Lab 02 Data Wrangling for Dataframes

RE 519 Data Analytics and Visualization | Autumn 2025

Throughout the Lab 2, we will learn how to do data wrangling and explore two real estate datasets of King County maintained by Andy Krause at Zillow:

1. All single family and townhouse sales from 1999 through December 2023.
2. All single family and townhouse properties as of December 31, 2023.

You can find the datasets and Readme file in this repository.

The due day for each lab can be found on the course wesbite. The submission should include Rmd, html, and any other files required to rerun the codes. **For Lab 1–3, the use of any generative AI tool (ChatGPT, Copilot, etc.) is prohibited**. We will run AI detection on each submission. More information about Academic Integrity and the Use of AI.

Recommended Reading

Chapter 4-5, Modern Data Science with R. Baumer et al. 2024.

Lab 02-A: Modifying Dataframes

Let's first import the datasets and necessary packages. **If you are using a Windows computer**, you need to install RTools first. After installing this data package with `devtools::install_github('andykrause/kingCoData')`, you can load the data with: `data(kingco_sales)` and `data(kingco_homes)`. You can find the datasets and Readme file in this repository.

```
library('tidyverse')
#install.packages("devtools") # if you are using a Windows computer
#devtools::install_github('andykrause/kingCoData') # You only need to run the installation once
library('kingCoData') # load the data package
data(kingco_sales) # load the sale data
```

Have a look of this data through `View()` function or just click the dataset name in `Environment` panel, we can know there are 591,513 entries (records), 48 columns (features). Go ahead and see what features do we have for each record using `?`. Note: it only shows help if the dataset comes from a documented data package, like we did here.

```
# check all column names
colnames(kingco_sales)
```

```
##  [1] "sale_id"          "pinx"             "sale_date"        "sale_price"
##  [5] "sale_nbr"         "sale_warning"     "join_status"      "join_year"
##  [9] "latitude"         "longitude"        "area"             "city"
## [13] "zoning"           "subdivision"      "present_use"      "land_val"
## [17] "imp_val"          "year_built"       "year_reno"        "sqft_lot"
## [21] "sqft"             "sqft_1"           "sqft_fbsmt"       "grade"
```

```
## [25] "fbsmt_grade"      "condition"        "stories"          "beds"
## [29] "bath_full"        "bath_3qtr"        "bath_half"        "garb_sqft"
## [33] "gara_sqft"        "wfnt"             "golf"             "greenbelt"
## [37] "noise_traffic"    "view_rainier"     "view_olympics"    "view_cascades"
## [41] "view_territorial" "view_skyline"     "view_sound"       "view_lakewash"
## [45] "view_lakesamm"    "view_otherwater"  "view_other"       "submarket"
```

```
?kingco_sales
```

We saw a brunch of variables but let's talk about the first two:

- `sale_id`: The unique transaction identifying code. Primary key for this data! Serves as the **primary key** of the dataset — each row has a distinct `sale_id`.
- `pinx`: The unique property identifying code (Major + Minor). Serves as a **foreign key** linking sales to property records, which is `kingco_homes` in our case.

Pipe - A Unique Feature of R

One of the most distinctive features of R (especially through the tidyverse) is the pipe operator `%>%` or `|>`.

- chain commands together in a clear, left-to-right order.
- Instead of nesting multiple functions, you can read code step by step.
- closer to natural language and easier to follow in data analysis workflows.

System	Windows/Linux	Mac
Pipe Shortcut	Ctrl + Shift + M	Cmd + Shift + M

```
# example without pipe
mean(log(sqrt(c(1, 4, 9, 16))))
```

```
## [1] 0.7945135
```

```
# example with pipe
c(1, 4, 9, 16) %>%
  sqrt() %>%
  log() %>%
  mean()
```

```
## [1] 0.7945135
```

Simple Sampling from Dataframes

We can randomly select some rows from the data using `sample_n()`, every transaction has the same chance of being picked.

```
random_sample <- kingco_sales %>%
  sample_n(3)
random_sample
```

```
## # A tibble: 3 × 48
##   sale_id     pinx       sale_date  sale_price sale_nbr sale_warning join_status
##   <chr>       <chr>      <date>          <int>    <int> <chr>        <chr>
## 1 2001..35811 ..0339200… 2001-11-07     425000        1 "   "        nochg
## 2 2005..9953  ..6003501… 2005-03-22     392000        1 "   "        new
## 3 2018..764   ..2926049… 2018-01-08     656000        2 "   "        new
## # ℹ 41 more variables: join_year <dbl>, latitude <dbl>, longitude <dbl>,
## #   area <int>, city <chr>, zoning <chr>, subdivision <chr>, present_use <int>,
## #   land_val <int>, imp_val <int>, year_built <int>, year_reno <int>,
## #   sqft_lot <int>, sqft <int>, sqft_1 <int>, sqft_fbsmt <int>, grade <int>,
## #   fbsmt_grade <int>, condition <int>, stories <dbl>, beds <int>,
## #   bath_full <int>, bath_3qtr <int>, bath_half <int>, garb_sqft <int>,
## #   gara_sqft <int>, wfnt <int>, golf <dbl>, greenbelt <dbl>, …
```

The default setting is **without replacement**, which means each row can appear only once. But we can add additional augment like `sample_n(5, replace = TRUE)`, so that the same row can appear more than once.

If we run the code above several time, we will get different results! Others will not get same results as you did, the codes is NOT reproducible. So, we usually fix the random number generator using `set.seed()`.

