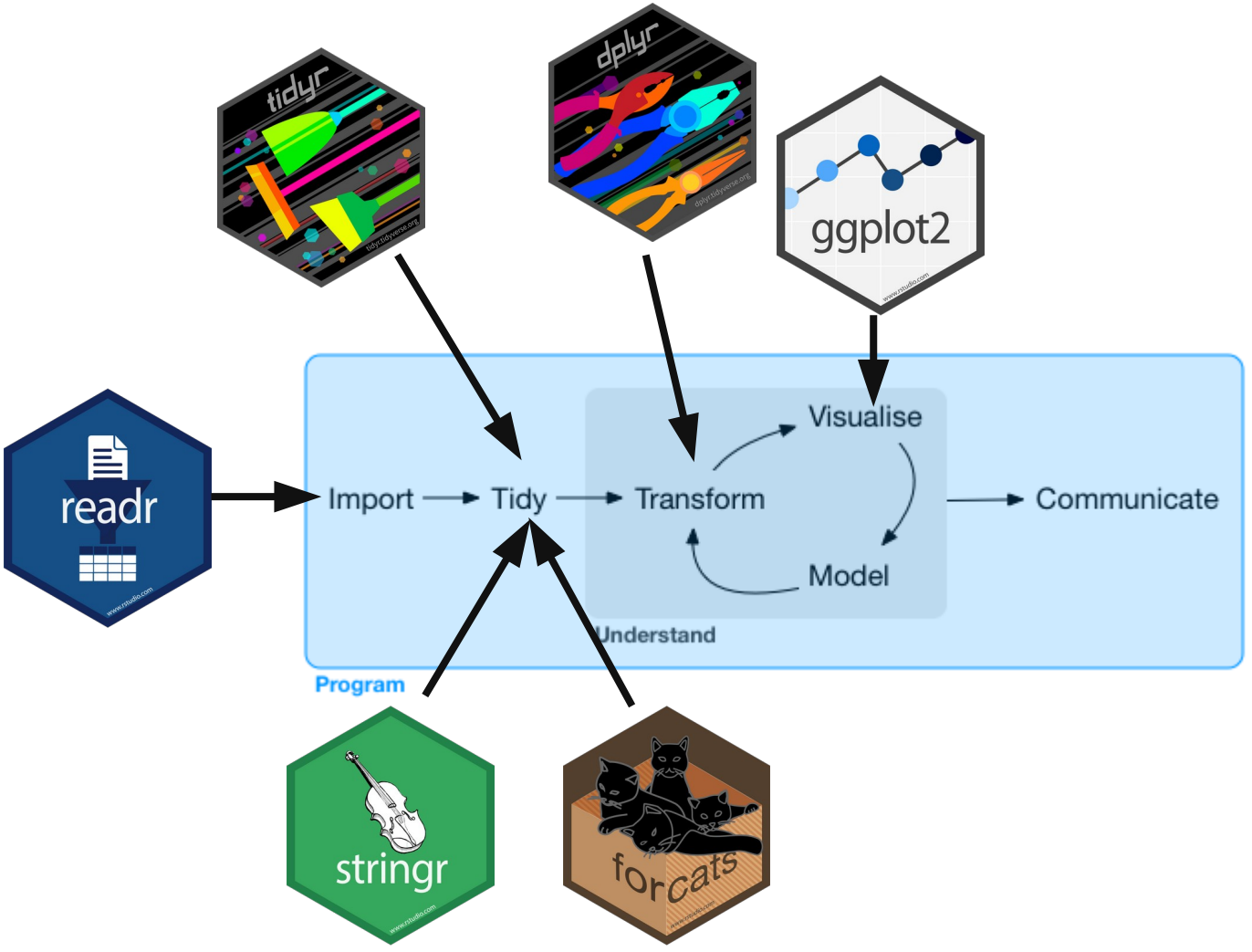
GG606

Pipes & Functions


tidyr
dplyr
ggplot2
readr
Import
Tidy
Transform
Visualise
Model
Communicate
Understand
Program
stringr
forcats

