>
1 2024-07-18         30
2 2024-07-19         29
3 2024-07-20         28
```

```

4 2024-07-21      27
5 2024-07-22      26
6 2024-07-23      25
7 2024-07-24      24
8 2024-07-25      23
9 2024-07-26      22
10 2024-07-27     21
# i 21 more rows

```

- The following code generates the weekends

```

l_day <- ymd("2024-08-17")
t_day <- today()

# Create sequence of dates from today until last day
sequence_day <- seq(t_day, l_day, by = "day")

weekends <- sequence_day[wday(sequence_day) %in% c(1, 7)] # 1 = Sunday, 7 = Saturday

print(weekends)

```

```

[1] "2024-07-20" "2024-07-21" "2024-07-27" "2024-07-28" "2024-08-03"
[6] "2024-08-04" "2024-08-10" "2024-08-11" "2024-08-17"

```

The output is a dataframe

```

l_day <- ymd("2024-08-17")
t_day <- today()

# Create sequence of dates from today until last day
sequence_day <- seq(t_day, l_day, by = "day")

weekends <- sequence_day[wday(sequence_day) %in% c(1, 7)]
df <- tibble(Weekend = weekends)

print(df)

```

```
# A tibble: 9 x 1
  Weekend
  <date>
1 2024-07-20
2 2024-07-21
3 2024-07-27
4 2024-07-28
5 2024-08-03
6 2024-08-04
7 2024-08-10
8 2024-08-11
9 2024-08-17
```

By For Loop

```
l_day <- ymd("2024-08-17")
t_day <- today()

sequence_day <- seq(t_day, l_day, by = "day")

# Initiate a vector for loop
weekends <- c() # or vector()

# Loop through each date from today to last day
for (dates in sequence_day) {
  d <- as_date(dates)
  if (wday(d) %in% c(1, 7)){
    weekends <- c(weekends, d) # append the date to the weekends vector.
    w <- as_date(weekends)
  }
}

# Print the weekends
print(w)
```

```
[1] "2024-07-20" "2024-07-21" "2024-07-27" "2024-07-28" "2024-08-03"
[6] "2024-08-04" "2024-08-10" "2024-08-11" "2024-08-17"
```

```
l_day <- ymd("2024-08-17")
t_day <- today()
```

```

sequence_day <- seq(t_day, l_day, by = "day")

weekends_df <- tibble(weekend = as_date(character()))

# Loop through each date from today to last day
for (dates in sequence_day) {
  d <- as_date(dates)
  if (wday(d) %in% c(1, 7)){
    weekends_df <- rbind(weekends_df, tibble(weekend = d)) # append the date to the dataframe
  }
}

# Print the weekends data frame
print(weekends_df)

```

```

# A tibble: 9 x 1
  weekend
  <date>
1 2024-07-20
2 2024-07-21
3 2024-07-27
4 2024-07-28
5 2024-08-03
6 2024-08-04
7 2024-08-10
8 2024-08-11
9 2024-08-17

```

Question

Mixed data Types

In R, a vector must have elements of the same type, so if we try to create a vector with mixed types, R will **coerce** them.

- Consider following vectors. These are mixed vectors and R coerced them to the most flexible type.

```
v1 <- c(1, 3.14, "a", TRUE)
v1
```

```
[1] "1"      "3.14" "a"      "TRUE"
```

```
typeof(v1)
```

```
[1] "character"
```

```
map(v1, typeof)
```

```
[[1]]
[1] "character"
```

```
[[2]]
[1] "character"
```

```
[[3]]
[1] "character"
```

```
[[4]]
[1] "character"
```

```
v2 <- c(FALSE, exp(1), -21, pi, TRUE)
v2
```

```
[1] 0.000000 2.718282 -21.000000 3.141593 1.000000
```

```
typeof(v2)
```

```
[1] "double"
```

```
map(v2, typeof)
```

```
[[1]]
[1] "double"
```

```
[[2]]
```

```
[1] "double"
```

```
[[3]]
```

```
[1] "double"
```

```
[[4]]
```

```
[1] "double"
```

```
[[5]]
```

```
[1] "double"
```

We can use a list to store elements of mixed types without coercion.