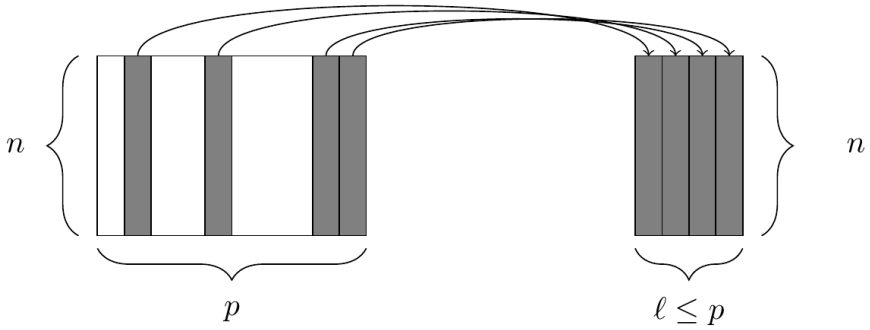
```
set.seed(413) # set seed, then running the same sampling code again will give the same rows
random_sample <- kingco_sales %>%
  sample_n(3)
random_sample
```

```
## # A tibble: 3 × 48
##   sale_id     pinx       sale_date  sale_price sale_nbr sale_warning join_status
##   <chr>       <chr>      <date>          <int>    <int> <chr>        <chr>
## 1 2014..3501  ..3575300… 2014-02-20    1300000        5 "   "        new
## 2 2007..3648  ..8089500… 2007-02-08    1000000       NA "   "        nochg
## 3 2005..44992 ..9238900… 2005-09-30     308000        1 "   "        reno - aft…
## # ℹ 41 more variables: join_year <dbl>, latitude <dbl>, longitude <dbl>,
```

```
## #   area <int>, city <chr>, zoning <chr>, subdivision <chr>, present_use <int>,
## #   land_val <int>, imp_val <int>, year_built <int>, year_reno <int>,
## #   sqft_lot <int>, sqft <int>, sqft_1 <int>, sqft_fbsmt <int>, grade <int>,
## #   fbsmt_grade <int>, condition <int>, stories <dbl>, beds <int>,
## #   bath_full <int>, bath_3qtr <int>, bath_half <int>, garb_sqft <int>,
## #   gara_sqft <int>, wfnt <int>, golf <dbl>, greenbelt <dbl>, …
```

Select Rows or Columns

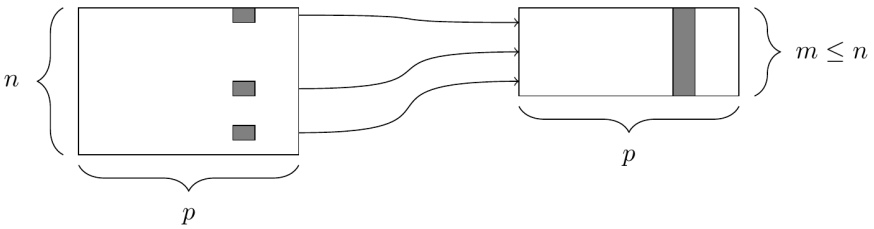
We use `select()` to select and create new dataframe with certain columns. Let's select a few columns and see whether there are some transactions for the property that can see both Lake Washington and Lake Sammammish.



The `select()` function. At left, a data frame, from which we retrieve only a few of the columns. At right, the resulting data frame after selecting those columns. Source: Modern Data Science with R.

```
# select a few useful columns
kingco_sales_subset <- kingco_sales %>%
  select(sale_id, city, sale_price, sale_date, view_lakewash, view_lakesamm)
```

We use `filter()` function to filter certain rows based on some conditions. We discussed the logical operations in Lab 1, such as `==`, `!=`, `>`, `<=`, `TRUE`, `&`, and `|`.



The `filter()` function. At left, a data frame that contains matching entries in a certain column for only a subset of the rows. At right, the

resulting data frame after filtering. Source: Modern Data Science with R.

```
kingco_sales_twolakes <- kingco_sales_subset %>%
  filter(
    view_lakewash == 1,
    view_lakesamm == 1,
    sale_date > as.Date("2020-01-01") # after year of 2020
  )
nrow(kingco_sales_twolakes)
```

```
## [1] 10
```

Interesting, there were only 10 transactions of properties with views of both Lake Washington and Lake Sammammish after the onset of the pandemic (2020). Let's look at how many of them are below 1.5 millions.

```
kingco_sales_twolakes %>%
  filter(
    sale_price < 1500000, # price below 1.5 million
  )
```

```
## # A tibble: 2 × 6
##   sale_id     city        sale_price sale_date  view_lakewash view_lakesamm
##   <chr>       <chr>            <int> <date>             <int>         <int>
## 1 2021..8432  KING COUNTY    1390000 2021-03-25             1             1
## 2 2023..21433 BELLEVUE       1468000 2023-08-07             1             1
```

If I want to know if any of them are in the BRK region (Bellevue, Redmond, Kirkland), `%in%` checks whether a value belongs to a set of values:

```
kingco_sales_twolakes %>%
  filter(
    sale_price < 1500000, # price below 1.5 million
    city %in% c('BELLEVUE', 'REDMOND', 'KIRKLAND')
  )
```

```
## # A tibble: 1 × 6
##   sale_id     city     sale_price sale_date  view_lakewash view_lakesamm
##   <chr>       <chr>         <int> <date>             <int>         <int>
## 1 2023..21433 BELLEVUE    1468000 2023-08-07             1             1
```

We can also remove column(s) by using `select(-)`, for example, removing the columns `view_lakewash` and `view_lakesamm` from dataframe `kingco_sales_twolakes`:

```
kingco_sales_twolakes_clean <- kingco_sales_twolakes %>%
  select(-c(view_lakewash, view_lakesamm))
head(kingco_sales_twolakes_clean,3)
```

```
## # A tibble: 3 × 4
##   sale_id     city        sale_price sale_date
##   <chr>       <chr>            <int> <date>
## 1 2021..8432  KING COUNTY    1390000 2021-03-25
## 2 2021..22666 KIRKLAND       2220000 2021-06-22
## 3 2024..3565  BELLEVUE       3100000 2024-03-06
```

We can combine those operation into single chunk of code. Suppose we want to know where are those two properties and some basic information of those properties.