Program

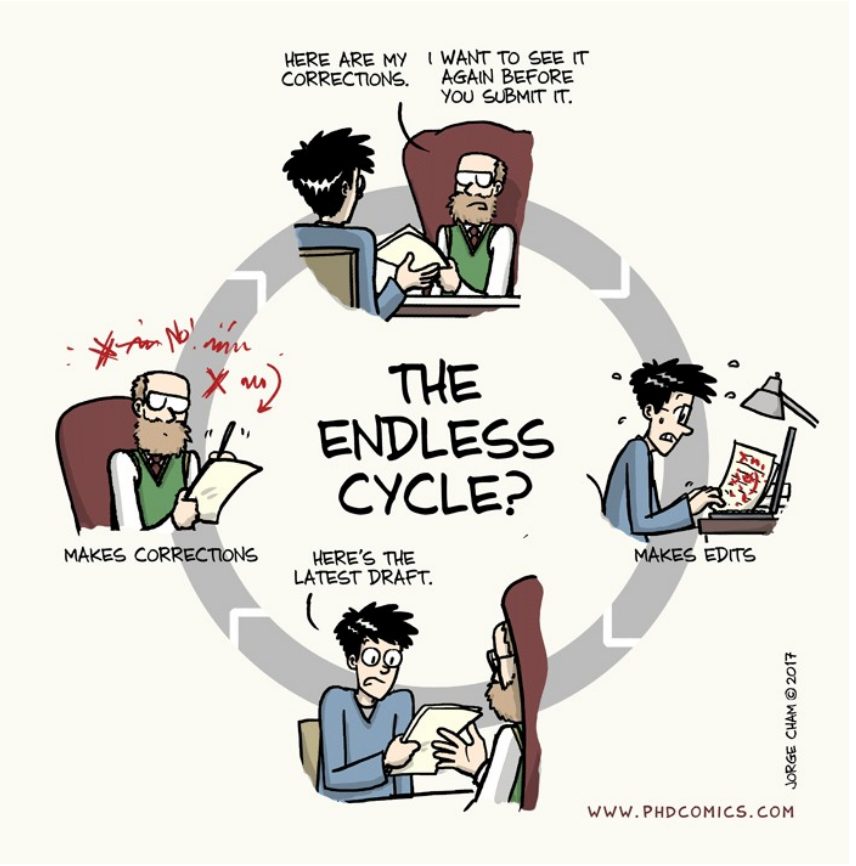
- Programming produces code
- Code is communication

Program

- Programming produces code
- Code is communication
  - Other people
  - Past & future self
- Getting better at programming means getting better at communicating