```
l1 <-list(1, 3.14, "a", TRUE)
```

```
typeof(l1)
```

```
[1] "list"
```

```
map(l1, typeof)
```

```
[[1]]
```

```
[1] "double"
```

```
[[2]]
```

```
[1] "double"
```

```
[[3]]
```

```
[1] "character"
```

```
[[4]]
```

```
[1] "logical"
```

****We can use `sapply()` from `{base}` package to find type of objects**

```
sapply(l1, typeof)
```

```
[1] "double"    "double"    "character" "logical"
```



```
sapply(v2, typeof)
```

```
[1] "double" "double" "double" "double" "double"
```

```
sapply(v1, typeof)
```

```
      1      3.14      a      TRUE  
"character" "character" "character" "character"
```

DataFrame

Having a column with mixed data types (like column 3 in the example) indicates that the data is not *tidy*. In *tidy* data, each column should contain only one type of data (e.g., all numbers, all characters, etc.).

```
df <- tibble::tibble(  
  Var1 = c(T, TRUE, F),  
  Var2 = c("Jim", "Steve", "Mary"),  
  Var3 = c(4.5, FALSE, -pi) # This column has mixed data types  
)  
df
```

```
# A tibble: 3 x 3  
  Var1 Var2 Var3  
  <lgl> <chr> <dbl>  
1 TRUE  Jim    4.5  
2 TRUE  Steve  0  
3 FALSE Mary  -3.14
```

When a column contains mixed types, R often coerces the entire column to the most flexible type (in this case, `dbl`) to accommodate all values.

```
typeof(df)
```

```
[1] "list"
```

```
class(df)
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

```
map(df, typeof)
```

```
$Var1  
[1] "logical"
```

```
$Var2  
[1] "character"
```

```
$Var3  
[1] "double"
```

```
sapply(df, typeof)
```

```
      Var1      Var2      Var3  
"logical" "character" "double"
```

Question

Is the “Special For Loop Method” a good method to be adapted as a function to determine values like median or standard deviation for any given dataframe?

- We can find the numerical summaries of a data set as follow

```
mpg |>  
  summarise(median(cty), mean(cty), sd(cty), sum(cty))
```

```
# A tibble: 1 x 4  
  `median(cty)` `mean(cty)` `sd(cty)` `sum(cty)`  
    <dbl>      <dbl>    <dbl>    <int>  
1      17      16.9     4.26    3945
```

- Or we can get the numerical summaries based on `class` of the cars

```
mpg |>  
  group_by(class) |>  
  summarise(median(cty), mean(cty), sd(cty), sum(cty), n())
```

```
# A tibble: 7 x 6
  class      `median(cty)` `mean(cty)` `sd(cty)` `sum(cty)` `n()``
  <chr>          <dbl>      <dbl>      <dbl>      <int> <int>
1 2seater          15        15.4        0.548         77     5
2 compact          20        20.1         3.39        946    47
3 midsize          18        18.8         1.95        769    41
4 minivan          16        15.8         1.83        174    11
5 pickup           13         13         2.05        429    33
6 subcompact       19        20.4         4.60        713    35
7 suv              13        13.5         2.42        837    62
```

- The summaries of all variables can be obtained by

```
mpg |>
  summary()
```

```
manufacturer      model      displ      year
Length:234      Length:234      Min.   :1.600      Min.   :1999
Class :character Class :character 1st Qu.:2.400      1st Qu.:1999
Mode  :character Mode  :character Median :3.300      Median :2004
                                Mean  :3.472      Mean  :2004
                                3rd Qu.:4.600      3rd Qu.:2008
                                Max.   :7.000      Max.   :2008

      cyl      trans      drv      cty
Min.   :4.000      Length:234      Length:234      Min.   : 9.00
1st Qu.:4.000      Class :character      Class :character 1st Qu.:14.00
Median :6.000      Mode  :character      Mode  :character Median :17.00
Mean   :5.889                                Mean   :16.86
3rd Qu.:8.000                                3rd Qu.:19.00
Max.   :8.000                                Max.   :35.00

      hwy      fl      class
Min.   :12.00      Length:234      Length:234
1st Qu.:18.00      Class :character      Class :character
Median :24.00      Mode  :character      Mode  :character
Mean   :23.44
3rd Qu.:27.00
Max.   :44.00
```

Create by For Loop

```
mean_vec <- vector(mode = "numeric", length = length(mpg))
for (i in seq_along(mpg)) {
  mean_vec[[i]] <- mean(mpg[[i]], na.rm = TRUE)
}
```