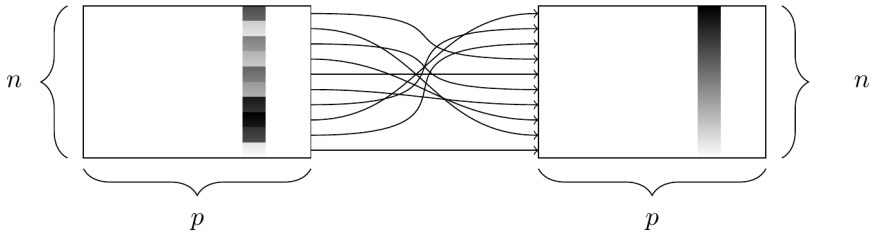
```
kingco_sales %>%
  filter(
    view_lakewash == 1,
    view_lakesamm == 1,
    sale_date >= as.Date("2020-01-01"),
    sale_price < 1500000
  ) %>%
  select(pinx, submarket, sale_price, sqft, year_built, grade)
```

```
## # A tibble: 2 × 6
##   pinx         submarket sale_price  sqft year_built grade
##   <chr>        <chr>          <int> <int>      <int> <int>
## 1 ..7526400010 R            1390000  2760       1970     9
## 2 ..9323600550 R            1468000  3340       1979     9
```

If I want to have the view to both lakes, it will cost me a lot. Sad.

Sort Based on Values

The function `sort()` will sort a vector but not a data frame. The `arrange()` function sorts a data frame.



The `arrange()` function. At left, a data frame with an ordinal variable. At right, the resulting data frame after sorting the rows in descending order of that variable. Source: Modern Data Science with R.

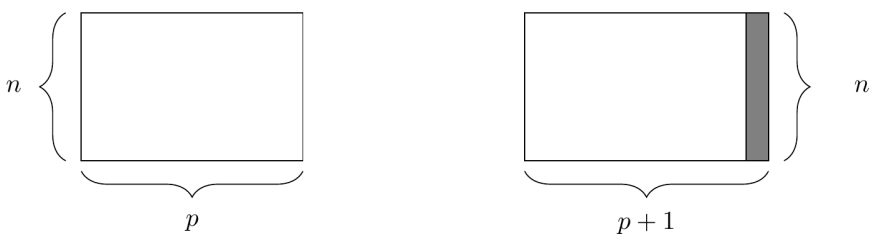
```
kingco_sales_twolakes %>%
  arrange(desc(sale_price))
```

```
## # A tibble: 10 × 6
##    sale_id     city        sale_price sale_date  view_lakewash view_lakesamm
##    <chr>       <chr>            <int> <date>             <int>         <int>
##  1 2024..27893 BELLEVUE       3550000 2024-12-06             1             1
##  2 2024..3565  BELLEVUE       3100000 2024-03-06             1             1
##  3 2024..22974 BELLEVUE       3000000 2024-10-01             1             1
##  4 2022..20951 BELLEVUE       2850000 2022-06-30             1             1
##  5 2021..22666 KIRKLAND       2220000 2021-06-22             1             1
##  6 2024..26855 BELLEVUE       2050000 2024-11-19             1             1
##  7 2020..33284 BELLEVUE       1800000 2020-10-20             1             1
##  8 2021..31882 BELLEVUE       1710000 2021-08-17             1             1
##  9 2023..21433 BELLEVUE       1468000 2023-08-07             1             1
## 10 2021..8432  KING COUNTY    1390000 2021-03-25             1             1
```

Note: We use `desc()` because the default sort order is ascending. If we want ascending order, we don't need to provide any argument.

Create, Re-define, and Rename Variables

Currently, we have the columns for view to Lake Washington and Lake Sammamish separately, we can use `mutate` to create a new 1/0 binary variable to indicate. Meanwhile, it's hard to count the number like 1390000; let's add a new variable with million dollars as the unit.



The `mutate()` function. At right, the resulting data frame after adding a new column. Source: Modern Data Science with R.

```
kingco_sales_add <- kingco_sales %>%
  mutate(
    sale_price_million = sale_price / 1e6, # 1e6 is scientific notation for 1 * 10^6 = 1,000,000
    view_likes = ifelse(view_lakewash == 1 & view_lakesamm == 1, 1, 0)
  )
```

Here we use `ifelse(test, yes, no)`, which is a vectorized conditional function in R: if the test condition is TRUE, it returns the value in yes, and vice versa.

Oops, we wrote `view_likes` as the column name for the binary variable to indicate with the view to both LAKES. We can use `rename()` function to change.

```
# `kingco_sales_add <- kingco_sales_add`: overwrite it with a modified version of itself after applying
kingco_sales_add <- kingco_sales_add %>%
  rename(view_lakewash_lakesamm = view_likes)
```

We can recode variables based on some conditions. For example, we want, in `city` of `kingco_sales_twolakes`, BRK region can be represented by `BRK` rather than separate city names. This operation is based on `recode()`. Similarly, `case_when()` can be used for more flexible recoding, creating categories based on sale price ranges.

```
kingco_sales_twolakes %>%
  mutate(
    price_category = case_when(
      sale_price < 500000   ~ "Low",
      sale_price < 1000000  ~ "Mid",
      TRUE                  ~ "High" # for others, we set as 'High'
    ),
    city_group = recode(
      city,
      "BELLEVUE" = "BRK",
      "REDMOND"  = "BRK",
      "KIRKLAND" = "BRK"
    )
  )
```

```
## # A tibble: 10 × 8
##    sale_id     city        sale_price sale_date  view_lakewash view_lakesamm
```

```
##    <chr>       <chr>            <int> <date>             <int>         <int>
##  1 2021..8432  KING COUNTY    1390000 2021-03-25             1             1
##  2 2021..22666 KIRKLAND       2220000 2021-06-22             1             1
##  3 2024..3565  BELLEVUE       3100000 2024-03-06             1             1
##  4 2024..22974 BELLEVUE       3000000 2024-10-01             1             1
##  5 2020..33284 BELLEVUE       1800000 2020-10-20             1             1
##  6 2023..21433 BELLEVUE       1468000 2023-08-07             1             1
##  7 2021..31882 BELLEVUE       1710000 2021-08-17             1             1
##  8 2022..20951 BELLEVUE       2850000 2022-06-30             1             1
##  9 2024..26855 BELLEVUE       2050000 2024-11-19             1             1
## 10 2024..27893 BELLEVUE       3550000 2024-12-06             1             1
## # ℹ 2 more variables: price_category <chr>, city_group <chr>
```