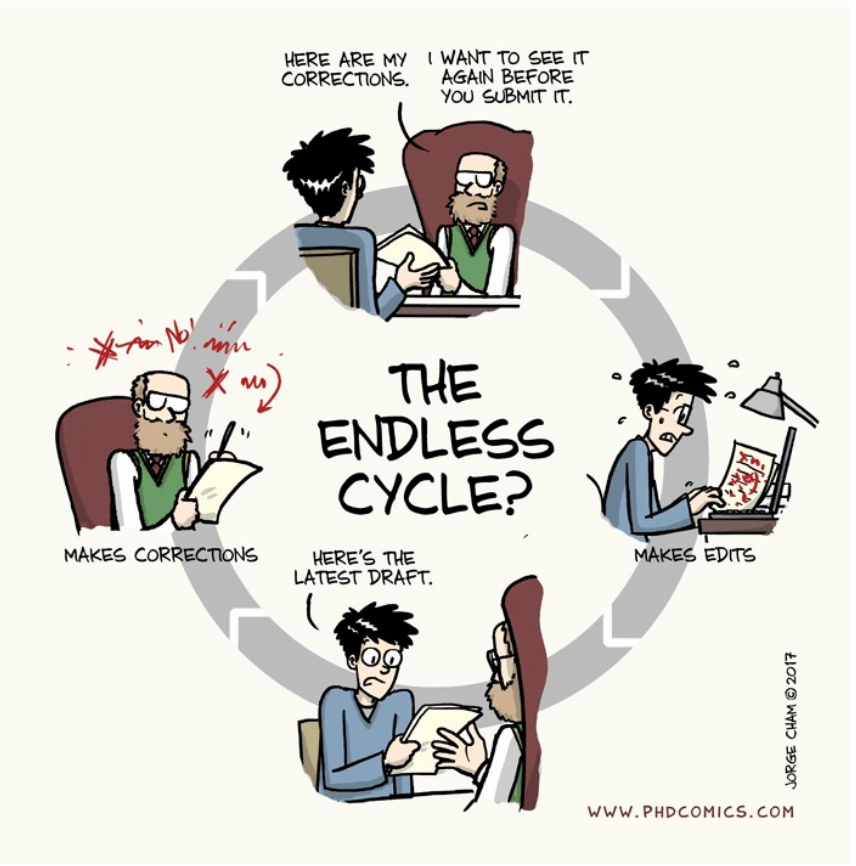
Prose

- How many (rounds of) edits for your thesis?



Prose

- How many (rounds of) edits for your thesis?
- Obvious what your code does?
- Rewrite while ideas are fresh



Time

- Balance of time/effort now vs later
- Pipes: %>% and |>
- Copy-and-paste
- Functions

Time

- Applies equally to spreadsheets, R, Matlab, Python, Julia
- Plan ahead as much as you can
- Maybe you have to redo the work in a new implementation

`magrittr` pipe

- `tidyverse` packages load `magrittr::%>%`
- Great example using pseudocode
  - *Note: ask about pseudocode*

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

This is a popular Children’s poem that is accompanied by hand actions.

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

This is a popular Children's poem that is accompanied by hand actions.

How would we do this in (pseudo)code?

Four ways in code

- Verbs are functions: `hop()`, `scoop()`, `bop()`
  -
  -
  -
  -

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

Four ways in code

- Verbs are functions: `hop()`, `scoop()`, `bop()`

1) Save each intermediate step as a new object
2) Overwrite the original many times
3) Compose functions
4) Use the pipe

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

1) Intermediate steps

- `foo_foo_1 ← hop(foo_foo, through = forest)`
- `foo_foo_2 ← scoop(foo_foo_1, up = field_mice)`
- `foo_foo_3 ← bop(foo_foo_2, on = head)`

1) Intermediate steps

- `foo_foo_1 ← hop(foo_foo, through = forest)`
- `foo_foo_2 ← scoop(foo_foo_1, up = field_mice)`
- `foo_foo_3 ← bop(foo_foo_2, on = head)`
- Natural names for intermediates?
- Cluttered, easy to make mistakes

2) Overwrite original

- `foo_foo ← hop(foo_foo, through = forest)`
- `foo_foo ← scoop(foo_foo, up = field_mice)`
- `foo_foo ← bop(foo_foo, on = head)`

