MODUL 7
TIDYVERSE DAN IMPORTING DATA
(TIBBLE, DOT OPERATOR, PURRR, TIDYVERSE CONDITIONALS)

A. Tujuan Praktikum
- Memahami tipe data *tibble*
- Memahami penggunaan *dot operator* dalam proses manipulasi data
- Menggunakan paket `purrr` dalam proses manipulasi data
- Memahami penggunaan *tidyverse conditionals* dalam proses manipulasi data
- Memahami langkah-langkah dan fungsi yang digunakan untuk *importing data*

B. Alokasi Waktu
1 x pertemuan = 120 menit

C. Dasar Teori
Data yang *tidy* harus disimpan dalam tipe *data frame*. Tipe *data frame* telah dibahas pada modul sebelumnya dan sebagai contoh, kita juga telah menggunakan *dataset* contoh "*murders*" yang bertipe *data frame* di beberapa bagian modul sebelumnya. Pada modul 6, kita telah membahas fungsi `group_by`, yang dapat digunakan untuk mengelompokkan data sebelum melakukan penghitungan-penghitungan statistik. Namun, pada bagian modul sebelumnya, kita belum membahas lebih lanjut di mana informasi mengenai grup data tersebut disimpan dalam *data frame*?

```
murders %>% group_by(region)
#> # A tibble: 51 x 6
#> # Groups: region [4]
#> state abb region population total rate
#> <chr> <chr> <fct> <dbl> <dbl> <dbl>
#> 1 Alabama AL South 4779736 135 2.82
#> 2 Alaska AK West 710231 19 2.68
#> 3 Arizona AZ West 6392017 232 3.63
#> 4 Arkansas AR South 2915918 93 3.19
#> 5 California CA West 37253956 1257 3.37
#> # ... with 46 more rows
```

TIBBLE
Berdasarkan *output* pengelompokan data menggunakan fungsi `group_by` diatas, dapat dilihat bahwa pada baris pertama, terdapat informasi *tibble* diikuti dengan besar dimensinya. Sebagai alternatif, kita juga dapat mengetahui *class* atau tipe data dari objek yang dihasilkan menggunakan:

```
murders %>% group_by(region) %>% class()
#> [1] "grouped_df" "tbl_df" "tbl" "data.frame"
```

Tbl atau *tibble*, adalah salah satu jenis *data frame* khusus. Biasanya, hasil implementasi fungsi `group_by` dan `summarize` akan selalu ditampilkan dalam tipe data ini. Namun, fungsi `group_by` memiliki jenis tbl khusus, yaitu: *grouped_df*. Manipulasi data dengan `dplyr` menggunakan fungsi `select`, `filter`, `mutate`, dan `arrange` akan menghasilkan *output* dengan tipe data yang

konsisten atau sama dengan *input*nya: jika *input* berupa *data frame* biasa, maka output manipulasi data juga akan berbentuk *data frame*, sedangkan jika *input* yang diterima berupa *tibble*, maka tipe data yang dihasilkan pada *output*nya juga akan sama dengan *input*nya.

Tibble sangat mirip dengan *data frame*. Namun, terdapat beberapa perbedaan penting antara keduanya yang akan kita bahas selanjutnya. Pertama, hasil yang ditampilkan oleh *tibble* lebih mudah dibaca daripada *data frame*. Silahkan coba bandingkan tampilan *dataset* "*murders*" dengan perintah `murders` dan coba ubah *dataset* tersebut dalam bentuk *tibble* menggunakan `as_tibble(murders)`.