Warning in mean.default(mpg[[i]], na.rm = TRUE): argument is not numeric or logical: returning NA
 Warning in mean.default(mpg[[i]], na.rm = TRUE): argument is not numeric or logical: returning NA
 Warning in mean.default(mpg[[i]], na.rm = TRUE): argument is not numeric or logical: returning NA
 Warning in mean.default(mpg[[i]], na.rm = TRUE): argument is not numeric or logical: returning NA
 Warning in mean.default(mpg[[i]], na.rm = TRUE): argument is not numeric or logical: returning NA
 Warning in mean.default(mpg[[i]], na.rm = TRUE): argument is not numeric or logical: returning NA

```
mean_vec
```

```
[1]      NA      NA    3.471795 2003.500000    5.888889      NA
[7]      NA 16.858974 23.440171      NA      NA
```

For non_numerical variables you get NA.

- Let select all numerical variables of mpg data.

```
mpg |>
  select(c(displ, year, cyl, cty, hwy)) ->
  mpg_num
mpg_num
```

```
# A tibble: 234 x 5
  displ  year  cyl  cty  hwy
  <dbl> <int> <int> <int> <int>
1   1.8  1999     4    18    29
2   1.8  1999     4    21    29
3     2  2008     4    20    31
4     2  2008     4    21    30
5   2.8  1999     6    16    26
```

```

6   2.8  1999    6   18   26
7   3.1  2008    6   18   27
8   1.8  1999    4   18   26
9   1.8  1999    4   16   25
10  2    2008    4   20   28
# i 224 more rows

```

```

mean_vec <- vector(mode = "numeric", length = length(mpg_num)) # Initiate a vector for loop
for (i in seq_along(mpg_num)) {
  mean_vec[[i]] <- mean(mpg_num[[i]], na.rm = TRUE)
}
mean_vec

```

```
[1] 3.471795 2003.500000 5.888889 16.858974 23.440171
```

Let's find the standard deviation of each column

```

sd_vec <- vector()
for (i in seq_along(mpg_num)) {
  sd_vec[[i]] <- sd(mpg_num[[i]], na.rm = TRUE)
}
sd_vec

```

```
[1] 1.291959 4.509646 1.611534 4.255946 5.954643
```

Create by functions from {base} package

- *We can use `colMeans()` function from {base} package.

```
colMeans(mpg_num)
```

```

      displ      year      cyl      cty      hwy
3.471795 2003.500000 5.888889 16.858974 23.440171

```

There are other functions `rowMeans()`, `colSums()` and `rowSums()`. Unfortunately, there is no “colSDs()” function

```
colSums(mpg_num)
```