2) Overwrite original

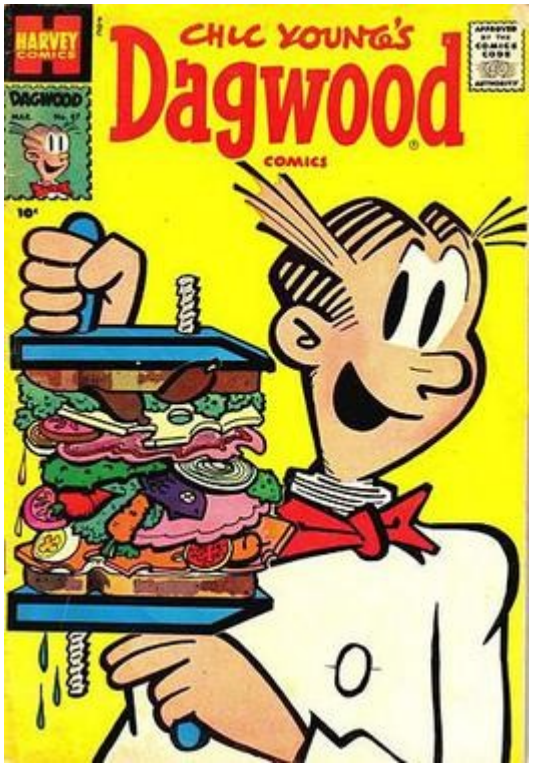
- `foo_foo ← hop(foo_foo, through = forest)`
- `foo_foo ← scoop(foo_foo, up = field_mice)`
- `foo_foo ← bop(foo_foo, on = head)`
- How to debug?
- Repetition obscures what's being changed

3) Function composition

- ```
  bop(
    scoop(
      hop(foo_foo, through = forest),
      up = field_mice
    ),
    on = head
  )
  ```
  ```

3) Function composition

- ```
  bop(
    scoop(
      hop(foo_foo, through = forest),
      up = field_mice
    ),
    on = head
  )
  ```
- Reading inside-out, right-to-left
- Arguments far apart


  ```

4) Pipe

- ```
  foo_foo %>%
    hop(through = forest) %>%
    scoop(up = field_mice) %>%
    bop(on = head)
  ```

Python: sspipe, toolz
  ```

4) Pipe

- ```
  foo_foo %>%
    hop(through = forest) %>%
    scoop(up = field_mice) %>%
    bop(on = head)
  ```
- Focus on verbs, not nouns
- Easily read
  ```

4) Pipe

- ```
  foo_foo %>%
    hop(through = forest) %>%
    scoop(up = field_mice) %>%
    bop(on = head)
  ```

Call this

- ```
  my_pipe ← function(.) {
    . ← hop(., through = forest)
    . ← scoop(., up = field_mice)
    bop(., on = head)
  }
  ```

Get this

- ```
  my_pipe(foo_foo)
  ```

Equivalent to this
Pays no attention to current env
  ```

Other `magrittr` tools

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
```

```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
```

Tee pipe returns left-hand side instead of right-hand side

Other `magrittr` tools

```
mtcars %$%
   cor(disp, mpg)
```

```
> mtcars %>%
+     cor(disp, mpg)
Error in cor(., disp, mpg) : invalid 'use' argument
In addition: Warning message:
In if (is.na(na.method)) stop("invalid 'use' argument") :
  the condition has length > 1 and only the first element will be used
```

Exposition pipe ‘explodes’ out variables for functions that need vectors

Other `magrittr` tools

```
mtcars ← mtcars %>%
  transform(cyl = cyl * 2)

mtcars %<>% transform(cyl = cyl * 2)
```

Assignment pipe combines ← with %>%

functions

Why functions

- Good name makes code easier to understand
- Only update on one place (not many)
- Eliminate changes of making incidental mistakes when copy-pasting
- True across all languages

When should you write a function?

- ```
  df ← tibble::tibble(
    a = rnorm(10),
    b = rnorm(10),
    c = rnorm(10),
    d = rnorm(10)
  )
  ```
- ```
  df$a ← (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
  ```
- ```
  df$b ← (df$b - min(df$b, na.rm = TRUE)) /
    (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
  ```
- ```
  df$c ← (df$c - min(df$c, na.rm = TRUE)) /
    (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
  ```
- ```
  df$d ← (df$d - min(df$d, na.rm = TRUE)) /
    (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
  ```

Good example
What does it do?
  ```

When should you write a function?

- ```
  df$a ← (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) -
     min(df$a, na.rm = TRUE))
  ```

How many inputs
  ```

When should you write a function?

- ```
  (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) -
     min(df$a, na.rm = TRUE))
  ```
- `x ← df$a`

How many inputs? 1
Duplication?
  ```

When should you write a function?

