Iterations and Functionals

Stat 220

Bastola

February 09 2022

Why repeat ourselves?

```
tinydata <- tribble(
  ~case, ~x, ~y, ~z,
  "a", 5, 3, -2,
  "b", 7, 1, -5,
  "c", 9, 12, -3
)
```

```
tinydata
# A tibble: 3 × 4
  case      x     y     z
  <chr> <dbl> <dbl> <dbl>
1 a         5     3    -2
2 b         7     1    -5
3 c         9    12    -3
```

Find the mean of each columns

```
mean(tinydata$x)
[1] 7
```

```
mean(tinydata$y)
[1] 5.333333
```

```
mean(tinydata$z)
[1] -3.333333
```

Iteration

> Iteration is the process of repeating the same action over and over again

Multiple ways to do in R

- **loops** using `for`, `while`, etc
- **vectorized** functions that apply the same function to every element of a vector
- **functional** functions that apply the same function to elements in a vector, matrix, data frame, or list

for loops

A way to iterate through a series of items stored as data object in R.

```
items <- c("grapes","bananas","chocolate","bread")
for(i in items){
  print(i)
}
[1] "grapes"
[1] "bananas"
[1] "chocolate"
[1] "bread"
```

```
i <- items[1]
print(i)
[1] "grapes"
```

```
i <- items[2]
print(i)
[1] "bananas"
```

for loop components

the for() function which we use to specify

- what object we're drawing from and
- what object we are writing to.

```
for( i  in  items  )
     ^         ^
     |         |___ object we are drawing from
     |
obj. we write each item to
```

for loop components

The brackets {}

- Inside the brackets we house the code that is going to happen each iteration.

```
for( i  in  items  ){
    |~~~~~~~~~~~~~~~~~|
    |~~~~~~~~~~~~~~~~~|
    |~~~~~~~~~~~~~~~~~| code we need perform on each iteration.
    |~~~~~~~~~~~~~~~~~|
    |~~~~~~~~~~~~~~~~~|
   }
```

for() loops and storing output

```
letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
basket <- rep(NA,10) # numeric vector of length 3
basket
 [1] NA NA NA NA NA NA NA NA NA NA
```

```
# Each loop, we store the output of some code.
for(i in 1:10){
  basket[i] <- str_glue(letters[i],letters[i+1])
}
basket
 [1] "ab" "bc" "cd" "de" "ef" "fg" "gh" "hi" "ij" "jk"
```

For loops `tinydata`

- Let's iterate calculation of column means:

```
my_means <- rep(NA, 3)
my_means
[1] NA NA NA
```

```
for (i in 1:3) {  # three columns to get the mean for
  my_means[i] <- mean(tinydata[[i+1]])  # mean of col. i+1 (skip col. 1)
}
my_means
[1]  7.000000  5.333333 -3.333333
```

For loops: preallocation output space

- About 12 seconds without preallocation and less than a second with (elapsed time).

Without preallocation

```
system.time({
  output <- NULL
  for (i in 1:100000) {
    output <- c(output, i)
  }
})
   user  system elapsed
 12.432   0.056  12.496
```

With preallocation

```
system.time({
  output <- rep(NA, 100000)
  for (i in 1:100000) {
    output[i] <- i
  }
})
   user  system elapsed
   0.01    0.00    0.01
```

For loops: index vector

- seq_along(df) index based on columns of data frame

```
seq_along(tinydata)
[1] 1 2 3 4
```

- Another common way of indexing

```
1:nrow(tinydata)
[1] 1 2 3
```

- Along the columns

```
1:ncol(tinydata)
[1] 1 2 3 4
```

For loop with ifelse

```
my_means <- rep(NA, ncol(tinydata))

for (i in seq_along(tinydata)){  # iterate over all columns
  my_means[i] <- ifelse(is.numeric(tinydata[[i]]), mean(tinydata[[i]]), NA)
}
```

```
my_means
[1]         NA   7.000000   5.333333 -3.333333
```

Function for conditional evaluation

- if x is numeric then standardize, else just return x

```
standardize <- function(x, ...){   # ... used for arbitrary number of arguments
  if (is.numeric(x)){              # condition
    (x - mean(x, ...))/sd(x, ...)  # if TRUE, standardize
  } else{                          # else (FALSE)
    x                              # return x unchanged
  }
}
```

```
standardize(c(1,2,3,4))
[1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

