dplyr

Making data manipulation easy

Overview: SQL but BETTER!

- Helps you manipulate your data in a variety of ways.

- Similar to SQL, dplyr relies on action functions
  - ***select*** variables of interest
  - ***join*** different datasets together
  - create new variables via ***mutate***
  - ***filter*** your data to a specific subset
  - ***summarise*** your data over specific partitions or ***groups***

- Unlike SQL these functions can be used in any order to best suit your project. Sounds awesome, right?

A brief intro on pipes

- A pipe is something that helps you flow from one function to another
- It tells R to take the data from the left side of the pipe and apply the function occurring on the right side of the pipe
- The most commonly pipe is: %>%

- Why should I use this?
  - So you don't need to create multiple temporary or sub-datasets
  - Instead you can make just one dataset!

dplyr::filter

- Want to limit your data to a specific subset? Yeah I do!
- Similar to SQL's 'where' and 'having' statements
- Can filter based on character or numeric variables using inequalities or other useful functions in R.

dplyr::filter

- Common filtering functions:
  1) var_name == 1 (if numeric)
  2) var_name == 'b' (if character)
  3) var_name != 1 (if numeric)
  4) var_name != 'b' (if character)
  5) var_name > 1
  6) var_name < 1
  7) var_name %in% c('a','b','c')

- Combine filtering functions
  - AND criteria (“&” or separating arguments using a comma “,”)
  - OR criteria (“|”)

dplyr::filter sample code

```r
# filter dataframe to only IV administered drugs
med_order %>% filter(ROUTE == 'IV')

# filter dataframe where medications were administered and patient's age is less than 2 years old
med_ord %>%  filter(ADMIN_IND != 0 & age_yr < 2.0)

# filter dataframe where patient is in 3 EAST or 3 WEST
patient %>%  filter(DEPT_NM %in% c('3 EAST', '3 WEST'))
```

dplyr::filter exercise

- Filter blood culture data to only look at blood cultures ordered in the NICU and store as a new dataframe

dplyr::select

- Similar to SQL's select statement
- Choose variables you wish to keep in your dataset
- CANNOT create new variables in this function
- BONUS FACT: you can rename variables in this statement if you'd like

```
dplyr::select(new_var_name = old_var_name)
```

dplyr::select variables

- Option 1: Write out each variable you'd like to keep
- Option 2: Write out the variables you'd like to discard prefixed by a "-" sign
  - Most useful when it's easier to drop one or two variables instead of calling all remaining variables in a dataset

```
dplyr::select(-useless_variable)
```

vs.

```
dplyr::select( useful_var1, useful_var2, useful_var3, useful_var4, useful_var5)
```

dplyr::select sample code

```
#Select patient MRN, patient name and date of birth, rename full_name field by pat_name
patient %>%  select(pat_mrn_id,pat_name = full_name,dob)

#Select everything from patient table except for upd_dt and upd_by
patient %>%  select(-upd_dt,-upd_by)
```

dplyr::select exercise prompts

- Limit the blood culture data set to only patient information, blood culture type, blood culture department, and blood culture date and store as a new dataframe

dplyr::join

- Do you want to join your data frame with another data frame? Yeah I do!

- Similar to SQL, we use join functions to combine different data frames

dplyr::join

- The various ways to join data frames are as follows:

1) left_join(): Returns all rows from x and only rows from y where there are matching values in x. If there are multiple matches for a value in x, it will return all the rows that match.

2) right_join(): Returns all rows from y and only rows from x where there are matching values in y. If there are multiple matches for a value in y, it will return all the rows that match.

3) full_join(): Returns all rows from both x and y regardless if there are matching values in y.

4) inner_join(): Returns all rows from x and y only where there are matching values for x in y.

5) anti_join(): All rows in x that do not have a match in y

dplyr::other ways to combine data frames

- union(): Stacking data frames. Rows that appear in both data frames won't be duplicated.

- bind_rows(): Stacking data frames. Rows that appear in both data frames will be duplicated.

dplyr::join sample code

```
#Join visit table with patient table on pat_key
visit %>% inner_join(patient, by = "pat_key")
#Join by two columns like visit_key and pat_lda_key if the joining columns are called the same name in two tables
visit %>% left_join(pat_lda, by = c("visit_key","pat_lda_key"))
#Join by a column called differently in two tables
visit %>% inner_join(med_ord, by = c("visit_key" = "vis_key"))
```

dplyr::arrange

- dplyr::arrange(x): order rows by values of x low to high
- dplyr::arrange(desc(x)): order rows by values of x high to low

dplyr::mutate

- Want to create an additional column in your dataset? Yeah I do!
- In SQL, you use the select statement, but in dplyr you use the *mutate* function
- Create new variables based on already existing variables in your dataframe

dplyr::mutate

- Within the *mutate* function, you can perform the following types of operations:

1. Conditional statements using *ifelse* (R's counterpart to SQL's "case when" statement)
2. Mathematical calculations
3. Variable formatting
4. Other types of indicators (indicators that rely on group_by)

dplyr::mutate

- Conditional statements: creating a new variable based on a value of an existing variable
- ifelse

```
dplyr::mutate(new_var_low_ind = ifelse(old_var < 10, 1, 0))
```

- case_when (note: TRUE ~ 0 is the same as saying "else=0")

```
dplyr::mutate(new_var_low_ind = case_when(old_var < 10 ~ 1, TRUE ~ 0))
```

  - case_when is most useful when you have multiple conditions

dplyr::mutate

- Conditional statements
- Mathematical calculations

```
dplyr::mutate(num_risk_factors = risk_factor1_ind + risk_factor2_ind + risk_factor3_ind)
```

```
dplyr::mutate(height_inches_total = height_ft*12 + height_inches)
```

dplyr::mutate sample code

```r
# group patient by different age ranges
patient %>% mutate(age_group = case_when( age<2 ~ '0-2',
                                          age<10 ~ '2-10',
                                          age <18 ~ '10-18',
                                          TRUE ~ '18+'))
# convert patiet birth weight from oz to g
patient %>% mutate(ped_birth_wt_in_g = ped_birth_wt_in_oz * 28.34)

# lable pennsylvania patients as 1
patient %>% mutate(penn_ind = ifelse(state == 'PENNSYLVANIA',1,0))
```

dplyr::mutate

- Conditional statements
- Mathematical calculations
- Variable formatting

```
dplyr::mutate(dept_factor = as.factor(dept_char_nm))
```

```
dplyr::mutate(indicator1 = as.numeric(ind_from_sql_chunk))
```

dplyr::mutate

- Other types of indicators (indicators that rely on group_by)
  - Can use group by to apply functions over that specified group
  - Similar to using over partition by

```
dplyr::group_by(dept) %>%
  dplyr::mutate(row_num_by_dept = row_number())
```

```
dplyr::group_by(visit_key) %>%
  dplyr::mutate(max_oxygen_per_pat = max(oxygen_level))
```

dplyr::mutate window functions used with group_by

Use this if you want to see how a row relates to the other ones around it (such as, difference between subsequent lab values)

- lead: similar to SQL, the value one before
- lag: similar to SQL, the value one after
- dense_rank: ranks with no gaps
- row_number: similar to SQL, number of the row
- cummean: cumulative mean
- cumsum: cumulative sum
- And more!

dplyr::mutate sample code

```
#find the last visit date for each patient
visit %>% group_by(pat_key) %>%
  mutate(last_visit_dt = max(hosp_admit_dt))
```

dplyr::mutate exercise prompts

Create new variables for each and save in a new dataframe:

- Convert blood culture date and date of birth into date format
- Calculate patient age when the blood culture was done
- Flag patients who were under 90 days when the blood culture was done
- Label the order that blood cultures were drawn for each patient (hint: you might need to use “arrange” to specify the order that the blood cultures are in)

dplyr::summarise

- summarise function is used to roll up your data into level specified by the group_by function
- Useful for exploring distributions and descriptive statistics in your data
- Often used to ready your dataframe for visualizations

dplyr::summarise

- Useful functions are:
  - nth: Nth value in the column
  - n: count of number of rows
  - n_distinct: count of distinct values
  - min: minimum value in the column
  - max: maximum value in the column
  - mean: average value of the column
  - median: median value of the column
  - var: variance of the column
  - sd: standard deviation of the column

dplyr::summarise sample code

```
# number of medication orders per visit
med_ord %>%  group_by(VISIT_KEY) %>%
  summarise(num_ord = n())

# average LOS of different units
visit %>% group_by(DEPT_ID) %>%
  summarise(avg_los = mean(los))

# Bonus: conditional summarise LOS for flag equals to 1
visit %>% group_by(DEPT_ID) %>%
  summarize_each_(avg_los = mean(los[flag == 1]))
```

dplyr::summarise exercise prompts

- In the blood culture data, determine the count for each line type (central vs non-central line) and store as a new dataframe

dplyr::using pipes

- Take the code you created in the last 7 exercises and use pipes to perform all the actions in one step.
  - Reminder of the exercises:

1. Filter blood culture data to only look at blood cultures ordered in the NICU
2. Limit the blood culture data set to only patient information, blood culture type, blood culture department, and blood culture date

Create new variables for each:

3. Convert blood culture date and date of birth into date format
4. Calculate patient age when the blood culture was done
5. Flag patients who were under 90 days when the blood culture was done
6. Label the order that blood cultures were drawn for each patient (hint: you might need to use “arrange” to specify the order that the blood cultures are in

7. In the blood culture data, determine the count for each line type (central vs non-central line)