# Summer R

## Session 2: Data frames

Dominic Bordelon, Research Data Librarian, ULS

# Agenda

1. Introductions and review
2. R Projects
3. The pipe operator
4. Selecting and ordering df variables
5. Sorting rows in a data frame
6. Filtering (subsetting) rows
7. Summarizing and grouping a data frame
8. Base R plotting

# The dplyr package

“dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges.”

# Introductions

# ICA and Homework review

# R Projects (.RProj files)

# R Projects

- A file (`.RProj`) which lives in the root of a project directory
- RStudio uses this file to:
  - set working directory
  - remember command history (console + History pane)
  - remember project-specific option settings (Tools menu)
- It's recommended to create an RProject for each of your projects that use RStudio, to help keep your work organized.
- File > New Project…

# The pipe operator

# Pipe operator, %>%

- A tidyverse feature (magrittr package)
- Written as: `%>%` (percent greater-than percent)
- `exprA %>% exprB` evaluates expression A, and then sends its output to expression B as input

```
my_values <- c(1.33, 1.66, 2.33)

mean(my_values) %>% round(1)

# is the same as:

round(mean(my_values), 1)
```

Note that the first argument of `round()` has disappeared in the piped version, because it is filled by the mean just calculated. `1` is the `digits` argument, i.e., one decimal place.

# Pipelines

We can string together as many piped expressions as we want. For example, this code calculates temperature changes from three experimental trials, averages and rounds them, then prints a short statement:

```
t_initial <- c(25.04, 24.88, 25.23)
t_final <- c(35.82, 35.88, 35.67)

(t_final - t_initial) %>%
  mean() %>%
  signif(4) %>%
  paste("°C avg. ΔT")
```

```
[1] "10.74 °C avg. ΔT"
```

# Pipeline object assignment

Like other expressions, a pipeline can have its result assigned to an object.

Revising the same example: the value is calculated and assigned in its own pipe (as `delta_t_avg`), then combined with text afterwards.

```
delta_t_avg <- (t_final - t_initial) %>%
  mean() %>%
  signif(4)

paste(delta_t_avg, "°C avg. ΔT")
```

```
[1] "10.74 °C avg. ΔT"
```

# Benefits of the pipe

1. Avoid function wrapping (which is hard to read)
2. Avoid storing too many intermediate results in the environment (using object assignment)
3. The pipeline is easy to read as a procedure
4. Pipelines are easy to modify, e.g., to add new intermediate calls, or to cut them short when a problem has appeared.

# Keyboard shortcuts revisited

| Function | Windows | macOS |
|---|---|---|
| Execute line | Ctrl-Enter | ⌘-Enter |
| Assignment operator `<-` | Alt - (Alt-hyphen) | ⌥ - (Option-hyphen) |
| Pipe operator `%>%` | Ctrl-Shift-M | ⌘-Shift-M |

# Selecting and ordering df variables

# Step 0: load data

```
1 polyps <- read_csv("data/polyps.csv")
2 names(polyps)
```

```
[1] "participant_id" "sex"            "age"            "baseline"
[5] "treatment"      "number3m"       "number12m"
```

# Selecting variables (columns) with `dplyr::select()`

- We often only want a small number of the data frame's variables when investigating or visualizing a question
- `select(df, ...)` returns `df` with only the columns listed in `...`, one column or column-expression per argument

```
1  polyps %>%
2    select(participant_id:baseline, age)
```

```
# A tibble: 22 × 4
   participant_id sex      age baseline
   <chr>          <chr>  <dbl>    <dbl>
 1 001            female    17        7
 2 002            female    20       77
 3 003            male      16        7
 4 004            female    18        5
 5 005            male      22       23
 6 006            female    13       35
 7 007            female    23       11
 8 008            male      34       12
 9 009            male      50        7
10 010            male      19      318
# … with 12 more rows
```

# `select()` notes

- Indicate ranges with `:` (colon)
  - `age:baseline` is columns age through baseline (inclusive)
- Numeric indices may be used instead of names
- Columns in the result will follow whatever order you use with `select()`, i.e., you can rearrange at the same time that you select
- A minus sign (`-`) prepended to a column name *negates* or *excludes* it ("all columns except x")