- ```
  (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) -
     min(df$a, na.rm = TRUE))
  ```
- ```
  x ← df$a
  ```
- ```
  (x - min(x, na.rm = TRUE)) /
    (max(x, na.rm = TRUE) -
     min(x, na.rm = TRUE))
  ```

How many inputs? 1
Duplication?
  ```

When should you write a function?

- `rng ← range(x, na.rm = TRUE)`
- `(x - rng[1]) / (rng[2] - rng[1])`

Intermediate calculations into name variable
Still work?

When should you write a function?

arguments

name

- ```
  rescale01 ← function(x) {
    rng ← range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] – rng[1])
  }
  ```

body

- ```
  rescale01(c(0, 5, 10))
  ```

Simplified

Still work? More tests

When should you write a function?

- Pick a unique and descriptive name: `rescale01`
- Inputs to the function, arguments, inside `function()`
- Your code in the body of the function with `{}` immediately after `function( ... )`

Make function after making it work with simple input

When should you write a function?

- `rescale01(c(0, 5, 10))`
- `rescale01(c(-10, 0, 10))`
- `rescale01(c(1, 2, 3, NA, 5))`

Make function after making it work with simple input
Informal tests, Unit testing

When should you write a function?

- `df$a ← rescale01(df$a)`
- `df$b ← rescale01(df$b)`
- `df$c ← rescale01(df$c)`
- `df$d ← rescale01(df$d)`

One set of code
Easy to read

When should you write a function?

- `rescale01(c(1:10, Inf))`

```
> rescale01(c(1:10, Inf))
 [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Easy to fix

When should you write a function?

- `rescale01(c(1:10, Inf))`
- ```
  rescale01 ← function(x) {
    rng ← range(x, na.rm = TRUE, finite = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
  }
  ```
- `rescale01(c(1:10, Inf))`

Easy to fix
  ```

Make Functions

- `mean(is.na(x))`
- `x / sum(x, na.rm = TRUE)`
- `sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)`

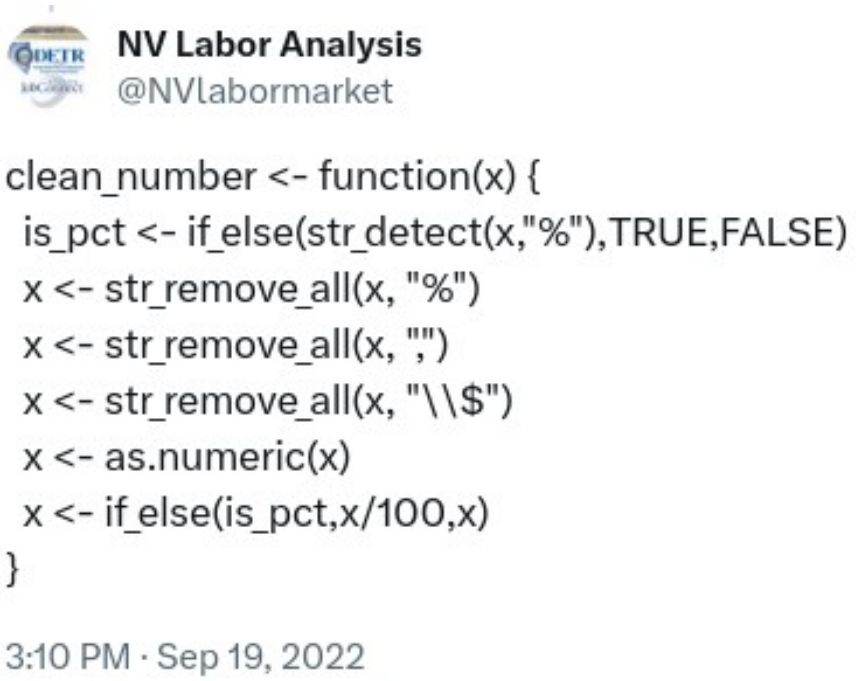
`case_when` in functions

- Force the data to be within a range by using `case_when()` inside the function
- Other `case_when()` examples?

```
clamp ← function(x, min, max) {
  case_when(
    x < min ~ min,
    x > max ~ max,
    .default = x
  )
}

clamp(1:10, min = 3, max = 7)
#>  [1] 3 3 3 4 5 6 7 7 7 7
```