Perbedaan kedua antara *data frame* dan *tibble* akan kita jelaskan melalui contoh implementasi kasus berikut. Setelah melakukan pengelompokan data dengan tipe *data frame*, jika kita melakukan *subsetting* terhadap *data frame* menggunakan *script* dibawah ini, maka satu kolom hasil *subsetting* tersebut akan dianggap memiliki tipe data selain *data frame*:

```
class(murders[,4])
#> [1] "numeric"
```

Berbeda dengan *tibble* hasil *subsetting* tetap akan dianggap memiliki tipe *data frame* seperti contoh hasil dibawah ini:

```
class(as_tibble(murders)[,4])
#> [1] "tbl_df" "tbl" "data.frame"
```

Penggunaan tipe data *tibble* akan sangat berguna jika kita menggunakan `tidyverse`. Hal ini dikarenakan fungsi-fungsi yang dimiliki oleh paket tersebut membutuhkan *input* yang tipenya adalah *data frame*. Sebaliknya, jika tipe data yang dimiliki adalah *tibble*, namun kita membutuhkan *subsetting* data yang hasilnya berupa vektor, dapat digunakan operator aksesor ($) seperti contoh berikut:

```
class(as_tibble(murders)$population)
#> [1] "numeric"
```

Perbedaan ketiga terdapat pada proses *error handling* yang dihasilkan oleh *data frame* dan *tibble*. Misalnya, kita secara tidak sengaja menuliskan nama variabel yang salah atau tidak terdapat pada *data frame*, yaitu: *Population* (variabel yang benar adalah *population*). Hasil yang ditampilkan pada tipe *data frame* adalah NULL tanpa peringatan, yang pada akhirnya dapat mempersulit proses *debug*.

```
murders$Population
#> NULL
```

Sebaliknya, jika kita mencoba langkah yang sama pada tipe data *tibble*, maka selain tampilan NULL, maka akan ditampilkan pula peringatan informatif:

```
as_tibble(murders)$Population
#> Warning: Unknown or uninitialised column: 'Population'.
#> NULL
```

Perbedaan selanjutnya: pada tipe *data frame*, vektor harus berisi numerik, karakter, atau logika, sedangkan *tibble* dapat terdiri dari objek yang lebih kompleks, seperti *list* atau *function*. Pada *script* selanjutnya, kita akan mencoba membuat *tibble* yang terdiri dari beberapa fungsi statistik:

```
tibble(id = c(1, 2, 3), func = c(mean, median, sd))
#> # A tibble: 3 x 2
#> id func
#> <dbl> <list>
#> 1 1 <fn>
#> 2 2 <fn>
#> 3 3 <fn>
```

Untuk membuat *data frame* baru dalam format *tibble*, dapat digunakan fungsi `tibble` seperti contoh berikut.

```
grades <- tibble(names = c("John", "Juan", "Jean", "Yao"),
exam_1 = c(95, 80, 90, 85),
exam_2 = c(90, 85, 85, 90))
```

R juga memiliki fungsi `data.frame`, yang dapat digunakan untuk membuat *data frame*. Perbedaan penting lainnya antara *data frame* dan *tibble* adalah: secara *default*, *data.frame* baru akan menjadikan data karakter sebagai faktor tanpa memberikan peringatan atau pesan khusus:

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
exam_1 = c(95, 80, 90, 85),
exam_2 = c(90, 85, 85, 90))
class(grades$names)
#> [1] "factor"
```

Untuk mengatasi permasalahan tersebut, dapat ditambahkan argumen `stringAsFactors` sehingga data karakter akan tetap terdeteksi memiliki kelas karakter (bukan faktor):

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
exam_1 = c(95, 80, 90, 85),
exam_2 = c(90, 85, 85, 90),
stringsAsFactors = FALSE)
class(grades$names)
#> [1] "character"
```

Konversi *data frame* menjadi *tibble*, dapat dilakukan dengan menggunakan fungsi `as_tibble`.

```
as_tibble(grades) %>% class()
#> [1] "tbl_df" "tbl" "data.frame"
```

DOT OPERATOR

Salah satu keuntungan menggunakan operator *pipe* (%>%) adalah: kita tidak perlu mendefinisikan nama objek baru tiap proses memanipulasi *data frame*. Sebagai contoh, dua *script* selanjutnya akan mencoba menghitung tingkat pembunuhan rata-rata ("*rate*") pada `region == "South"`. Tanpa menggunakan operator *pipe*, pada tiap baris *script*, kita akan diminta mendefinisikan nama objek baru untuk menyimpan hasil manipulasi data yang kita lakukan:

```
tab_1 <- filter(murders, region == "South")
tab_2 <- mutate(tab_1, rate = total / population * 10^5)
rates <- tab_2$rate
median(rates)
#> [1] 3.4
```