# `select()` example

```
1 # select cols participant_id thru treatment except age, and col. no. 7
2
3 polyps %>%
4   select(participant_id:treatment, -age, 7)
```

```
# A tibble: 22 × 5
   participant_id sex    baseline treatment number12m
   <chr>          <chr>     <dbl> <chr>          <dbl>
 1 001            female        7 sulindac          NA
 2 002            female       77 placebo          63
 3 003            male          7 sulindac          2
 4 004            female        5 placebo          28
 5 005            male         23 sulindac         17
 6 006            female       35 placebo          61
 7 007            female       11 sulindac          1
 8 008            male         12 placebo           7
 9 009            male          7 placebo          15
10 010            male        318 placebo          44
# … with 12 more rows
```

# Sorting rows in a data frame

# Sorting df rows with `dplyr::arrange()`

- Often we want to sort our data frame by values in one or more columns, e.g., alphabetically by name, or (more realistically) by treatment group and by outcome variable
- `arrange(df, ...)` returns `df` with the rows reordered according to arguments (which are variable names)

```
1  polyps %>%
2    arrange(baseline) %>% print(n=7)
```

```
# A tibble: 22 × 7
  participant_id sex      age baseline treatment number3m number12m
  <chr>          <chr>  <dbl>    <dbl> <chr>        <dbl>     <dbl>
1 004            female    18        5 placebo          5        28
2 001            female    17        7 sulindac         6        NA
3 003            male      16        7 sulindac         4         2
4 009            male      50        7 placebo          7        15
5 012            female    23        8 sulindac         1         3
6 020            female    23       10 sulindac         6         3
7 007            female    23       11 sulindac         6         1
# … with 15 more rows
```

# `arrange()` notes

- The function `desc()` ("descending") can be wrapped around a variable to reverse it
- You can combine sort-variables sequentially, by giving more arguments: `arrange(colA, colB)` will sort by `colA`, and then sort by `colB` after that (as a tie-breaker).
- Each argument can be an expression if needed (column name will usually be good enough)

# `arrange()` example

```
1 polyps %>%
2   arrange(treatment, sex, desc(age))
```

```
# A tibble: 22 × 7
   participant_id sex      age baseline treatment number3m number12m
   <chr>          <chr>  <dbl>    <dbl> <chr>         <dbl>     <dbl>
 1 017            female    22       54 placebo          45        46
 2 002            female    20       77 placebo          67        63
 3 004            female    18        5 placebo           5        28
 4 006            female    13       35 placebo          31        61
 5 009            male      50        7 placebo           7        15
 6 008            male      34       12 placebo          20         7
 7 019            male      34       30 placebo          30        50
 8 014            male      30       11 placebo          20        10
 9 015            male      27       24 placebo          26        40
10 013            male      22       20 placebo          16        28
# … with 12 more rows
```

# ICA 2.1: Pipes, selecting, and sorting

# Filtering (subsetting) rows

# Step 0: logical expressions

- `logical` is one of R's basic data types, for representing `TRUE` and `FALSE`; expressions which produce a logical value are *logical expressions*
- Fundamentally, `TRUE`/`FALSE` are needed for application if-else logic
- Numeric comparators `> >= == != <= <` all produce logical results
- There are also functions which return logical values (we'll see some later)
- logical AKA Boolean or conditional

# Logical expression examples

```
t_final
```

```
[1] 35.82 35.88 35.67
```

```
t_final > 32
```

```
[1] TRUE TRUE TRUE
```

```
t_final > 35.84
```

```
[1] FALSE  TRUE FALSE
```

```
t_final <= 35.82
```

```
[1]  TRUE FALSE  TRUE
```

```
t_final == 35.67  # equals
```

```
[1] FALSE FALSE  TRUE
```

```
t_final != 35.67  # does not equal
```

```
[1]  TRUE  TRUE FALSE
```

# Filtering rows with `dplyr::filter()`

- We often need to exclude certain rows from our data, or isolate only certain rows of interest, depending on values in the data: for example, only one gender, or exclude patients whose BMI is outside the range of study, etc.
- `filter(df, logical_expr)` returns `df` but only with rows for which `logical_expr` evaluates as `TRUE`

```
1 polyps %>%
2   filter(age > 24)
```

```
# A tibble: 6 × 7
  participant_id sex     age baseline treatment number3m number12m
  <chr>          <chr> <dbl>    <dbl> <chr>        <dbl>     <dbl>
1 008            male     34       12 placebo         20         7
2 009            male     50        7 placebo          7        15
3 014            male     30       11 placebo         20        10
4 015            male     27       24 placebo         26        40
5 019            male     34       30 placebo         30        50
6 022            male     42       12 sulindac         8         4
```