NV Labor Stats




```
clean_number ← function(x) {
  is_pct ← str_detect(x, "%")
  num ← x ▷
    str_remove_all("%") ▷
    str_remove_all(",") ▷
    str_remove_all(fixed("$")) ▷
    as.numeric()
  if_else(is_pct, num / 100, num)
}


clean_number("$12,300")
#> [1] 12300
clean_number("45%")
#> [1] 0.45
```

Specialised one-step

```
fix_na ← function(x) {
  if_else(x %in% c(-2000, -999,
          -200), NA, x)
}
```

- Sometimes NA data (or other special data) are coded with different values and need to be fixed for the data are used

Indirect & tidy evaluation

- embracing {{ }}
- data masking & tidy selection
- !! bang bang

Indirection bc of tidy evaluation

- Goal of function is to compute the mean of `mean_var` grouped by `group_var`
- How dplyr verbs work

```
grouped_mean ← function(df,
group_var, mean_var) {
  df ▷
    group_by(group_var) ▷
    summarize(mean(mean_var))
}
```

Indirection bc of tidy evaluation

```
diamonds ▷ grouped_mean(cut, carat)
#> Error in `group_by()`:
#> ! Must group by variables found in
`.data`.
#>    Column `group_var` is not found.
```

Indirection bc of tidy evaluation

```
df <- tibble(
  mean_var = 1,
  group_var = "g",
  group = 1,
  x = 10,
  y = 100
)

df |> grouped_mean(group, x)
#> # A tibble: 1 × 2
#>   group_var `mean(mean_var)`
#>   <chr>                <dbl>
#> 1 g                        1
df |> grouped_mean(group, y)
#> # A tibble: 1 × 2
#>   group_var `mean(mean_var)`
#>   <chr>                <dbl>
#> 1 g                        1
```

```
df ▷ group_by(group_var) ▷
summarize(mean(mean_var))
```

instead of

```
df ▷ group_by(group) ▷
summarize(mean(x))
```

or

```
df ▷ group_by(group) ▷
summarize(mean(y))
```

Embracing {}

```
grouped_mean ← function(df, group_var, mean_var) {
  df ▷
    group_by({{ group_var }}) ▷
    summarize(mean({{ mean_var }}))
}

df ▷ grouped_mean(group, x)
#> # A tibble: 1 × 2
#>   group `mean(x)`
#>   <dbl>     <dbl>
#> 1     1        10
```

When to embrace

- Look in documentation ;-)
- *Data-masking*: this is used in functions like `arrange()`, `filter()`, and `summarize()` that compute with variables.
- *Tidy-selection*: this is used for functions like `select()`, `relocate()`, and `rename()` that select variables.

Data exploration case

```
summary6 <- function(data, var) {
  data |> summarize(
    min = min({{ var }}, na.rm = TRUE),
    mean = mean({{ var }}, na.rm = TRUE),
    median = median({{ var }}, na.rm = TRUE),
    max = max({{ var }}, na.rm = TRUE),
    n = n(),
    n_miss = sum(is.na({{ var }})),
    .groups = "drop"
  )
}

diamonds |> summary6(carat)
#> # A tibble: 1 × 6
#>     min  mean median   max     n n_miss
#>   <dbl> <dbl>  <dbl> <dbl> <int>  <int>
#> 1   0.2 0.798    0.7  5.01 53940      0
```