```
standardize(c("a", "b", "2", NA), na.rm = TRUE)
[1] "a" "b" "2" NA
```

Standardizing tinydata

- Allocate storage in a new data frame:

```
scaled_tinydata <- tinydata %>%
  mutate(
    x = NA,
    y = NA,
    z = NA
  )
```

```
scaled_tinydata
# A tibble: 3 × 4
  case  x     y     z
  <chr> <lgl> <lgl> <lgl>
1 a     NA    NA    NA
2 b     NA    NA    NA
3 c     NA    NA    NA
```

Standardizing tinydata

- For loop for iteration:

```
for (i in seq_along(tinydata)){
    scaled_tinydata[, i] <- standardize(tinydata[[i]])
}
```

```
scaled_tinydata
# A tibble: 3 × 4
  case      x      y      z
  <chr> <dbl>  <dbl>  <dbl>
1 a        -1 -0.398  0.873
2 b         0 -0.740 -1.09
3 c         1  1.14   0.218
```

Vectorization

- A vectorized function will apply the same operation (function) to each element of a vector.
  - avoid loops by applying operations to each element of a vector

Vectorization

```
x <- c(10,20,30,40)
log10(x)   # log10 is a vectorized function
[1] 1.000000 1.301030 1.477121 1.602060
```

- The for loop version

```
out <- rep(NA, 4)
for (i in 1:4)
  { out[i] <- log10(x[i]) }
out
[1] 1.000000 1.301030 1.477121 1.602060
```

Your Turn 1

Please git clone the github repository on simple iterations. Write a `for` loop that calculates the mean of the numeric variables in the `penguins` data set and stores the means in a named vector.

```
Rows: 344
Columns: 8
$ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel…
$ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse…
$ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, …
$ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, …
$ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186…
$ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, …
$ sex               <fct> male, female, female, NA, female, male, female, male…
$ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007…
```

04:00

Functionals

- A **functional** function will apply the same operation (function) to each element of a vector, matrix, data frame or list.

- base-R: `apply` family of commands

  -`purrr` package: `map` family of commands

`apply` family of commands

- R has a family of commands that apply a function to different parts of a vector, matrix or data frame.

lapply(X, FUN):

applies the FUN to each element in the vector/list X. Returns a list with length equal to that of the vector

sapply:

works like `lapply` but returns a vector (so `FUN` can only return one value)

`apply` family of commands

- R has a family of commands that apply a function to different parts of a vector, matrix or data frame.

apply(matrix, MARGIN, FUN):

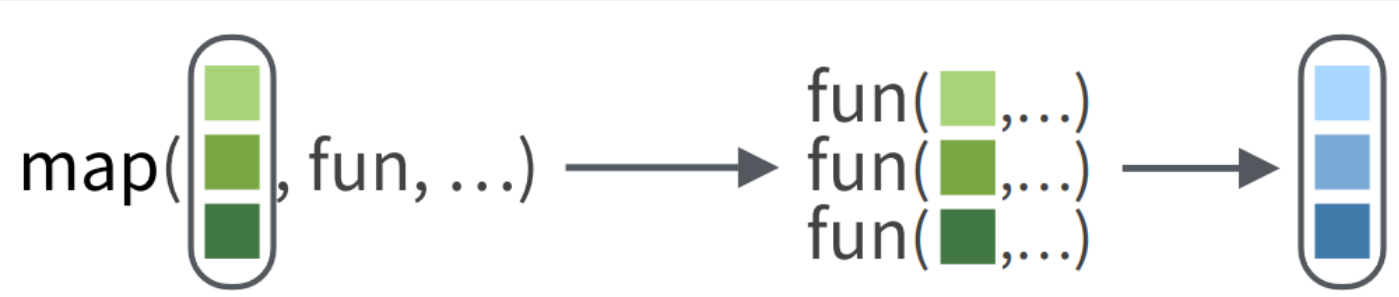
applies the function FUN to the matrix. $MARGIN$ given (1=row, 2=column, c(1,2)=rows and cols). Returns an atomic **vector** or **matrix**

tapply(x,y,FUN)):

applies FUN to atomic vector (variable) x for each group in categorical variable y. Returns an atomic **vector** with a $\dim$ $names$ attribute

purr package

powerful package for iteration with the same functionality as apply commands, but more readable (according to Hadley)



- map(.x, .f) maps the function .f to elements in the vector/list .x

`lapply` tiny example