Dengan menggunakan operator *pipe*, hasil manipulasi data yang dilakukan akan memiliki nilai yang sama, namun proses pendefinisian nama objek baru telah digantikan dengan operator `%>%`:

```
filter(murders, region == "South") %>%
mutate(rate = total / population * 10^5) %>%
summarize(median = median(rate)) %>%
pull(median)
#> [1] 3.4
```

Pada contoh selanjutnya: misalnya, untuk keperluan analisis sementara, kita ingin mengakses hasil dari proses manipulasi data pada baris kedua. Jika kita telah mendefinisikan objek baru untuk menyimpan data hasil manipulasi seperti pada *script* pertama, dengan mudah kita dapat mengevaluasi hasil dengan memanggil nama variabel yang digunakan. Namun, jika kita menggunakan operator *pipe* seperti pada contoh *script* kedua, pengaksesan hasil dari baris manipulasi kedua dapat dilakukan dengan *dot operator*:

```
rates <-filter(murders, region == "South") %>%
mutate(rate = total / population * 10^5) %>%
.$rate
median(rates)
#> [1] 3.4
```

PURR PACKAGE

Dalam modul 5, kita telah mempelajari tentang fungsi `sapply`, yang dapat digunakan untuk menerapkan fungsi yang sama untuk setiap elemen vektor (bersifat *element-wise*). Misalnya, kita akan membuat sebuah fungsi untuk menghitung jumlah bilangan bulat '*n*' pertama untuk beberapa nilai '*n*' menggunakan `sapply` :

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

Paket `purrr` yang akan kita bahas pada bagian ini, juga memiliki fungsi yang mirip dengan `sapply` tetapi memiliki keterkaitan yang lebih baik terhadap fungsi `tidyverse` lainnya. Keuntungan utamanya adalah kita dapat lebih mengontrol tipe data yang dihasilkan oleh paket `purrr`. Sebaliknya, `sapply` dapat menghasilkan beberapa tipe data objek yang berbeda; misalnya, jika kita mengharapkan hasil komputasi berupa numerik, dalam suatu kondisi tertentu `sapply` justru mengubah hasil komputasi menjadi bertipe karakter. Dengan menggunakan `purrr`, objek yang dihasilkan akan memiliki tipe sesuai dengan yang telah ditentukan atau menampilkan kesalahan jika hasil yang diinginkan tidak memungkinkan.

Fungsi `purrr` pertama yang akan kita pelajari adalah `map`, yang berfungsi sangat mirip dengan `sapply` tetapi selalu menghasilkan *output* dalam tipe data *list*:

```
library(purrr)
s_n <- map(n, compute_s_n)
class(s_n)
#> [1] "list"
```

Jika output yang kita inginkan berupa vektor numerik, kita bisa menggunakan `map_dbl` yang akan selalu memberikan *output* dengan tipe vektor numerik.

```
s_n <- map_dbl(n, compute_s_n)
class(s_n)
#> [1] "numeric"
```

Fungsi `purrr` yang sangat berguna untuk berinteraksi dengan `tidyverse` adalah `map_df`, yang akan selalu menghasilkan komputasi dalam bentuk *tibble data frame*. Untuk menghasilkan tipe data tersebut, fungsi `map_df` membutuhkan input argumen berupa fungsi yang memiliki tipe vektor atau *list* seperti contoh berikut:

```
compute_s_n <- function(n){
  x <- 1:n
  tibble(sum = sum(x))
}
s_n <- map_df(n, compute_s_n)
```

TIDYVERSE CONDITIONALS

Analisis data akan sering melibatkan satu atau lebih operasi bersyarat. Pada bagian ini kita akan membahas mengenai dua fungsi `dplyr` yang menyediakan fungsionalitas lebih lanjut untuk melakukan operasi bersyarat.