📚 TODO: Modifying Dataframes

4 points

Using the dataset `kingco_homes` and the operations we learned, especially pipe (%>%), to finish following tasks:

1. Select the columns: `pinx`, `city`, `land_val`, `imp_val`, `sqft`, `view_lakewash`, `view_lakesamm`
2. Filter the rows: all properties in Bellevue
3. Add a new variable called `total_val` which represents the sum of `land_val` (land value) and `imp_val` (improvements value)
4. Sort the dataframe based on `total_val` in descending order, and show the first 10 records using `head()`
5. Calculate the average (mean value) and variance of `total_val` for:
    - all properties in Bellevue
    - all properties with the view to both Lake Washington and Sammamish in Bellevue
6. Report the difference in average and variance

```
# TODO
```

Lab 02-B: Summarizing and Joining Dataframes

Summarizing Data

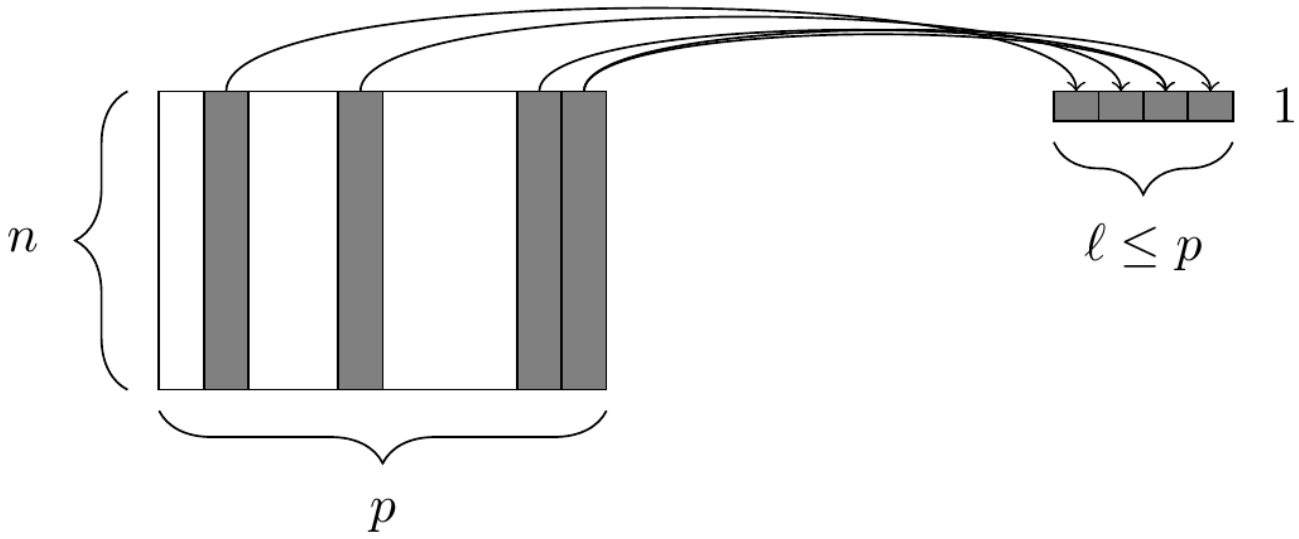
`summary()` is a base R built-in function that provides quick descriptive statistics. For dataframes, it applies `summary()` to each variable.

```
summary(kingco_sales[, c("pinx", "sale_date", "land_val")])
```

```
##      pinx             sale_date             land_val
##  Length:591513      Min.   :1999-01-01   Min.   :       0
##  Class :character   1st Qu.:2004-11-04   1st Qu.:  233000
##  Mode  :character   Median :2012-04-23   Median :  380000
##                     Mean   :2011-09-29   Mean   :  471369
##                     3rd Qu.:2018-06-25   3rd Qu.:  600000
##                     Max.   :2025-01-08   Max.   :27046000
```

You may notice that we are using `kingco_sales[, c("pinx", "sale_date", "land_val")]` instead of `select()` to get the columns. This is the base R way of extracting something from vectors, lists, or even dataframes , while `select()` comes from `dplyr`. Both return the same result, but `select()` is usually preferred in a tidyverse workflow because it is more readable and flexible. You can learn more about `[]` by looking at `?Extract`.

We always want to know more than what `summary()` can provide, `summarize()` (same as `summarise()`) is a function in `dplyr` to summarise each group down to one row. But how do we want to summarise a dataframe? Let's test with our `kingco_sales` data.



Suppose we want to explore all transactions in Seattle after Jan 1, 2020.

```
kingco_sales %>%
  filter( # find all transactions in Seattle after Jan 1, 2020 first
    city == "SEATTLE",
    sale_date >= as.Date("2020-1-1")
  ) %>%
  summarise(
    N = n(), # number of rows
    avg_sale_price = mean(sale_price), # average sale price
    highest_sale_price = max(sale_price), # max sale price
    percent_view_rainier = sum(view_rainier)/n() * 100, # percentage of those sales with a Rainier view
    unique_property = length(unique(pinx)) # the number of unique property in all transactions
  )
```

```
## # A tibble: 1 × 5
##       N avg_sale_price highest_sale_price percent_view_rainier unique_property
##   <int>          <dbl>              <int>                <dbl>           <int>
## 1 34858       1055507.           16200000                 1.45           32475
```

```
colnames(kingco_sales)
```

```
##  [1] "sale_id"          "pinx"             "sale_date"        "sale_price"
##  [5] "sale_nbr"         "sale_warning"     "join_status"      "join_year"
##  [9] "latitude"         "longitude"        "area"             "city"
## [13] "zoning"           "subdivision"      "present_use"      "land_val"
## [17] "imp_val"          "year_built"       "year_reno"        "sqft_lot"
## [21] "sqft"             "sqft_1"           "sqft_fbsmt"       "grade"
## [25] "fbsmt_grade"      "condition"        "stories"          "beds"
## [29] "bath_full"        "bath_3qtr"        "bath_half"        "garb_sqft"
## [33] "gara_sqft"        "wfnt"             "golf"             "greenbelt"
## [37] "noise_traffic"    "view_rainier"     "view_olympics"    "view_cascades"
## [41] "view_territorial" "view_skyline"     "view_sound"       "view_lakewash"
## [45] "view_lakesamm"    "view_otherwater"  "view_other"       "submarket"
```