displ	year	cyl	cty	hwy
812.4	468819.0	1378.0	3945.0	5485.0

There are 234 rows. Thus there are 234. We use `head()` to show only the first 3 outputs

```
rowSums(mpg_num)
```

```
[1] 2051.8 2054.8 2065.0 2065.0 2049.8 2051.8 2062.1 2048.8 2045.8 2062.0
[11] 2060.0 2047.8 2049.8 2059.1 2057.1 2046.8 2059.1 2059.2 2055.3 2047.3
[21] 2055.3 2042.7 2051.0 2054.7 2050.7 2064.2 2062.2 2062.0 2054.3 2046.3
[31] 2038.7 2044.5 2051.4 2066.4 2052.1 2064.5 2060.6 2047.4 2049.0 2046.3
[41] 2046.3 2058.3 2058.3 2045.3 2045.8 2044.8 2056.8 2057.0 2051.7 2049.7
[51] 2038.9 2039.9 2053.7 2053.7 2041.7 2040.2 2038.2 2038.9 2050.7 2041.7
[61] 2050.7 2039.2 2052.7 2038.9 2048.7 2041.7 2050.7 2050.7 2048.7 2041.7
[71] 2038.2 2039.2 2051.7 2038.9 2039.6 2040.4 2051.4 2040.0 2043.0 2040.0
[81] 2050.0 2052.6 2042.0 2040.2 2040.2 2040.6 2040.6 2050.6 2038.4 2051.4
[91] 2052.8 2051.8 2061.0 2058.0 2047.6 2048.6 2058.6 2057.6 2055.4 2065.6
[101] 2060.6 2061.6 2056.6 2060.6 2073.8 2074.8 2073.8 2064.0 2049.4 2050.4
[111] 2065.4 2066.4 2051.5 2051.5 2064.3 2050.0 2053.0 2062.0 2061.0 2057.7
[121] 2056.7 2057.7 2056.0 2051.7 2044.0 2042.7 2041.7 2053.7 2052.7 2047.1
[131] 2037.0 2050.2 2050.4 2037.6 2040.4 2039.4 2051.4 2040.0 2050.0 2052.6
[141] 2042.0 2055.4 2051.4 2068.5 2069.5 2063.5 2062.5 2052.0 2052.0 2061.5
[151] 2039.3 2040.3 2052.0 2051.6 2052.1 2050.8 2052.8 2063.8 2062.3 2048.5
[161] 2047.5 2061.5 2058.5 2060.5 2055.5 2052.2 2050.2 2050.5 2050.5 2059.5
[171] 2061.5 2058.5 2061.5 2040.7 2041.7 2042.4 2040.4 2054.0 2051.7 2055.2
[181] 2053.2 2066.4 2066.4 2052.0 2052.0 2064.5 2053.2 2055.2 2066.4 2067.4
[191] 2052.0 2052.0 2062.3 2058.8 2061.8 2065.8 2078.8 2074.8 2037.7 2052.7
[201] 2040.7 2041.7 2053.7 2040.4 2042.4 2051.0 2054.0 2055.0 2050.0 2064.0
[211] 2065.0 2048.8 2081.9 2055.0 2050.0 2065.0 2064.0 2065.5 2065.5 2046.8
[221] 2048.8 2083.9 2074.9 2055.0 2050.0 2063.5 2064.5 2054.8 2051.8 2061.0
[231] 2064.0 2049.8 2051.8 2060.6
```

```
rowMeans(mpg_num) |>
  head(n = 3)
```

```
[1] 410.36 410.96 413.00
```

Create by `map()` from `{purrr}` package

`map_*()` takes a vector (or list or data frame) as input, applies a provided function on each element of that vector, and outputs a vector of the same length.

- ``map()`` returns a list.
- ``map_lgl()`` returns a logical vector.
- ``map_int()`` returns an integer vector.
- ``map_dbl()`` returns a double vector.
- ``map_chr()`` returns a character vector.

```
map_dbl(mpg_num, mean)
```

```
map(mpg, summary)
```

```
$manufacturer
```

```
  Length      Class      Mode
    234 character character
```

```
$model
```

```
  Length      Class      Mode
    234 character character
```

```
$displ
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.  Max.
 1.600  2.400   3.300   3.472  4.600   7.000
```

```
$year
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.  Max.
 1999   1999   2004   2004   2008   2008
```

```
$cyl
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.  Max.
 4.000  4.000   6.000   5.889  8.000   8.000
```

```
$trans
```

```
  Length      Class      Mode
    234 character character
```

```
$drv
```

```
  Length      Class      Mode
    234 character character
```

```
$cty
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.  Max.
 9.00  14.00   17.00   16.86  19.00  35.00
```

```
$hwy
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
12.00  18.00   24.00   23.44  27.00   44.00
```

```
$fl
  Length      Class      Mode
    234 character character
```

```
$class
  Length      Class      Mode
    234 character character
```

```
map_chr(mpg, typeof)
```

```
manufacturer      model      displ      year      cyl      trans
"character" "character" "double" "integer" "integer" "character"
      drv      cty      hwy      fl      class
"character" "integer" "integer" "character" "character"
```

```
map_chr(mpg, class)
```

```
manufacturer      model      displ      year      cyl      trans
"character" "character" "numeric" "integer" "integer" "character"
      drv      cty      hwy      fl      class
"character" "integer" "integer" "character" "character"
```

Find the number of unique value in each column

```
map_int(mpg, function(x) length(unique(x)))
```

```
manufacturer      model      displ      year      cyl      trans
      15      38      35      2      4      10
      drv      cty      hwy      fl      class
      3      21      27      5      7
```

Generate 7 random normals for each of $\mu = -10, 0, 10, \dots, 100$.