1. `case_when`

Fungsi `case_when` berguna untuk membuat vektor pernyataan bersyarat. Fungsi ini mirip dengan `IFELSE` tetapi dapat menampilkan sejumlah opsi nilai, tidak hanya BENAR atau SALAH. Berikut adalah contoh implementasi `case_when` untuk menganalisa sifat angka: negatif, positif, atau 0:

```
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative", x > 0 ~ "Positive", TRUE ~ "Zero")
#> [1] "Negative" "Negative" "Zero" "Positive" "Positive"
```

Umumnya, fungsi ini digunakan untuk mengkategorikan *input* berdasarkan variabel yang ada. Sebagai contoh, misalkan kita ingin membandingkan tingkat pembunuhan di tiga kelompok regional: *New England*, *West Coast*, *South*, dan lainnya (*others*). Untuk setiap negara bagian, kita akan mengidentifikasi apakah lokasinya berada di *New England*, jika tidak kita lakukan evaluasi lagi apakah termasuk regional *West Coast*, terakhir, akan dilakukan evaluasi apakah berada pada regional *South*. Jika hasil evaluasi pada ketiganya bernilai SALAH, output yang ditampilkan adalah diluar tiga kategori tersebut (*others*).

```
murders %>%
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New England",
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) %>%
  group_by(group) %>%
  summarize(rate = sum(total) / sum(population) * 10^5)
#> # A tibble: 4 x 2
#> group rate
#> <chr> <dbl>
#> 1 New England 1.72
#> 2 Other 2.71
#> 3 South 3.63
#> 4 West Coast 2.90
```

2. `between`

Operasi umum lainnya yang sering digunakan dalam analisis data adalah untuk menentukan apakah suatu nilai berada di dalam suatu interval. Sebagai contoh untuk memeriksa apakah elemen-elemen vektor x berada di antara a dan b kita dapat melakukan evaluasi dengan:

```
x >= a & x <= b
```

Namun, pada `tidyverse`, kita memiliki fungsi `between` untuk melakukan operasi yang sama.

```
between(x, a, b)
```

IMPORTING DATA

Tahap terpenting sebelum kita melakukan impor data adalah menentukan alamat dimana berkas atau dokumen yang akan diimpor disimpan. Praktek terbaik pertama adalah menyimpan berkas data, skrip analisis, dan keluaran analisis di dalam satu direktori utama serta membaginya ke dalam beberapa subdirektori. Sebagai contoh, dalam modul ini semua material disimpan dalam satu direktori "praktikumDS". Praktek terbaik kedua adalah menuliskan alamat berkas atau dokumen relatif terhadap direktori utama. Hindari menulis alamat berkas relatif terhadap alamat *hardisk*, contohnya "C:/Documents/.." atau "/home/user/Documents".

1. Alamat Berkas

Dalam modul ini kita akan menggunakan paket `here` untuk menentukan dan menuliskan alamat berkas atau dokumen. Aktifkanlah paket `here` terlebih dahulu

```
library(here)
```

Kemudian periksalah alamat direktori utama berada dengan cara menjalankan fungsi `dr_here`. Mengapa direktori utama terdeteksi dan berada pada alamat tersebut?

```
dr_here()
```

Kita juga dapat menggunakan fungsi `here` untuk mendapatkan alamat direktori utama. Selanjutnya bagaimana cara untuk menuliskan alamat suatu berkas atau dokumen dengan cara menggunakan fungsi `here`? Dalam contoh berikut kita dapat menentukan alamat dari berkas "*udara_bandung.xlsx*" yang tersimpan dalam subdirektori data-raw.

```
here() # dimanakah alamat direktori utama/Project berada?
here("data-raw", "udara_bandung.xlsx")
```

Keuntungan menggunakan fungsi `here` adalah kita tidak perlu menuliskan alamat berkas atau dokumen secara manual. Selain itu fungsi `here` juga akan menangani masalah perbedaan cara penulisan alamat yang umumnya berbeda antar sistem operasi komputer.