```
lapply(tinydata, FUN = mean)
$case
[1] NA

$x
[1] 7

$y
[1] 5.333333

$z
[1] -3.333333
```

- a 3x4 data frame is **summarized** in a list of length 4.

- R sees tinydata as a list whose elements are column vectors (variables)
- the FUN is applied to each list element
- a list is returned
- length is the number of variables in the data frame

map

In purrr, the map function is equivalent to lapply

```
library(purrr)
map(tinydata, .f = mean)
$case
[1] NA

$x
[1] 7

$y
[1] 5.333333

$z
[1] -3.333333
```

`sapply` tiny example

Output is an atomic vector (`simplify`)

```
sapply(tinydata, FUN = mean)
     case          x          y          z
       NA   7.000000   5.333333 -3.333333
```

- a 3x4 data frame is **summarized** in a vector of length 4.

map_dbl

`map_dbl` is equivalent to `sapply`

```
map_dbl(tinydata, .f = mean)
     case          x          y          z
       NA  7.000000  5.333333 -3.333333
```

map_df

`map_df` returns a data frame instead of a vector

```
map_df(tinydata, .f = mean)
# A tibble: 1 × 4
   case     x     y     z
  <dbl> <dbl> <dbl> <dbl>
1    NA     7  5.33 -3.33
```

- No equivalency in base-R `apply`!

Iterate or dplyr?!

- `summarize_all`, `summarize_if`, `summarize_at` are all options that apply `.funs` to **columns** of a data frame
- `if` option needs a logical function that determines which **columns** to apply the `.funs` to

```
tinydata %>%
  summarize_if(is.numeric, .funs = mean)
# A tibble: 1 × 3
      x     y     z
  <dbl> <dbl> <dbl>
1     7  5.33 -3.33
```

functionals: single function that mutates

`standardize` function gives us a list of standardized values

```
lapply(tinydata, FUN = standardize)
$case
[1] "a" "b" "c"

$x
[1] -1  0  1

$y
[1] -0.3982161 -0.7395442  1.1377602

$z
[1]  0.8728716 -1.0910895  0.2182179
```

- a 3x4 data frame is **mutated** to a list of 4 vectors of length 3 each

lapply tiny example

Using dplyr::bind_cols converts the list to a data frame with variables equal to list entries

```
lapply(tinydata, FUN = standardize) %>%
  bind_cols()
# A tibble: 3 × 4
  case      x      y      z
  <chr> <dbl>  <dbl>  <dbl>
1 a        -1 -0.398  0.873
2 b         0 -0.740 -1.09
3 c         1  1.14   0.218
```

map_df

In `purrr`, the `map_df` is equal to `lapply` + `bind_cols`:

```
map_df(tinydata, .f = standardize)
# A tibble: 3 × 4
  case      x      y      z
  <chr> <dbl>  <dbl>  <dbl>
1 a        -1 -0.398  0.873
2 b         0 -0.740 -1.09
3 c         1  1.14   0.218
```

- a 3x4 data frame is mutated to **standardized** 3x4 data frame

Iterate or dplyr?!

- `mutate_all`, `mutate_if`, `mutate_at` are all options that apply `.funs` to columns of a data frame
- if option needs a logical function that determines which columns to apply the `.funs` to

```
tinydata %>%
  mutate_if(is.numeric, .funs = standardize)
# A tibble: 3 × 4
  case      x      y      z
  <chr> <dbl>  <dbl>  <dbl>
1 a        -1 -0.398  0.873
2 b         0 -0.740 -1.09
3 c         1  1.14   0.218
```

applying multiple functions

- Let's get the 0.1 and 0.9 quantile for variables in `tinydata`

```
quantile(tinydata$x, probs = c(.1, .9))
10% 90%
5.4 8.6
```

```
quantile(tinydata$y, probs = c(.1, .9))
 10%  90%
 1.4 10.2
```

```
quantile(tinydata$z, probs = c(.1, .9))
 10%  90%
-4.6 -2.2
```

- the function output is a vector of length 2 (same lengths as probs)

map_df: getting quantiles