1. Create by loop


```
set.seed(123)
df <- tibble(row=1:7)
for (i in seq(-10, 100, by =10)){
  df <- cbind(df, tibble(rnorm(mean = i, n=7)))
}
df
```

row	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)
1	1	-10.560476	-1.2650612
2	2	-10.230177	-0.6868529
3	3	-8.441292	-0.4456620
4	4	-9.929492	1.2240818
5	5	-9.870712	0.3598138
6	6	-8.284935	0.4007715
7	7	-9.539084	0.1106827
	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)
1		19.78203	28.86186
2		18.97400	31.25381
3		19.27111	30.42646
4		19.37496	29.70493
5		18.31331	30.89513
6		20.83779	30.87813
7		20.15337	30.82158
	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)
1		48.73460	59.91663
2		52.16896	60.25332
3		51.20796	59.97145
4		48.87689	59.95713
5		49.59712	61.36860
6		49.53334	59.77423
7		50.77997	61.51647
	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)	rnorm(mean = i, n = 7)
1		78.98142	89.50897
2		78.92821	87.69083
3		80.30353	91.00574
4		80.44821	89.29080
5		80.05300	89.31199
6		80.92227	91.02557
7		82.05008	89.71523

- The above has three issues
 - I. We used `cbind()` from `base` package. we want to use `bind_cols()` from `dplyr`

- II. We do not need the first row. We need to remove it.
- III. The column names are not very good. So we can give it a better name. I use `names(df)` function

```
set.seed(123)
df <- tibble(row=1:7)
for (i in seq(-10, 100, by =10)){
  df <- bind_cols(df, tibble(rnorm(mean = i, n=7)))
}
```

New names:

New names:

New names:

New names:

New names:

New names:

New names:

New names:

New names:

New names:

New names:

* `rnorm(mean = i, n = 7)` -> `rnorm(mean = i, n = 7)...2`

* `rnorm(mean = i, n = 7)` -> `rnorm(mean = i, n = 7)...3`

```
df |>
  select(-1) ->
  df_new
df_new
```

A tibble: 7 x 12

```
`rnorm(mean = i, n = 7)...2` rnorm(mean = i, n = 7)...1 rnorm(mean = i, n = ~2
      <dbl>                                <dbl>                                <dbl>
1      -10.6                                -1.27                                9.44
2      -10.2                                -0.687                               11.8
3       -8.44                                -0.446                               10.5
4       -9.93                                 1.22                                8.03
5       -9.87                                 0.360                               10.7
6       -8.28                                 0.401                                9.53
7       -9.54                                 0.111                                8.93
# i abbreviated names: 1: `rnorm(mean = i, n = 7)...3`,
# 2: `rnorm(mean = i, n = 7)...4`
```

```
# i 9 more variables: `rnorm(mean = i, n = 7)...5` <dbl>,
#   `rnorm(mean = i, n = 7)...6` <dbl>, `rnorm(mean = i, n = 7)...7` <dbl>,
#   `rnorm(mean = i, n = 7)...8` <dbl>, `rnorm(mean = i, n = 7)...9` <dbl>,
#   `rnorm(mean = i, n = 7)...10` <dbl>, `rnorm(mean = i, n = 7)...11` <dbl>,
#   `rnorm(mean = i, n = 7)...12` <dbl>, ...
```

```
names(df_new) <- LETTERS[1:12]
df_new
```

```
# A tibble: 7 x 12
      A      B      C      D      E      F      G      H      I      J      K      L
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 -10.6 -1.27  9.44  19.8  28.9  40.7  48.7  59.9  68.5  79.0  89.5  98.8
2 -10.2 -0.687 11.8  19.0  31.3  40.6  52.2  60.3  70.6  78.9  87.7 100.
3 -8.44 -0.446 10.5  19.3  30.4  39.9  51.2  60.0  70.1  80.3  91.0  99.9
4 -9.93  1.22  8.03  19.4  29.7  39.7  48.9  60.0  70.2  80.4  89.3 100.
5 -9.87  0.360 10.7  18.3  30.9  39.6  49.6  61.4  70.4  80.1  89.3 100.
6 -8.28  0.401  9.53  20.8  30.9  39.3  49.5  59.8  69.5  80.9  91.0  99.6
7 -9.54  0.111  8.93  20.2  30.8  39.8  50.8  61.5  69.7  82.1  89.7 101.
```

