Introduction to R - Day 2

Lidiya Mishieva

29 January, 2026

Intro

- Day 1: basic syntax, classes, objects, functions
- **Day 2: base package, tidy programming & tidyverse**
- Day 3: RMarkdown / Quarto
- Day 4: Git & RStudio

Recap last week

- Assignment
- Object types
- Functions
- Packages

Tidy paradigm - intuitive ‘readable’ syntax

- Consider this story:
    1. *Little bunny Foo Foo*
    2. *Went hopping through the forest*
    3. *Scooping up the field mice*
    4. *And bopping them on the head*
- Translate into code:
-
```
# define the object
foo_foo <- little_bunny()
# define the first operation
foo_foo_1 <- hop(foo_foo, through = forest)
# define the second operation
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
# define the thirs operation
foo_foo_3 <- bop(foo_foo_2, on = head)
```

Tidy paradigm - intuitive ‘readable’ syntax

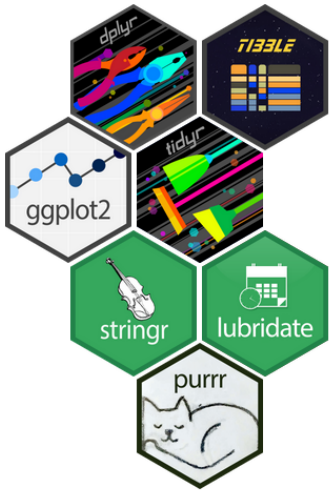
- `%>%` is the pipe operator (part of `magrittr` package), passing the results of the previous operation to the next:
  - ```
    foo_foo %>%
      hop(through = forest) %>%
      scoop(up = field_mice) %>%
      bop(on = head)
    ```
  - The idea is to do this:
  - ```
    use_this %>%
      do_this_first() %>%
      do_this_second() %>%
      do_this_third()
    ```
  - Instead of this:
  - ```
    do_this_third(do_this_second(do_this_first(use_this)))
    ```
    ```

Tidy paradigm - intuitive ‘readable’ syntax

- R 4.1.0 introduced a *native* pipe operator `|>`
- The essence of the behavior between `%>%` and `|>` is the same
  - “pipe an object forward to a function”
- But there are some differences
  - `%>%` is more complicated, but also more flexible
  - Details about the differences are summarized here:
    - https://tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/

Tidy paradigm - tidyverse

Tidyverse



R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Tidy paradigm - pharmaverse

Tidy paradigm - tibbles and dataframes

- tibbles are very similar to data.frames
- output of tidyverse functions is usually a tibble (if performed on a data.frame)
- unlike data frames, tibbles don’t show the entire dataset when you print it:
-

```r
# load data
penguins <- datasets::penguins
# object class is data.frame
class(penguins)
# print the data
penguins
# convert to tibble
penguins <- tibble::as_tibble(penguins)
# object class is tibble now
class(penguins)
# print the data
penguins
# we can also do the same with dataframes by uisnd head()
head(as.data.frame(penguins))
```

Tidy paradigm - tibbles and dataframes

- tibbles cannot access a column when you provide a partial name of the column, but data frames can
-
```r
penguins_df <- as.data.frame(penguins)
penguins_tibble <- tibble::as_tibble(penguins)

# works on dataframes
penguins_df$sp
# does not work on tibbles
penguins_tibble$sp
penguins_tibble$species
```

Tidy paradigm - tibbles and dataframes

- when you access only one column of a tibble, it will keep the tibble structure. But when you access one column of a data frame, it will become a vector
-
```
penguins_tibble[,"species"]
class(penguins_tibble[,"species"]) # will stay a tibble

penguins_df[,"species"]
class(penguins_df[,"species"]) # will beocme a vector
```

Tidy paradigm - tibbles and dataframes

- when assigning a new column to a tibble, the input will not be recycled, which means you have to provide an input of the same length of the other columns
- but a data frame will recycle the input

```
# will return an error
penguins_tibble$new_var <- 5:6
# will recycle the input across the rows
penguins_df$new_var <- 5:6
penguins_df$new_var
```

Tidy paradigm - tibbles and dataframes

- tibbles don’t support support arithmetic operations on all columns well, the result will be converted into a data frame without any notice
- ```
  1 penguins_tibble$new_var <- penguins_tibble$bill_len^2
  2 class(penguins_tibble)
  ```

> **Note**
>
> Take home message: tibbles and data.frames both work well in most cases, just be aware what object class you are working with.
  ```

Working with data

Working with data - import/export

- different import functions (and even packages) for different data formats
  - importing csv files:

```
# baseR:
read.csv(path)
# tidy:
readr::read_csv(path)
readr::read_csv2(path) # uses a comma as separator
```

  - importing excel files:

```
readxl::read_excel(path)
```

  - importing other common data files :

```
haven::read_dta(path)
haven::read_sas(path)
haven::read_sav(path)
haven::read_stata(path)
haven::read_spss(path)
```

Working with data - import/export

- export functions are usually just called `write_` instead of `read_` (check the documentations), but you have to provide the object to be exported and the path in separate arguments, e.g.:

```
1  write_csv(data, file="somepath.csv")
```

- export to excel is not straightforward, better to export csv and then open using excel if needed

Working with data - merging (base)