Groupby

One important and useful function is `group_by`. We know that, before giving data to `summarise()`, we already select all transactions in Seattle. However, many times, we care about the differences across the groups, like Seattle vs. Redmond. Those city information are categorical data in the column `city`. We can easily use `group_by` to summarise by group. Now, we want to explore all transactions after Jan 1, 2020 across cities.

```
kingco_sales %>%
```

```r
  filter( # find all transactions after Jan 1, 2020 first
    sale_date >= as.Date("2020-1-1")
  ) %>%
  group_by(city) %>% # group by city and same `summarise()` will be carried out for each city
  summarise(
    N = n(),
    avg_sale_price = mean(sale_price),
    highest_sale_price = max(sale_price),
    percent_view_rainier = sum(view_rainier)/n() * 100,
    unique_property = length(unique(pinx))
  ) %>%
  arrange(desc(avg_sale_price))
```

```
## # A tibble: 41 × 6
##    city              N avg_sale_price highest_sale_price percent_view_rainier
##    <chr>         <int>          <dbl>              <int>                <dbl>
##  1 HUNTS POINT      32       6342016.           32000000                0
##  2 YARROW POINT     60       5114816.           18000000                0
##  3 MEDINA          220       4995038.           20000000                2.73
##  4 CLYDE HILL      208       4337786.           12100000                1.92
##  5 MERCER ISLAND  1313       2651048.           28500000                9.75
##  6 BEAUX ARTS       26       2585796.            4452200                0
##  7 BELLEVUE       5330       1888176.           22750000                0.994
##  8 KIRKLAND       4674       1598201.           15400000                0.984
##  9 SAMMAMISH      4124       1576538.            7000000                0.364
## 10 NEWCASTLE       712       1490214.            4180000                0.562
## # ℹ 31 more rows
## # ℹ 1 more variable: unique_property <int>
```

Joining Dataframes

We mentioned two variables are keys: `sale_id` and `pinx`. `pinx`, property ID, exists in both datasets, which serves as the **primary key** for `kingco_homes` — each row has a distinct `pinx`. Meanwhile, it is a **foreign key** in `kingco_sales`: in `kingco_sales`, same property can associate with more than one transactions.

Variable	kingco_sales	kingco_homes
sale_id	✔ (unique transaction ID, primary key)	✘ (not included)
pinx	✔ (property ID, foreign key)	✔ (property ID, primary key)

If we want to join those two dataframes, `pinx` is the only "shared knowledge" we have for both datasets. Joining them is a little bit complicated, let's use a simple example.

```r
df1 <- data.frame(
  id = c(1, 2, 3),
  name = c("Tom", "Jerry", "Spike") # Tom, Jerry, and the bulldog called Spike
)

df2 <- data.frame(
  id = c(2, 3, 4, 2),
  score = c(85, 90, 95, 87) # notice that we have two records for id = 2 (Jerry)
)


# inner join: keeps only rows where id exists in both data frames
inner_join(df1, df2, by = "id")
```

```
##   id  name score
## 1  2 Jerry    85
## 2  2 Jerry    87
## 3  3 Spike    90
```

```r
# left join: keeps all rows from df1 (left table). If no match in df2, fill with NA.
left_join(df1, df2, by = "id")
```

```
##   id  name score
## 1  1   Tom    NA
## 2  2 Jerry    85
## 3  2 Jerry    87
## 4  3 Spike    90
```

```r
# right join: keeps all rows from df2 (right table). If no match in df1, fill with NA.
right_join(df1, df2, by = "id")
```

```
##   id  name score
## 1  2 Jerry    85
## 2  2 Jerry    87
## 3  3 Spike    90
## 4  4  <NA>    95
```

```r
# full join: keeps all rows from both tables. Missing values are filled with NA.
full_join(df1, df2, by = "id")
```

```
##   id  name score
## 1  1   Tom    NA
## 2  2 Jerry    85
## 3  2 Jerry    87
## 4  3 Spike    90
## 5  4  <NA>    95
```

Joining King County Datasets

Back to our datasets, let's first check the differences between `pinx` in both datasets.

```
# check whether kingco_sales$pinx are all in kingco_homes$pinx
all(kingco_sales$pinx %in% kingco_homes$pinx)
```

```
## [1] FALSE
```

```
# check the number of `pinx` values from sales are not in homes
setdiff(kingco_sales$pinx, kingco_homes$pinx) %>%
  length()
```

```
## [1] 4368
```