2. CReate by map_

```
set.seed(123)
map_dfc(seq(-10, 100, by = 10), rnorm, n = 7)
```

New names:

```
* `` -> `...1`
* `` -> `...2`
* `` -> `...3`
* `` -> `...4`
* `` -> `...5`
* `` -> `...6`
* `` -> `...7`
* `` -> `...8`
* `` -> `...9`
* `` -> `...10`
* `` -> `...11`
* `` -> `...12`
```

```
# A tibble: 7 x 12
  ...1 ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...11 ...12
```

	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	-10.6	-1.27	9.44	19.8	28.9	40.7	48.7	59.9	68.5	79.0	89.5	98.8
2	-10.2	-0.687	11.8	19.0	31.3	40.6	52.2	60.3	70.6	78.9	87.7	100.
3	-8.44	-0.446	10.5	19.3	30.4	39.9	51.2	60.0	70.1	80.3	91.0	99.9
4	-9.93	1.22	8.03	19.4	29.7	39.7	48.9	60.0	70.2	80.4	89.3	100.
5	-9.87	0.360	10.7	18.3	30.9	39.6	49.6	61.4	70.4	80.1	89.3	100.
6	-8.28	0.401	9.53	20.8	30.9	39.3	49.5	59.8	69.5	80.9	91.0	99.6
7	-9.54	0.111	8.93	20.2	30.8	39.8	50.8	61.5	69.7	82.1	89.7	101.

Table

- To create a frequency table of categorical variables, such as class, and its graph, follow these steps:

```
table(mpg$class) ->
t1

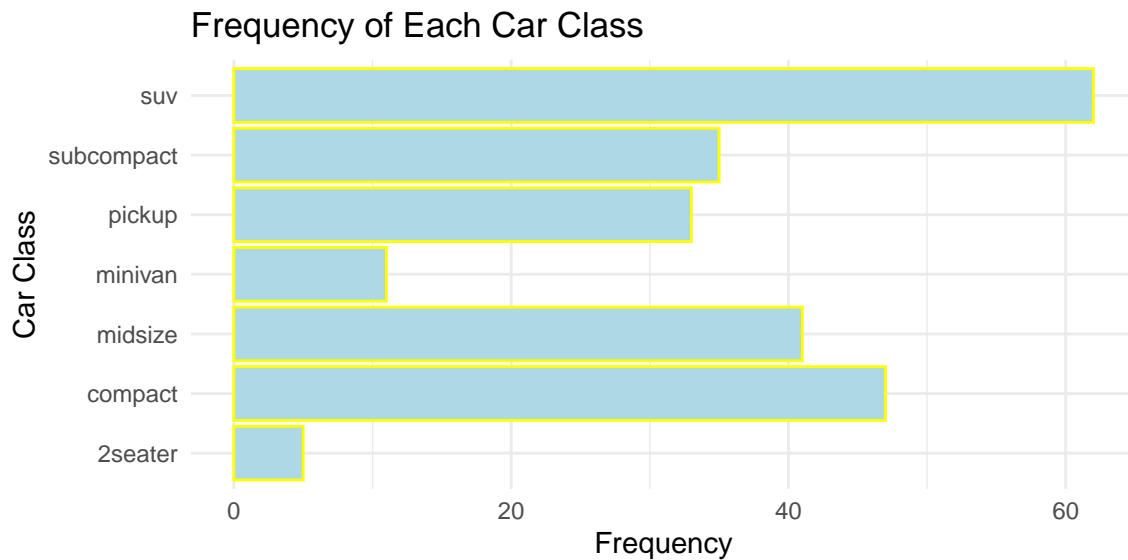
as.data.frame(t1) ->
df

colnames(df) <- c("Class", "Count")

df
```

	Class	Count
1	2seater	5
2	compact	47
3	midsize	41
4	minivan	11
5	pickup	33
6	subcompact	35
7	suv	62

```
df |>
ggplot(aes( y =Class, x= Count)) +
geom_bar(stat = "identity", color = "yellow", fill = "lightblue") +
theme_minimal() +
labs(title = "Frequency of Each Car Class",
      y = "Car Class",
      x = "Frequency")
```