# `filter()` notes

- String comparison uses `==` or functions (e.g., “name contains X”)
- The expression may contain multiple variables/columns, but it is evaluated only once for each row in the source df
- Combining multiple expressions:
  - `&`, the AND operator \(\rightarrow\) true iff both A and B are true
  - `|`, the OR operator \(\rightarrow\) true if A or B is true (or both)
  - `xor()` is the exclusive OR \(\rightarrow\) true iff A or B is true (but not both)
- Any logical expression may be negated/reversed by prepending with `!`

# `filter()` example

```
1 # retain non-placebo patients aged 18 and older:
2
3 polyps %>%
4   filter(treatment != "placebo", age >= 18)
```

```
# A tibble: 7 × 7
  participant_id sex      age baseline treatment number3m number12m
  <chr>          <chr>  <dbl>    <dbl> <chr>         <dbl>     <dbl>
1 005            male      22       23 sulindac         16        17
2 007            female    23       11 sulindac          6         1
3 012            female    23        8 sulindac          1         3
4 016            male      23       34 sulindac         27        33
5 020            female    23       10 sulindac          6         3
6 021            female    22       20 sulindac          5         1
7 022            male      42       12 sulindac          8         4
```

# Summarizing and grouping a data frame

# Base R `summary()`

- `summary()` is a base R function that prints a summary of an object; the exact format depends on the object and data type
  - numeric vector: five-number summary (very convenient!)
  - factor, i.e. categorical vector: count of cases in each category
  - dataframe: `summary()` of each column
- Try out `summary()` on new objects as you come across them, in addition to printing them on the console.
- `summary()` is very quick and easy, but its results aren't in a convenient format for further calculation/reporting. Its output is also fixed.

# Summarizing a data frame with `dplyr::summarize()`

- Often we want to aggregate or simplify multiple rows of information into one row, e.g., mean of a variable for all observations.
- `dplyr::summarize()` applies aggregating functions to variables and returns a data frame.

```
1  polyps %>%
2    summarize(mean(age), sd(age))
```

```
# A tibble: 1 × 2
  `mean(age)` `sd(age)`
        <dbl>     <dbl>
1        24.1      9.12
```

# `summarize()` notes

- Aggregation functions
  - center: `mean()`, `median()`
  - spread: `sd()`, `IQR()`
  - range: `min()`, `max()`
  - position: `first()`, `last()`, `nth()`
  - count: `n()`, `n_distinct()`
  - logical: `any()`, `all()`
- To name the columns in the output data frame, use named arguments.
- Usually we will summarize after grouping (next section)

# `summarize()` example

```
# summarize the table's number of patients,
# avg age, avg baseline, and max baseline:

polyps %>%
  summarize(row_count = n(),
            avg_age = round(mean(age), 1),
            avg_baseline = round(mean(baseline), 1),
            max_baseline = max(baseline))
```

```
# A tibble: 1 × 4
  row_count avg_age avg_baseline max_baseline
      <int>   <dbl>        <dbl>        <dbl>
1        22    24.1           41          318
```

# Grouping data frame rows with `dplyr::group_by()`

- We often want to identify and examine subgroups within the data using categorical variables.
- `group_by(df, variable)` returns a data frame with each row grouped according to its value for `variable`; this grouped df is then used with other dplyr functions (especially `summarize()`)

```
1  polyps %>%
2    group_by(treatment) %>% print(n=7)
```

```
# A tibble: 22 × 7
# Groups:   treatment [2]
  participant_id sex      age baseline treatment number3m number12m
  <chr>          <chr>  <dbl>    <dbl> <chr>        <dbl>     <dbl>
1 001            female    17        7 sulindac         6        NA
2 002            female    20       77 placebo        67        63
3 003            male      16        7 sulindac         4         2
4 004            female    18        5 placebo          5        28
5 005            male      22       23 sulindac        16        17
6 006            female    13       35 placebo        31        61
7 007            female    23       11 sulindac         6         1
# … with 15 more rows
```

# `group_by()` notes

- More than one grouping is possible (and sequence matters)
- If you want to group e.g. by numeric ranges/bins, you'll first create a categorical variable using the numeric variable as a basis (we will do this in session 4 or 5)
- Also compatible with other dplyr functions: `filter()`, `arrange()`, etc.