2. Impor berkas Microsoft Excel

Spreadsheet merupakan format dokumen yang sangat umum kita temukan dan pergunakan. Oleh karena itu, dalam modul ini pertama kita akan belajar bagaimana cara mengimpor berkas *spreadsheet* dari Microsoft Excel yang notabene sering kita pergunakan.

Secara baku, R tidak dapat mengenali dan mengimpor berkas Excel (.xls/.xlsx). Namun kita dapat menggunakan paket `readxl` untuk melakukan hal tersebut. Adapun fungsi yang dapat dipergunakan adalah `read_xls` atau `read_xlsx` tergantung ekstensi berkas Excel tersebut. Namun, kita juga dapat menggunakan satu fungsi `read_excel` yang secara otomatis dapat menebak ekstensi berkas Excel yang kita miliki.

Pertama-tama kita akan mengaktifkan paket `readxl` dan dan mengimpor berkas "*udara_bandung.xlsx*" sebagai obyek R bernama '*udara_bandung*'. Berkas tersebut berisi informasi mengenai kualitas udara di kota Bandung pada periode tertentu. Jangan lupa untuk menuliskan alamat berkas dengan menggunakan fungsi `here` seperti contoh berikut.

```
library(readxl)
udara_bandung <- read_excel(path = here("data-raw", "udara_bandung.xlsx"))
```

Selanjutnya cobalah buka berkas "*udara_bandung.xlsx*" menggunakan aplikasi Microsoft Excel atau sejenisnya. Ada berapa lembar kerja (*sheet*) yang terdapat dalam berkas tersebut? Bagaimana cara Anda mengimpor lembar kerja ke-2 dari berkas Excel tersebut dan menyimpannya sebagai obyek R bernama '*udara_badung_gedebage*'?

```
udara_bandung_gedebage <- read_excel(here("data-raw", udara_bandung.xlsx"),
  sheet = 2)
udara_bandung_gedebage
```

3. Impor berkas *Delimited*

Selain berkas *spreadsheet* dari Microsoft Excel, salah satu format berkas yang sering digunakan adalah *delimited files*. Berkas *delimited* merupakan format *universal* dalam artian dapat dikenali, dibaca, dan dipergunakan oleh banyak program tanpa harus menggunakan fitur tambahan. Ekstensi berkas *delimited* yang sering kita jumpai umumnya adalah .txt dan .csv.

Di R terdapat berbagai fungsi dan paket yang dapat digunakan untuk membaca dan menulis berkas *delimited*. Contohnya dengan paket `utils` (paket bawaan dari R) terdapat fungsi `read.table`, `read.csv`, `read.csv2`, `read.delim`, dan `read.delim2`. Selain itu terdapat pula fungsi `read_table`, `read_table2`, `read_csv`, `read_csv2`, `read_delim`, dan `read_delim2` dari paket `readr`. Terdapat juga fungsi dari paket lainnya dengan tujuan serupa misalnya `fread` dari `data.table` dan fungsi `vroom` dari paket `vroom`.

Mengapa ada banyak fungsi dan paket berbeda untuk tujuan yang sama? Hal utama yang menjadi dasar perbedaan adalah konsistensi, kemudahan penggunaan fungsi bagi pengguna,

dan kecepatan dalam membaca berkas. Berdasarkan pertimbangan dari aspek-aspek tersebut, dalam modul ini kita akan fokus menggunakan paket `vroom` untuk mengimpor berkas *delimited*.

Aktifkanlah paket `vroom` dan bacalah dokumentasi fungsi `vroom`

```
library(vroom)
help(vroom)
```

Sekarang cobalah impor berkas "*anggaran-dinkes-2013.csv*" yang berada dalam subdirektori data-raw dengan menggunakan fungsi `vroom` dan simpanlah sebagai obyek R dengan nama '*dinkes_2013_vroom'*. Kemudian lakukanlah inspeksi terhadap '*dinkes_2013_vroom*' tersebut. Ada berapa observasi dan variabel? Apa saja nama dan tipe dari setiap variabel data tersebut?

```
dinkes_2013_vroom <- vroom(here("data-raw", "anggaran-dinkes-2013.csv"))
dinkes_2013_vroom
str(dinkes_2013_vroom)
summary(dinkes_2013_vroom)
```