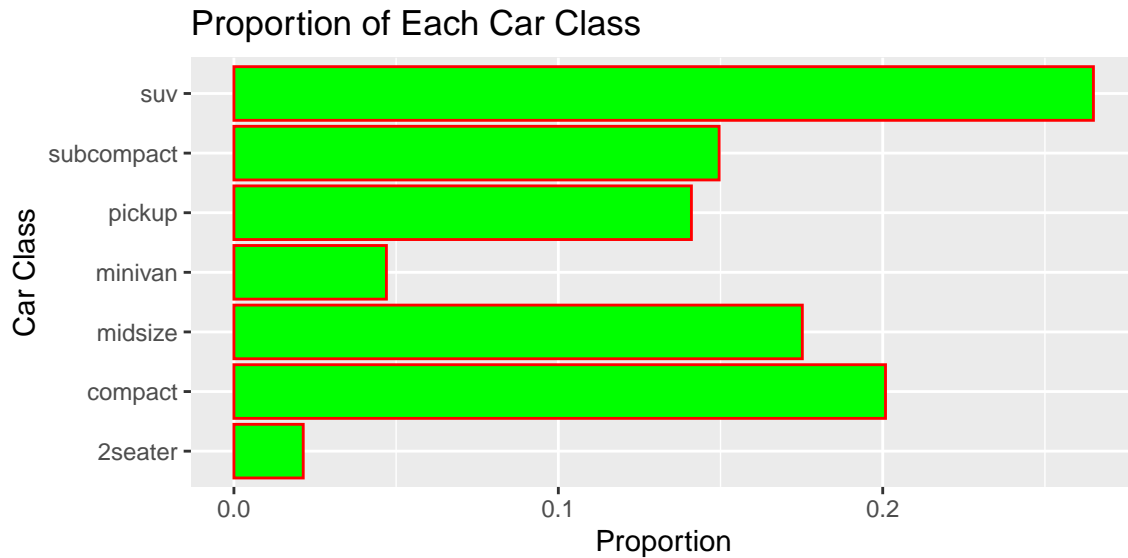


- Now calculate proportion of each class and draw its bargraph

```
df |>
  mutate(Prop = Count/sum(Count)) ->
  df_p
df_p
```

	Class	Count	Prop
1	2seater	5	0.02136752
2	compact	47	0.20085470
3	midsize	41	0.17521368
4	minivan	11	0.04700855
5	pickup	33	0.14102564
6	subcompact	35	0.14957265
7	suv	62	0.26495726

```
df_p |>
  ggplot(aes( y =Class, x= Prop)) +
  geom_bar(stat = "identity", color = "red", fill = "green") +
  theme_grey() +
  labs(title = "Proportion of Each Car Class",
       y = "Car Class",
       x = "Proportion")
```



Question

vectorization in R to duplicate, mutate (Create a new column)

- Go to https://biodash.github.io/codeclub/12_loops/ for “Vectorization and loops in R”

Consider the following data

```
table(mpg$class) ->
t1

as.data.frame(t1) ->
df

colnames(df) <- c("Class", "Count")

df
```

	Class	Count
1	2seater	5
2	compact	47
3	midsize	41
4	minivan	11
5	pickup	33

```
6 subcompact    35
7          suv    62
```

We want to create a new column called Proportion with `mutate()` and without mutate (vectorization)

mutate() function from dplyr package

```
df |>
  mutate(Proportion = Count/sum(Count)) ->
  df_prop
df_prop
```

	Class	Count	Proportion
1	2seater	5	0.02136752
2	compact	47	0.20085470
3	midsize	41	0.17521368
4	minivan	11	0.04700855
5	pickup	33	0.14102564
6	subcompact	35	0.14957265
7	suv	62	0.26495726

Create by \$

```
df$Proportion <- (df$Count/sum(df$Count))
df
```

	Class	Count	Proportion
1	2seater	5	0.02136752
2	compact	47	0.20085470
3	midsize	41	0.17521368
4	minivan	11	0.04700855
5	pickup	33	0.14102564
6	subcompact	35	0.14957265
7	suv	62	0.26495726

Create by Subsetting

```
df["Percentage"] <- df["Proportion"]*100
df
```

	Class	Count	Proportion	Percentage
1	2seater	5	0.02136752	2.136752
2	compact	47	0.20085470	20.085470
3	midsize	41	0.17521368	17.521368
4	minivan	11	0.04700855	4.700855
5	pickup	33	0.14102564	14.102564
6	subcompact	35	0.14957265	14.957265
7	suv	62	0.26495726	26.495726