- Good to use `groups = "drop"` to return ungrouped data

Great for grouped data

```
diamonds |>
  group_by(cut) |>
  summary6(carat)
#> # A tibble: 5 × 7
#>   cut         min  mean median   max     n n_miss
#>   <ord>     <dbl> <dbl>  <dbl> <dbl> <int>  <int>
#> 1 Fair       0.22 1.05    1     5.01  1610      0
#> 2 Good       0.23 0.849   0.82  3.01  4906      0
#> 3 Very Good  0.2  0.806   0.71  4    12082      0
#> 4 Premium    0.2  0.892   0.86  4.01 13791      0
#> 5 Ideal      0.2  0.703   0.54  3.5  21551      0
```

Data-masking vs. tidy-selection

```
count_missing <- function(df, group_vars, x_var) {
  df |>
    group_by({{ group_vars }}) |>
    summarize(
      n_miss = sum(is.na({{ x_var }})),
      .groups = "drop"
    )
}

flights |>
  count_missing(c(year, month, day), dep_time)
#> Error in `group_by()`:
#> ℹ In argument: `c(year, month, day)`.
#> Caused by error:
#> ! `c(year, month, day)` must be size 336776 or 1, not 1010328.
```

- Select variables inside a fucntion that uses data-masking
- `group_by()` uses data-masking not tidy-selection

Data-masking vs. tidy-selection

```
count_missing <- function(df, group_vars, x_var) {
  df |>
    group_by(pick({{ group_vars }})) |>
    summarize(
      n_miss = sum(is.na({{ x_var }})),
      .groups = "drop"
  )
}

flights |>
  count_missing(c(year, month, day), dep_time)
#> # A tibble: 365 × 4
#>    year month   day n_miss
#>   <int> <int> <int>  <int>
#> 1  2013     1     1      4
#> 2  2013     1     2      8
#> 3  2013     1     3     10
#> 4  2013     1     4      6
#> 5  2013     1     5      3
#> 6  2013     1     6      1
#> # i 359 more rows
```

- Select variables inside a fucntion that uses data-masking
- `group_by()` uses data-masking not tidy-selection
- `pick()` to use tidy-selection inside data-masking function

Ramon Gallego
@pollicipes

I wrote this function as a way of creating more readable contingency tables with two variables.

```
tally_wide <- function (tibble, rows, cols, wt = NULL){

  rows = enquo(rows)
  cols = enquo(cols)
  weight = enquo(wt)


  tibble %>%
    group_by(!!rows, !!cols) %>%
    tally (.,wt = {{wt}}) %>%
    pivot_wider(names_from=  !!cols,
                values_from = n, names_repair = "minimal")

}
```

5:06 PM · Sep 18, 2022

```
# https://twitter.com/pollicipes/status/1571606508944719876
count_wide <- function(data, rows, cols) {
  data |>
    count(pick(c({{ rows }}, {{ cols }}))) |>
    pivot_wider(
      names_from = {{ cols }},
      values_from = n,
      names_sort = TRUE,
      values_fill = 0
    )
}

diamonds |> count_wide(c(clarity, color), cut)
#> # A tibble: 56 × 7
#>   clarity color  Fair  Good `Very Good` Premium Ideal
#>   <ord>   <ord> <int> <int>       <int>   <int> <int>
#> 1 I1      D         4     8           5      12    13
#> 2 I1      E         9    23          22      30    18
#> 3 I1      F        35    19          13      34    42
#> 4 I1      G        53    19          16      46    16
#> 5 I1      H        52    14          12      46    38
#> 6 I1      I        34     9           8      24    17
#> # ℹ 50 more rows
```

Homework

- Functions for variance and skewness

$$\mathrm{Var}(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$$\mathrm{Skew}(x) = \frac{\frac{1}{n-2}\left(\sum_{i=1}^{n}(x_i - \bar{x})^3\right)}{\mathrm{Var}(x)^{3/2}}$$