If we want to focus on `kingco_sales` and use `kingco_homes` as attributes of those transactions, we can use `left_join()`. (Also `right_join()` if you put `kingco_sales` on the right.)

```
kingco_sales_w_info <- left_join(kingco_sales, kingco_homes, by = "pinx")
```

```
## Warning in left_join(kingco_sales, kingco_homes, by = "pinx"): Detected an unexpected many-to-many r
## ℹ Row 7103 of `x` matches multiple rows in `y`.
## ℹ Row 82242 of `y` matches multiple rows in `x`.
## ℹ If a many-to-many relationship is expected, set `relationship =
##   "many-to-many"` to silence this warning.
```

```
head(kingco_sales_w_info, 3)
```

```
## # A tibble: 3 × 88
##   sale_id   pinx         sale_date  sale_price sale_nbr sale_warning join_status
##   <chr>     <chr>        <date>          <int>    <int> <chr>        <chr>
## 1 1999..144 ..2734100475 1999-01-05     150000        1 "   "        demo
## 2 1999..258 ..1535200725 1999-01-05     235000        1 "   "        demo
## 3 1999..775 ..1939800005 1999-01-07     270000        1 "   "        demo
## # ℹ 81 more variables: join_year <dbl>, latitude.x <dbl>, longitude.x <dbl>,
## #   area.x <int>, city.x <chr>, zoning.x <chr>, subdivision.x <chr>,
## #   present_use.x <int>, land_val.x <int>, imp_val.x <int>, year_built.x <int>,
## #   year_reno.x <int>, sqft_lot.x <int>, sqft.x <int>, sqft_1.x <int>,
## #   sqft_fbsmt.x <int>, grade.x <int>, fbsmt_grade.x <int>, condition.x <int>,
## #   stories.x <dbl>, beds.x <int>, bath_full.x <int>, bath_3qtr.x <int>,
## #   bath_half.x <int>, garb_sqft.x <int>, gara_sqft.x <int>, wfnt.x <int>, …
```

We Have a Problem....

Oops, we received the warning from R: Detected an unexpected many-to-many relationship between `x` and `y`. We expect there are multiple `pinx` in transaction data but `pinx` should be unique in home data. That happens all the time when you work as a data analyst: all kinds of messy and unexpected data issues. Handling these challenges carefully is an essential part of real-world data analysis.

Let's investigate duplicates.

```
kingco_homes %>%
  add_count(pinx) %>%  # add the count of `pinx` as a new column
  filter(n > 1) %>%  # find duplicated properties
  head(3) # I add `head()` because otherwise all rows will be shown on the html when I knit this Rmd
```

```
##           pinx latitude longitude area        city     zoning
## 1 ..2877104110 47.68005 -122.3557   82     SEATTLE NC2-55 (M)
## 2 ..2877104110 47.68005 -122.3557   82     SEATTLE NC2-55 (M)
## 3 ..0924099001 47.57435 -121.6792   80 KING COUNTY          F
##                       subdivision present_use land_val imp_val year_built
## 1 GREEN LAKE CIRCLE RAILROAD ADD           6   376000  128900       1925
## 2 GREEN LAKE CIRCLE RAILROAD ADD           6        0       0       1925
## 3                                          2        0       0       1940
##   year_reno sqft_lot sqft sqft_1 sqft_fbsmt grade fbsmt_grade condition stories
## 1         0     3960 1780   1110          0     7           0         3     1.5
## 2         0     3960 1780   1110          0     7           0         3     1.5
## 3         0    76432 1260   1260          0     6           0         3     1.0
##   beds bath_full bath_3qtr bath_half garb_sqft gara_sqft wfnt golf greenbelt
## 1    4         1         0         0         0         0    0    0         0
## 2    4         1         0         0         0         0    0    0         0
## 3    2         1         0         0         0         0    8    0         0
##   noise_traffic view_rainier view_olympics view_cascades view_territorial
## 1             3            0             0             0                0
## 2             3            0             0             0                0
## 3             0            0             0             0                0
##   view_skyline view_sound view_lakewash view_lakesamm view_otherwater
## 1            0          0             0             0               0
## 2            0          0             0             0               0
## 3            0          0             0             0               2
##   view_other submarket n
## 1          0         B 2
## 2          0         B 2
## 3          0         N 2
```

We found `land_val` and `imp_val` are different while others are same in general. To simplify the problem, we can assume `land_val` and `imp_val` may be different while others are same. Also, we should select rows with high total values.

```
kingco_homes_clean <- kingco_homes %>%
  distinct() %>% # drop any completely duplicate rows
  group_by(pinx) %>%
  slice_max(order_by = land_val + imp_val, n = 1, with_ties = FALSE) %>% # within each group, keep just
  ungroup() # remove the grouping so the result is a normal data frame again
```

Join them again and we will not receive the warning.

```
kingco_sales_w_clean_info <- left_join(kingco_sales, kingco_homes_clean, by = "pinx")
head(kingco_sales_w_clean_info, 3)
```

```
## # A tibble: 3 × 88
##   sale_id   pinx         sale_date  sale_price sale_nbr sale_warning join_status
##   <chr>     <chr>        <date>          <int>    <int> <chr>        <chr>
## 1 1999..144 ..2734100475 1999-01-05     150000        1 "   "        demo
## 2 1999..258 ..1535200725 1999-01-05     235000        1 "   "        demo
## 3 1999..775 ..1939800005 1999-01-07     270000        1 "   "        demo
## # ℹ 81 more variables: join_year <dbl>, latitude.x <dbl>, longitude.x <dbl>,
## #   area.x <int>, city.x <chr>, zoning.x <chr>, subdivision.x <chr>,
## #   present_use.x <int>, land_val.x <int>, imp_val.x <int>, year_built.x <int>,
## #   year_reno.x <int>, sqft_lot.x <int>, sqft.x <int>, sqft_1.x <int>,
## #   sqft_fbsmt.x <int>, grade.x <int>, fbsmt_grade.x <int>, condition.x <int>,
## #   stories.x <dbl>, beds.x <int>, bath_full.x <int>, bath_3qtr.x <int>,
## #   bath_half.x <int>, garb_sqft.x <int>, gara_sqft.x <int>, wfnt.x <int>, …
```

Very Simple Git and GitHub

- Every change you make is saved in history, so you can always go back.
- If something breaks, you can roll back to a working version.
- Multiple people can work on the same project without overwriting each other.
- Your code and documents are safe, even if your computer crashes.
- Most of the data science ecosystem (R packages, Python libraries, statistical models, tutorials) lives on GitHub.