# group_by() example

```r
# group by treatment and sex,
# then calculate avg baseline and avg 3-month polyp count

polyps %>%
  group_by(treatment, sex) %>%
  summarize(avg_baseline = round(mean(baseline), 1),
            avg_number3m = round(mean(number3m), 1))
```

```
# A tibble: 4 × 4
# Groups:   treatment [2]
  treatment sex    avg_baseline avg_number3m
  <chr>     <chr>         <dbl>        <dbl>
1 placebo   female         42.8         37
2 placebo   male           60.3         66.6
3 sulindac  female         11.2          4.8
4 sulindac  male           42           34.5
```

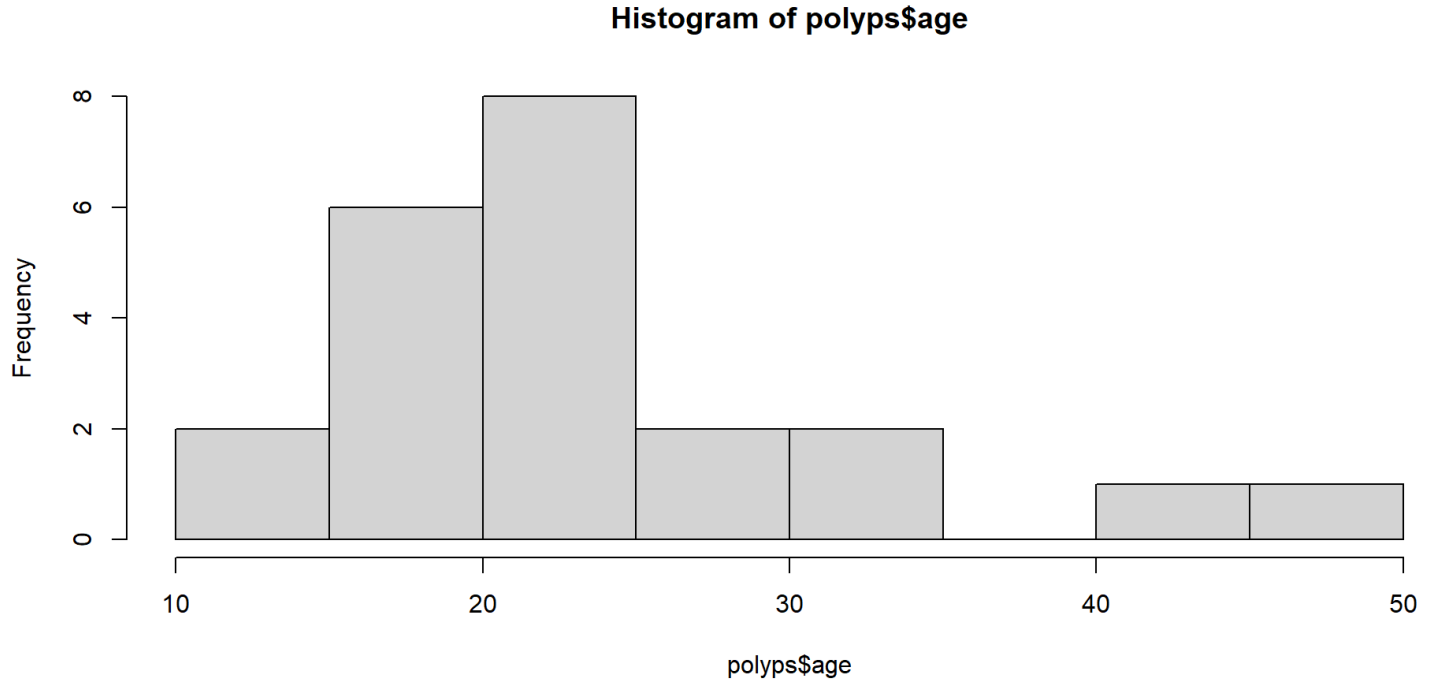
# ICA 2.2: Filtering, summarizing, and grouping

# Base R plotting

# Easy base R plots

- `hist(x)` gives a histogram for numeric vector x
- `barplot(x)` produces a bar plot of categorical variable x
- `plot(x, y)` produces a scatter plot of vectors x and y

```
1  hist(polyps$age)
```



# Wrap up

# Conclusion

We learned about:

- Managing our work with RProjects
- The pipe operator
- Selecting data frame variables, and sorting and filtering data frame rows
- Grouping and summarizing data frame rows for analysis by aggregation

Next time: visualizing with ggplot2!