```
tinydata %>%
  select_if(is.numeric) %>%   # only numeric columns
  map_df(
    .f = quantile,    # function to apply to cols
    probs = c(.1, .9)) # extra function arguments
# A tibble: 3 × 2
  `10%` `90%`
  <dbl> <dbl>
1   5.4   8.6
2   1.4  10.2
3  -4.6  -2.2
```

map_df: getting quantiles

Can use `.id` to record the variable names from tinydata:

```
tinydata %>%
  select_if(is.numeric) %>%
  map_df(
    .f = quantile,
    probs = c(.1, .9),
    .id = "variable")
# A tibble: 3 × 3
  variable `10%` `90%`
  <chr>    <dbl> <dbl>
1 x          5.4   8.6
2 y          1.4  10.2
3 z         -4.6  -2.2
```

`map_df` options

There are two types of `map_df`

- `map_dfr`: which row binds the list created by map
  - entries in the list are rows in the data frame
- `map_dfc`: which column binds the list created by map
  - entries in the list are columns in the data frame

Iterate or dplyr?!

- `summarize_all`, `summarize_if`, `summarize_at` can work with functions like `quantile` that return multiple values.
- the **form** of the output is a transposed version of `map_df`

```
tinydata %>%
    summarize_if(is.numeric, .funs = quantile, probs = c(.1, .9))
# A tibble: 2 × 3
      x     y     z
  <dbl> <dbl> <dbl>
1   5.4   1.4  -4.6
2   8.6  10.2  -2.2
```

- **rows** = 0.1 and 0.9 quantiles
- **cols** = variables

Iterate or dplyr?!

- We need to manually add a percentile variable to help us ID the value in each row

```
tinydata %>%
    summarize_if(is.numeric, .funs = quantile, probs = c(.1, .9)) %>%
    mutate(percentile = c(10,90))
# A tibble: 2 × 4
      x     y     z percentile
  <dbl> <dbl> <dbl>      <dbl>
1   5.4   1.4  -4.6         10
2   8.6  10.2  -2.2         90
```

Example: Energy Data

- Recall the wide version of the energy data:

```
energy %>% select(dayWeek, `Center_for_Mathematics_&_Computing`) %>% glimpse
Rows: 35,129
Columns: 2
$ dayWeek                                <fct> Tues, Tues, Tues, Tues, Tues, Tue…
$ `Center_for_Mathematics_&_Computing` <dbl> 14.3750, 14.0625, 14.3750, 14.062…
```

Quantiles from wide data

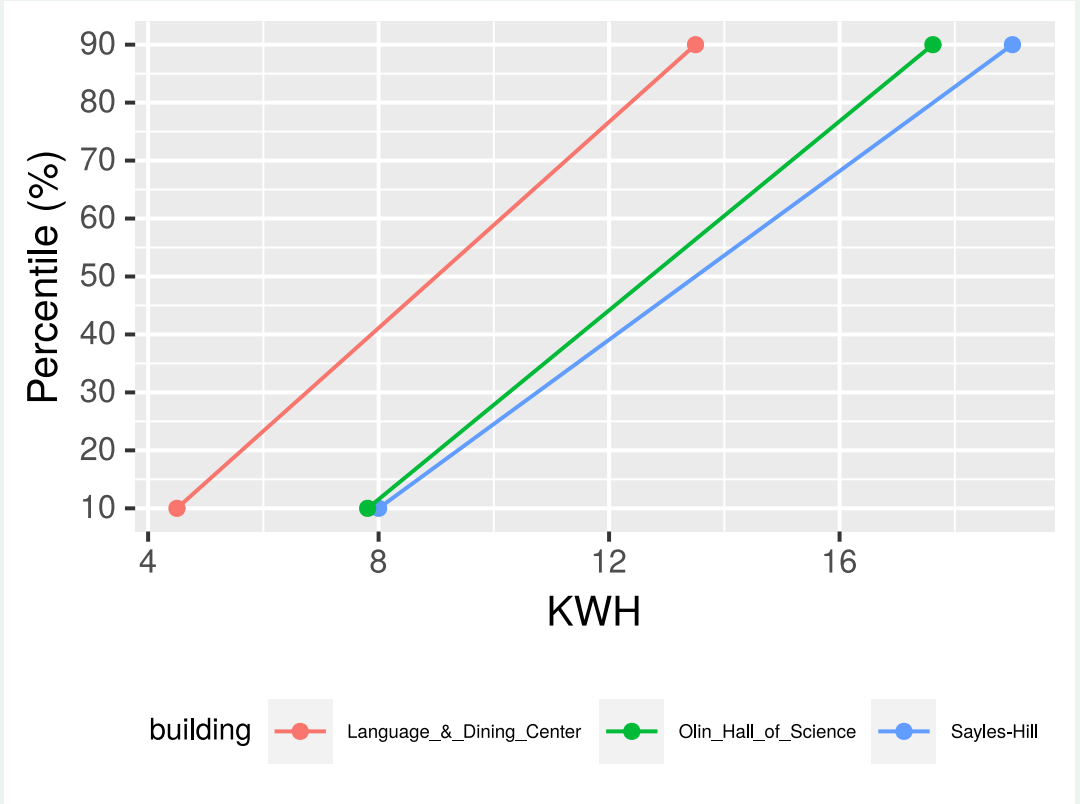
- Let's get 0.1 and 0.9 quantiles for 3 buildings:

```
energy_quant <- energy %>%
  select("Sayles-Hill" ,"Language_&_Dining_Center", "Olin_Hall_of_Science") %>%
  map_df(
    .f = quantile,
    probs = c(.1, .9),
    na.rm = TRUE,
    .id = "building")
```

```
# A tibble: 3 × 3
  building                `10%` `90%`
  <chr>                   <dbl> <dbl>
1 Sayles-Hill              8     19
2 Language_&_Dining_Center 4.5   13.5
3 Olin_Hall_of_Science     7.81  17.6
```

Plot of Quantiles

```
energy_quant   %>%
  mutate(percentile = parse_number(perc
  ggplot(aes(y = percentile, x = value
  geom_point() +
  geom_line(aes(group=building)) +
  labs(y="Percentile (%)",x="KWH") +
  scale_y_continuous(breaks=seq(10,90,b
  theme(legend.position="bottom",
        legend.text = element_text(size
        legend.title=element_text(size=
```



Quantiles from wide data

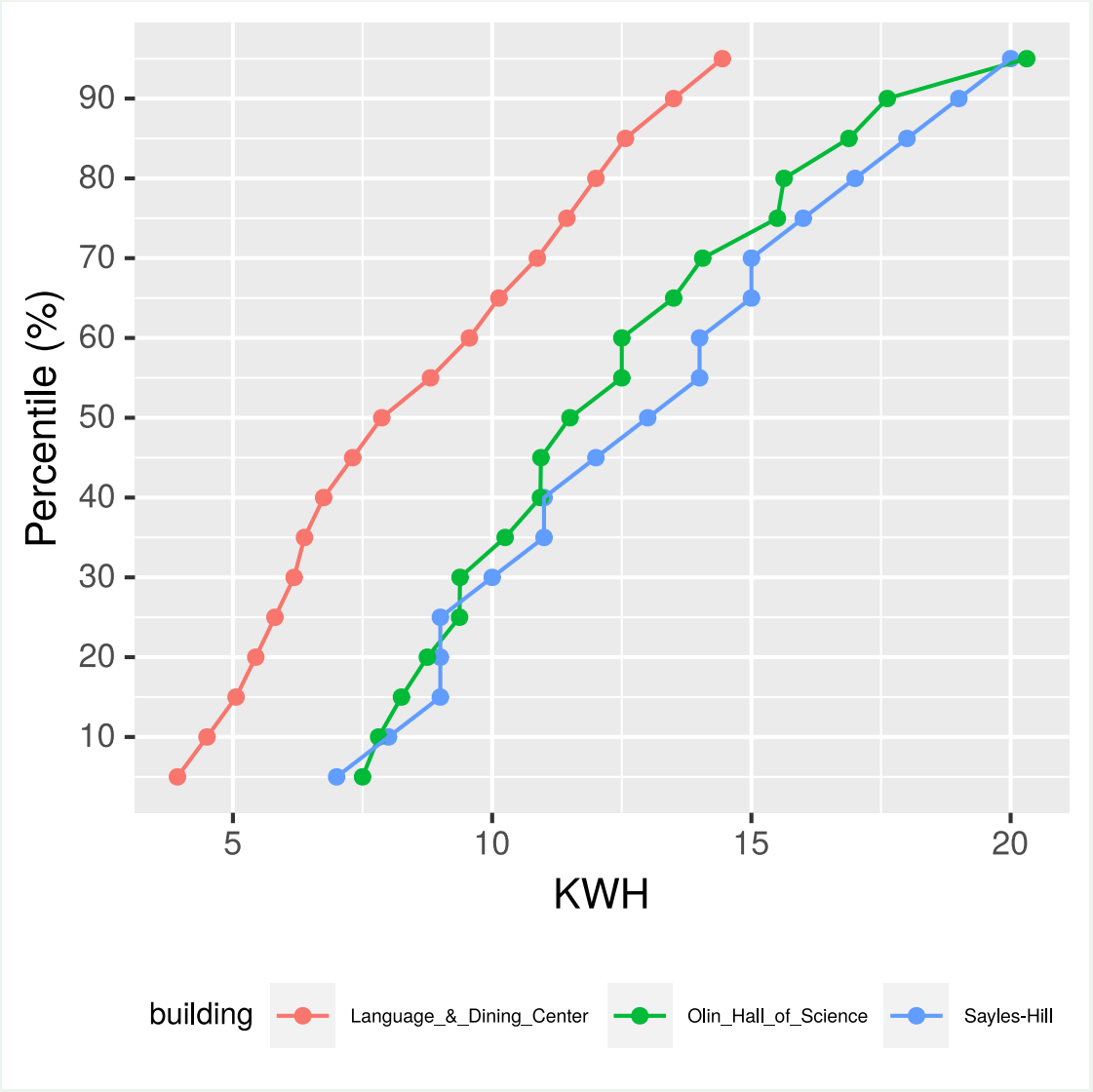
- create a vector of quantile probs:

```
p <- seq(0.05,0.95, by = .05)  # every 5th quantile
energy_quant <- energy %>%
  select("Sayles-Hill" ,"Language_&_Dining_Center", "Olin_Hall_of_Science") %>%
  map_df(
    .f = quantile,
    probs = p,
    na.rm = TRUE,
    .id = "building") %>%
  pivot_longer(
    names_to = "percentile",
    values_to = "value",
    cols = 1 + 1:length(p), # quantiles start in col 2
  )
```

Quantiles from wide data

```
energy_quant
# A tibble: 57 × 3
   building    percentile value
   <chr>       <chr>      <dbl>
 1 Sayles-Hill 5%             7
 2 Sayles-Hill 10%            8
 3 Sayles-Hill 15%            9
 4 Sayles-Hill 20%            9
 5 Sayles-Hill 25%            9
 6 Sayles-Hill 30%           10
 7 Sayles-Hill 35%           11
 8 Sayles-Hill 40%           11
 9 Sayles-Hill 45%           12
10 Sayles-Hill 50%           13
# … with 47 more rows
```

Quantiles from wide data



Quantiles from long data

- What if we have the long version of this data?

```
energy_long <- energy %>%
  pivot_longer(
    names_to = "building",
    values_to = "energyKWH",
    cols = 9:90) %>%
  filter(building %in% c("Sayles-Hill" ,
                         "Language_&_Dining_Center",
                         "Olin_Hall_of_Science") )
```

Quantiles from long data

Goal: get quantiles for every `building` and `dayWeek`

- We don’t have many columns of measurements to apply a function to
- We have groups (building and month) that we need to summarize with quantile (more than one output value)

Quantiles from long data

- Let's get quantiles for every building and day of the week:

```
energy_long_quant <- energy_long %>%
  group_by(building, dayWeek) %>%
  summarize(value = quantile(energyKWH, probs = c(.1, .9), na.rm = TRUE))
```

```
# A tibble: 42 × 3
# Groups:   building, dayWeek [21]
   building                    dayWeek value
   <chr>                       <fct>   <dbl>
 1 Language_&_Dining_Center    Mon      4.88
 2 Language_&_Dining_Center    Mon     14.1
 3 Language_&_Dining_Center    Tues     4.5
 4 Language_&_Dining_Center    Tues    13.9
```

Quantiles from long data

- Need to add a percentile

```
energy_long_quant <- energy_long_quant %>%
  mutate(percentile = c(10,90))
```