There are two main ways to use GitHub:

1. Terminal (Command Line) – more powerful and professional, gives you full control.
2. GitHub Desktop – easier for beginners, has a user-friendly interface.

We will using GitHub Desktop:

1. Download and install GitHub Desktop.
2. Sign up a free GitHub account.
3. Sign in with your GitHub account on GitHub Desktop.
4. Create a new public repository.
5. Click "Clone a repository" to copy from GitHub to your computer.
6. Add this `.Rmd` and `html` to your local repository.
7. Commit changes with a short message (e.g., "Lab 3 for RE 519").
8. Click "Push origin" to upload your changes back to GitHub.
9. You can check your changes on GitHub now.

We only covered every basic of Git and GitHub, you can learn more on the logic behind Git using Learn Git Branching.

📚 TODO: Summarizing and Joining Dataframes

2 points

Let's use the `kingco_homes_clean` data to do some summarise and analysis.

For all property in Seattle, what's the average total values per sqft [(`land_val` + `imp_val`)/`sqft`] by condition? Also, add the count of properties in each group in the summary table.

```
# TODO
```

📚 TODO: Sampling for Central Limit Theorem

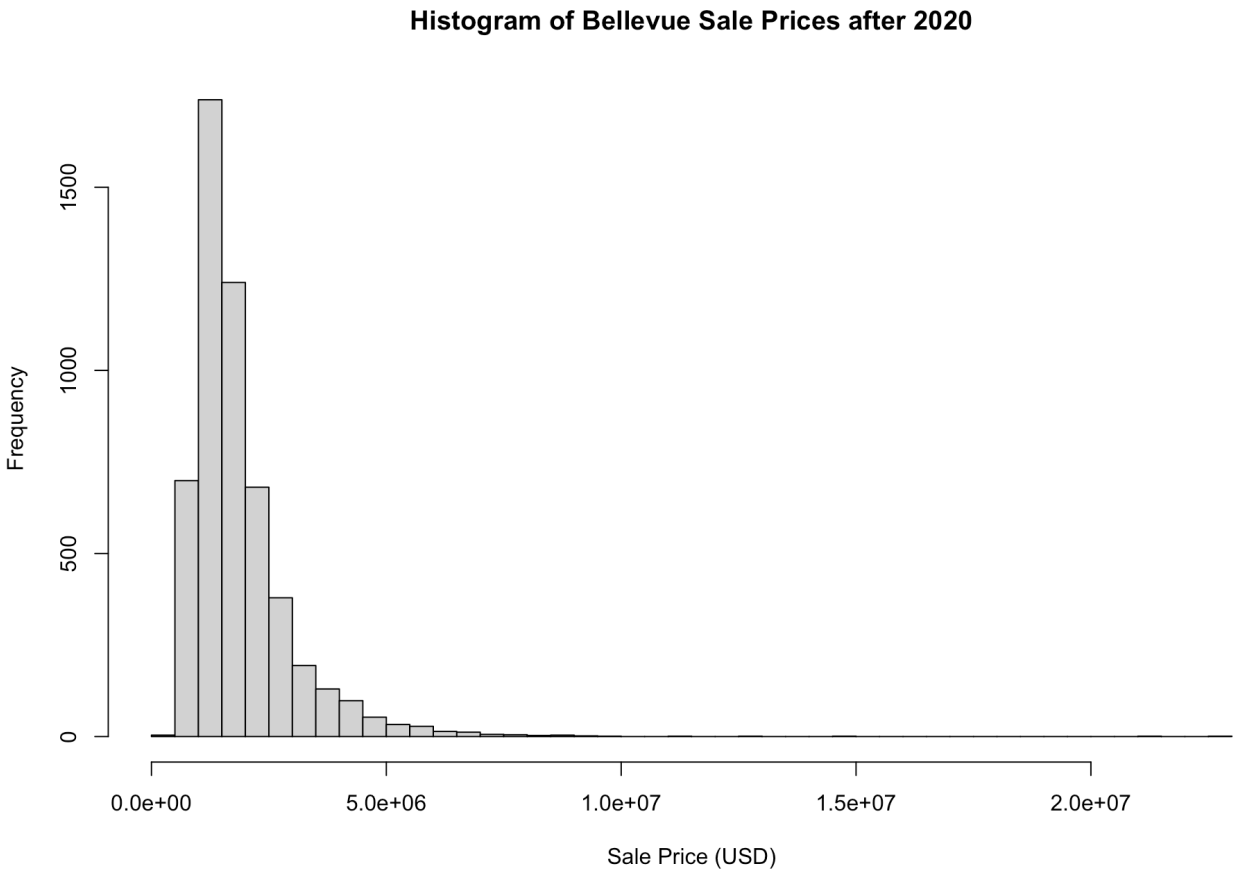
4 points

We are going to use the sale price data to understand one of the most important theorems in statistics, Central Limit Theorem (CLT). I *highly* recommend the visualization and this video from 3Blue1Brown to understand CLT.

```r
# filter transactions after 2020 in Bellevue
bellevue_after2020 <- kingco_sales %>%
  filter(
    city == "BELLEVUE",
    sale_date >= as.Date("2020-01-01")
  )
```

Let's visualize it and we can find the sale price is highly skewed (what is skewness?), not normally distributed. Right now, we are using a histogram to display this distribution (will be covered in later class).

```r
# visualize the distribution of sale prices
hist(
  bellevue_after2020$sale_price,
  breaks = 50,
  main = "Histogram of Bellevue Sale Prices after 2020",
  xlab = "Sale Price (USD)"
)
```




```r
# calculate the average of bellevue_after2020$sale_price
```