Sebagai perbandingan, sekarang imporlah berkas yang sama namun menggunakan fungsi `read.csv` dari paket `utils` dan simpanlah hasilnya sebagai obyek R bernama '*dinkes_2013_utils*'! Lakukanlah inspeksi serupa dengan '*dinkes_2013_vroom*'. Apakah ada perbedaan yang Anda temukan?

```
dinkes_2013_utils <- read.csv(here("data-raw", "anggaran-dinkes-2013.csv"))
dinkes_2013_utils
str(dinkes_2013_utils)
```

4. Impor banyak berkas *Delimited*

Bukalah subdirektori data-raw melalui Files pane (Ctrl + 5) atau melalui Files. Di dalam subdirektori tersebut terdapat beberapa berkas yang berisi informasi mengenai anggaran Dinas Kesehatan Kota Bandung pada tahun 2013 hingga 2018. Bagaimana cara untuk mengimpor semua berkas tersebut sekaligus?

Selanjutnya kita akan mencoba dengan dua cara yang berbeda untuk mengimpor beberapa berkas sekaligus. Cara pertama adalah dengan menggunakan *functional iteration* bawaan R, dan cara kedua adalah dengan menggunakan fungsi `vroom`.

Semua cara yang akan dilakukan memiliki satu tahap pertama yang sama, yaitu membuat daftar alamat berkas-berkas yang akan diimpor. Kita dapat membuat daftar alamat tersebut dengan menggunakan fungsi `list.files`. Sesuaikanlah *pattern* agar alamat yang terseleksi hanyalah alamat berkas yang relevan. Simpan daftar alamat tersebut dalam obyek R bernama 'berkas_anggaran_dinkes`.

```
berkas_anggaran_dinkes <- list.files(path = here("data-raw"), pattern =
"dinkes", full.names = TRUE)
berkas_anggaran_dinkes
```

Berikut merupakan cara pertama untuk mengimpor berkas-berkas tersebut dengan menggunakan *functional iteration*:

```
anggaran_dinkes_lapply <- lapply(berkas_anggaran_dinkes, read.csv)
anggaran_dinkes_lapply <- Reduce(rbind, anggaran_dinkes_lapply)
anggaran_dinkes_lapply
```

Pada cara kedua, kita akan mengimpor beberapa berkas *delimited* sekaligus menggunakan `vroom`:

```
anggaran_dinkes_vroom <- vroom(berkas_anggaran_dinkes)
anggaran_dinkes_vroom
```

Manakah cara yang paling mudah dan cepat? Pergunakan fungsi `identical` untuk menguji apakah obyek '*anggaran_dinkes_lapply*' dan '*anggaran_dinkes_vroom*' persis sama atau tidak.

```
identical(anggaran_dinkes_lapply, anggaran_dinkes_vroom)
```

D. Latihan

1. Gunakan `as_tibble` untuk mengkonversi tabel *dataset "US murders"* dalam bentuk *tibble* dan simpan dalam objek baru bernama '*murders_tibble*'.
2. Gunakan fungsi `group_by` untuk mengkonversi *dataset "US murders"* menjadi sebuah *tibble* yang dikelompokkan berdasarkan '*region*'.
3. Tulis *script tidyverse* yang menghasilkan *output* yang sama dengan perintah berikut:

```
exp(mean(log(murders$population)))
```

   Gunakan operator *pipe* sehingga setiap fungsi dapat dipanggil tanpa menambahkanargumen. Gunakan *dot operator* untuk mengakses populasi.
4. Gunakan `map_df` untuk membuat *data frame* yang terdiri dari tiga kolom: '*n'*, '*s_n'*, dan '*s_n_2'*. Kolom pertama harus berisi angka 1 hingga 100. Kolom kedua dan ketiga masing-masing harus berisi penjumlahan 1 hingga n, dimana n menyatakan jumlah baris.