```
energy_long_quant
# A tibble: 42 × 4
# Groups:   building, dayWeek [21]
   building                  dayWeek value percentile
   <chr>                     <fct>   <dbl>      <dbl>
 1 Language_&_Dining_Center Mon      4.88          10
 2 Language_&_Dining_Center Mon     14.1           90
 3 Language_&_Dining_Center Tues     4.5           10
```

Quantiles from long data

For all quantiles in `p <- seq(0.05,0.95, by = 0.05))`

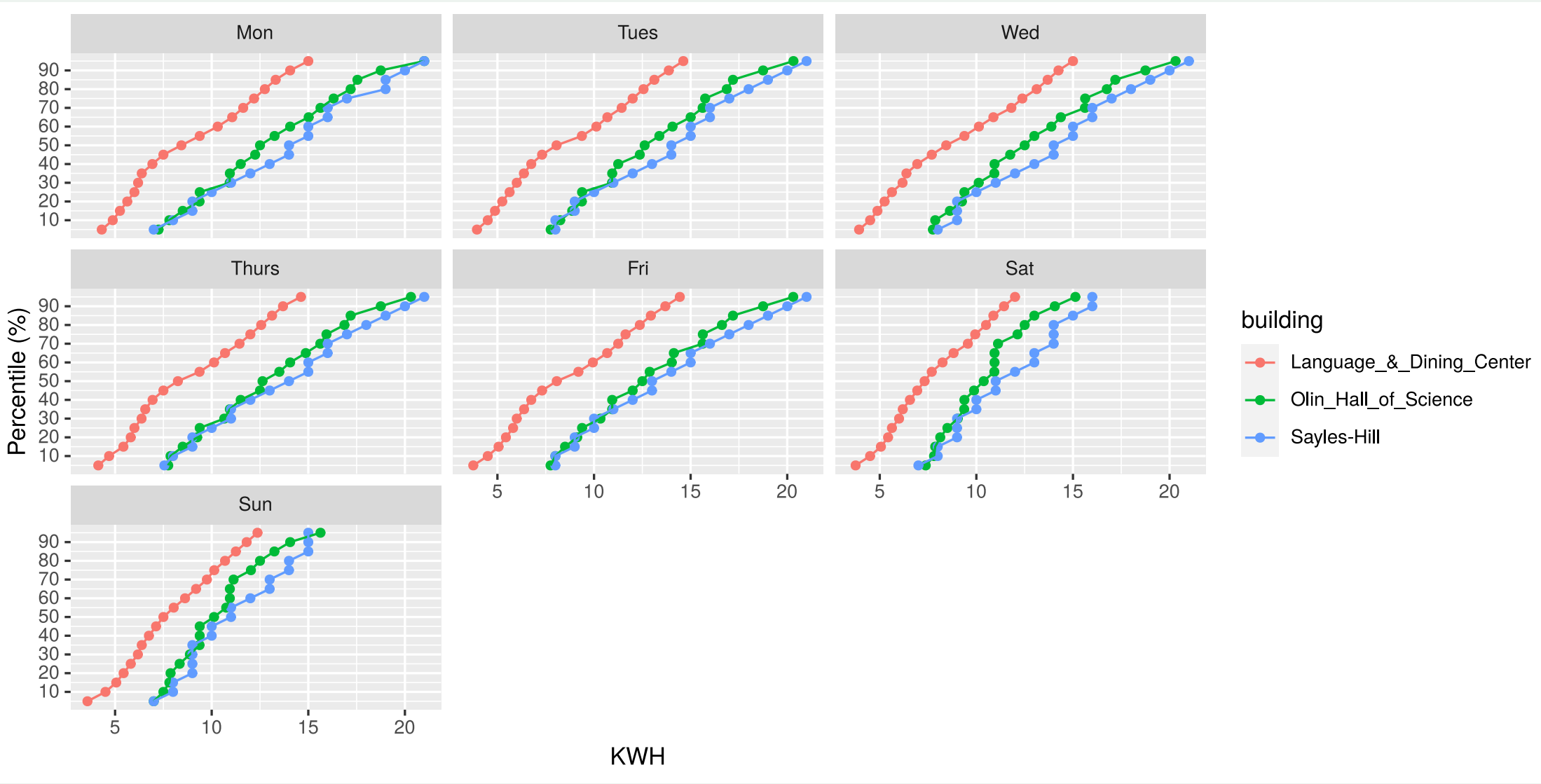
```
energy_long_quant <- energy_long %>%
  group_by(building, dayWeek) %>%
  summarize(value = quantile(energyKWH, probs = p, na.rm = TRUE))  %>%
  mutate(percentile = 100*p)
```