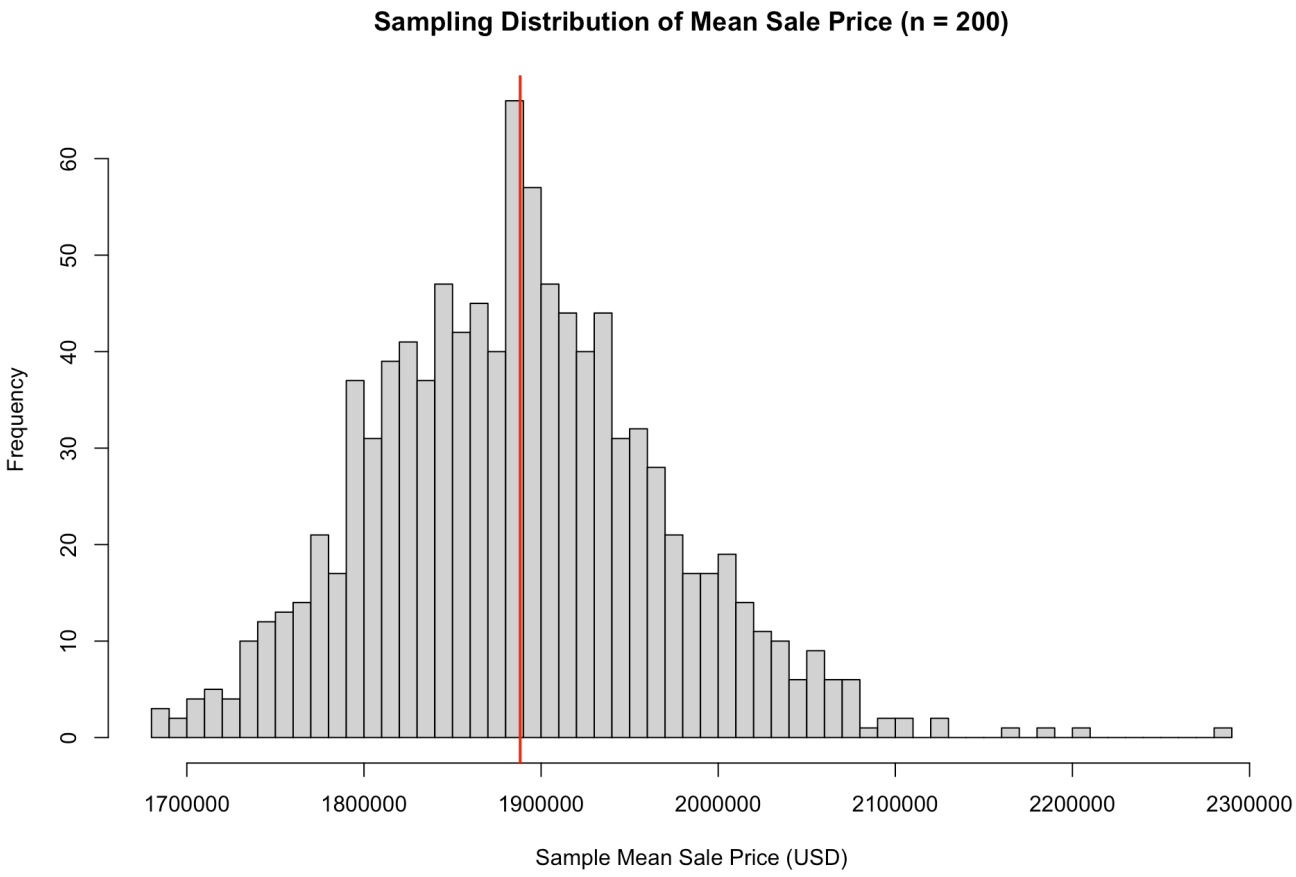
```
sale_price_mean = mean(bellevue_after2020$sale_price)
```

The mean value of sale price in Bellevue is $1,888,176. Suppose we do not know the population (all transactions), we only have samples. We can use `sample_n()` to draw a random sample from Bellevue transactions, compute the sample mean, then repeat 1,000 times.

```
# create sample means; you don't have to anything
set.seed(79)
sample_means <- replicate(
  1000,  # we repeat the process (sample -> sample mean -> record) for 1000 times
  bellevue_after2020 %>%
    sample_n(200, replace = TRUE) %>%
    summarise(avg_price = mean(sale_price)) %>%
    pull(avg_price)
)
```

Visualize the sample means with the red line as the 'real' average, we found the distribution of sample means approaches normality.

```
hist(
  sample_means,
  breaks = 50,
  main = "Sampling Distribution of Mean Sale Price (n = 200)",
  xlab = "Sample Mean Sale Price (USD)"
)
abline(v = sale_price_mean, col = "red", lwd = 2)
```



Repeat the process with smaller sample size, such as `n = 10`, and smaller repeating time, such as 200. What did you find? What interpretation can you give for the central limit theorem from your simulation results?

```
# TODO
```

📚 TODO: Explore More

4 points

So far, we've learned a lots functions, please use as many as possible to explore topics you are interested in about the King County datasets (or any other datasets you are interested iin). Please use at least 5 functions below and brief discuss your results.

- `select()` – choose columns
- `filter()` – filter rows by conditions
- `%in%` – check membership in a set
- `arrange()` + `desc()` – sort data

- `mutate()` – create or redefine variables
- `rename()` – change column names
- `recode()` and `case_when()` – recode variables and create categories
- `sample_n()` + `set.seed()` – random sampling
- `summary()` – base R quick statistics
- `summarise()` / `summarize()` – compute group summaries
- `group_by()` – aggregate by categories
- joins (`inner_join`, `left_join`, `right_join`, `full_join`) – combine datasets
- `distinct()` and `slice_max()` – handle duplicates

```
# TODO
```

📚 TODO: Using GitHub

2 points

After completing this lab, commit your changes with a short message, and then push them to your GitHub repository. Finally, copy the repository link from GitHub and submit it on the Canvas assignment page. Make sure that your repository is publicly visible.

Acknowledgement

The materials are developed by Haoyu Yue based materials from Dr. Feiyang Sun at UC San Diego, Siman Ning and Christian Phillips at University of Washington, Dr. Charles Lanfear at University of Cambridge.