- `rbind()`
  - stacks rows (adds observations)
  - data frames must have same columns (names & types)
  - example: `rbind(data1, data2)`
- `cbind()`
  - binds columns (adds variables)
  - same number of rows required
  - example: `cbind(data1, new_col)`
- `merge()`
  - join by keys
  - example: `merge(data1, data2, by = "id")`

Working with data - merging (tidy)

- **`dplyr::bind_rows()`**
  - tidyverse version of `rbind()`
  - handles mismatched columns (fills with `NA`)
  - example: `bind_rows(data1, data2)`
- **`dplyr::bind_cols()`**
  - tidyverse version of `cbind()`
  - row counts must match
  - example: `bind_cols(data1, data2)`
- **merging (all from `dplyr`)**
  - `left_join()` & `right_join()`
  - `inner_join()` & `full_join()`

Working with data - long to wide & wide to long (base)

```r
# load an example dataset from the package datasets (long format)
data_long <- datasets::Indometh

# long to wide
data_wide <- reshape(data_long,             # data in the long format
                     idvar = "Subject",     # names of one or more variables in long format that
                                            # identify multiple records from the same group/individual

                     timevar = "time",      # the variable in long format that differentiates multiple
                                            # records from the same group or individual

                     v.names = "conc",      # names of variables in the long format that correspond
                                            # to multiple variables in the wide format

                     direction = "wide")    # direction of reshaping

# wide to long
data_long <- reshape(data_wide,             # data in the wide format
                     idvar = "Subject",     # names of one or more variables in long format that
                                            # identify multiple records from the same group/individual

                     v.names = "conc",      # names of variables in the long format that correspond
                                            # to multiple variables in the wide format

                     direction = "long")    # direction of reshaping
```

Working with data - long to wide & wide to long (tidy)

```r
# load an example dataset from the package datasets (long format)
data_long <- datasets::Indometh

# long to wide
data_wide <- data_long %>%
  pivot_wider(
    names_from = time,          # which column (or columns) to get the name of the output
    values_from = conc,         # which column (or columns) to get the cell values from
    names_prefix = "conc.")     # string added to the start of every variable name

# wide to long
data_long <- data_wide %>%
  pivot_longer(
    cols = starts_with("conc"),  # columns to pivot into longer format (selection example, use only one)
    cols = conc.0.25:conc.8,     # columns to pivot into longer format (selection example, use only one)

    names_to = "time",           # a character vector specifying the new column or columns to create
                                 # from the information stored in the column names of data specified by cols

    names_prefix = "conc.",      # a regular expression used to remove matching text from the start of each
                                 # variable name

    values_to = "conc")          # a string specifying the name of the column to create from the data stored
                                 # in cell values
```

Working with data - subsetting/ filtering

- base R
  - `data[rows, cols]`
  - example: `data[data$age > 30, c("name", "age")]`
  - get row indices: `data[which(data$age > 30), ]`
- tidy
  - rows: `data %>% filter(age > 30)`
  - columns: `select(data, name, age)`
  - in a pipe: `data %>% select(name, age) %>% filter(age > 30)`
  - helper functions: `starts_with("string")`, `contains("string")`
  - or use regular expressions (regex):
    - https://r4ds.hadley.nz/regexps.html

Working with data - renaming variables (base)

- rename by name:
  - `names(data)[names(data) == "old"] <- "new"`
- rename by position:
  - `names(data)[2] <- "new_name"`
- with `colnames()`:
  - `colnames(data)[colnames(data) == "old"] <- "new"`

Working with data - renaming variables (tidy)

- `rename()`: `data %>% rename(new = old)`
- rename multiple variables at once (`rename_with()` expects a function, indicated by `~`):
  - `data %>% rename_with(~ paste0("var", seq_along(.x)))`
    - `paste0() # concatenate vectors after converting to character`
    - `seq_along(.x) # generate a sequence along with the length of .x`
    - `.x` placeholder for the columns of the dataframe; if your dataframe has columns `a, b, c`, then `.x = c("a","b","c")`
    - If we don't use `.x` instead of `seq_along(.x)`, we will concatenate var and variable names from the data instead of var and a sequence number
  - `data %>% rename_with(~ gsub("-", "_", .x))`
    - `gsub()` renames the pattern provided in the first argument, by the argument provided in the second

Working with data - transformations

- base R:
  - add new column:
    - `data$new_var <- data$x * 2`
  - conditional logic:
    - `data$group <- ifelse(data$age > 30, "old", "young")`
- tidy:
  - mutate():
    - `data %>% mutate(new_var = x * 2)`
  - conditional:
    - `data %>% mutate(group = if_else(age > 30, "old", "young"))`
  - multiple variables at once:
    - `data %>% mutate(across(starts_with("x"), log))`

Working with data - modeling

- Linear regression: `lm()`
  - `lm(y ~ x1 + x2, data = data)`
- Some generalized linear models (logistic, poisson): `glm()`
  - `glm(y ~ x1 + x2, data = data, family = binomial)`
  - `glm(y ~ x1 + x2, data = data, family = poisson)`
- Mixed-effects models: `lme4::lmer()` `lme4::glmer()`
  - `lme4::lmer(y ~ x1 + x2 + (1 | group), data = data)`
- Survival models:
  - Cox-model:
    - `survival::coxph(Surv(time, event) ~ x1 + x2, data = data)`
  - Kaplen-Meier:
    - `survival::survfit(Surv(time, event) ~ 1, data = data)`

Next session

- Creating plots and tables
- Reporting with RMarkdown/Quarto

The end

https://allisonhorst.com/