```
# A tibble: 399 × 4
# Groups:   building, dayWeek [21]
   building                    dayWeek value percentile
   <chr>                       <fct>   <dbl>      <dbl>
 1 Language_&_Dining_Center Mon      4.31           5
 2 Language_&_Dining_Center Mon      4.88          10
 3 Language_&_Dining_Center Mon      5.25          15
 4 Language_&_Dining_Center Mon      5.63          20
```

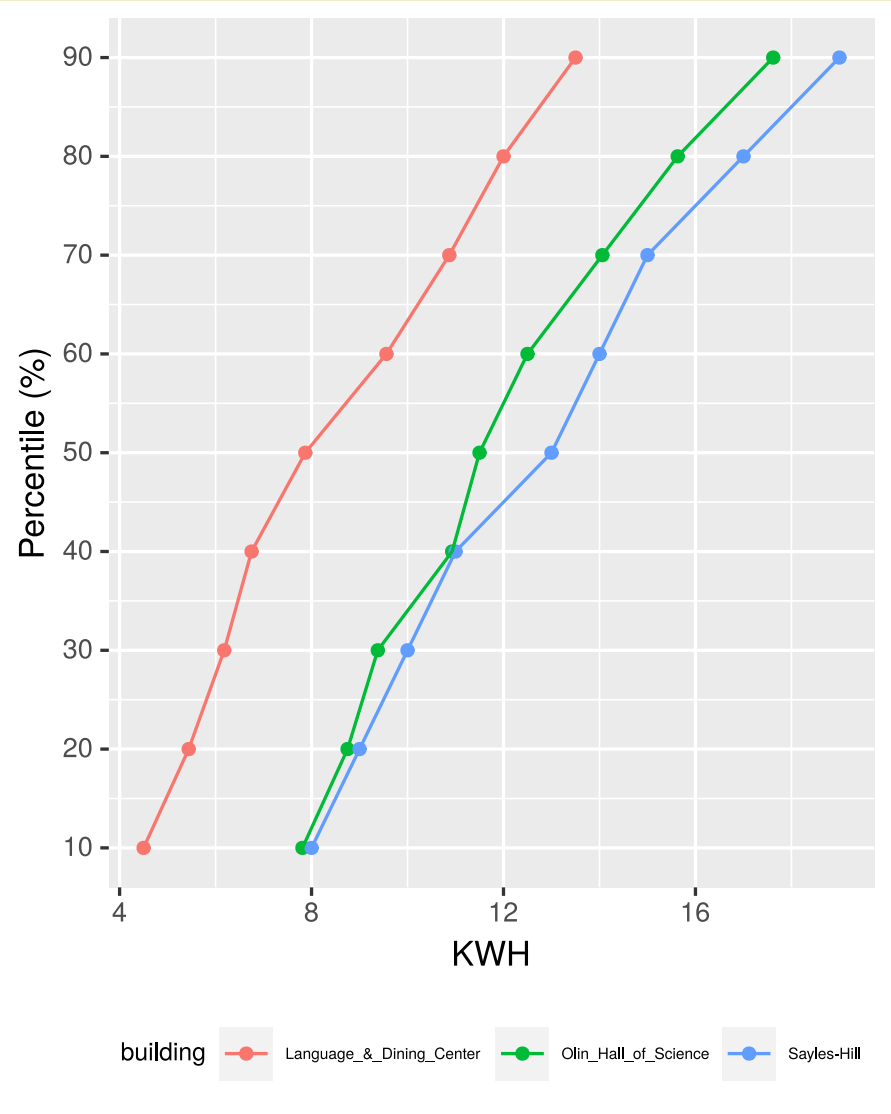
Quantiles from long data

```
energy_long_quant %>%
  ggplot(aes(y = percentile, x = value, color=building)) +
  geom_point() +
  geom_line(aes(group=building)) +
  labs(y="Percentile (%)",x="KWH") +
  scale_y_continuous(breaks=seq(10,90,by=10)) +
  facet_wrap(~dayWeek)
```

Quantiles from long data



Your Turn 2



Follow the prompts to plot the quantiles of energy consumption for the buildings `Sayles-Hill`, `Language_&_Dining_Center`, `Olin_Hall_of_Science`.

05:00