Lecture 8: Tidy evaluation

Contents

Lecture learning objectives:

By then end of this lecture & worksheet 8, students should be able to:

- Describe data masking as it relates to the `dplyr` functions. Explain the problems it solves for interactive programming and the problems it creates for programming in a non-interactive setting
- Explain what the `enquo()` function and the `!!` operator do in R in the context of data masking as it relates to the `dplyr` functions
- Use the `{{` (read: curly curly) operator (abstracts quote-and-unquote into a single interpolation step), the `:=` (read:

Skip to main content

```
library(gapminder)
library(tidyverse)
options(repr.matrix.max.rows = 5)
```

```
── Attaching core tidyverse packages ──────────────────────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.2     ✔ readr     2.1.4
✔ forcats   1.0.0     ✔ stringr   1.5.0
✔ ggplot2   3.4.3     ✔ tibble    3.2.1
✔ lubridate 1.9.2     ✔ tidyr     1.3.0
✔ purrr     1.0.2
```

```
── Conflicts ──────────────────────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

What Metaprogramming lets you do in R

- write `library(purrr)` instead of `library("purrr")`
- enable `plot(x, sin(x))` to automatically label the axes with `x` and `sin(x)`
- create a model object via `lm(y ~ x1 + x2, data = df)`
- and much much more (that you will see in Data Wrangling as we explore the tidyverse)

What is metaprogramming?

Code that writes code/code that mutates code.

Skip to main content

Our narrow focus on metaprogramming for this course:

Tidy evaluation

Why focus on tidy evaluation

In the rest of MDS you will be relying on functions from the tidyverse to do a lot of:

- data wrangling
- statistics
- data visualization

Tidy evaluation

The functions from the tidyverse are beautiful to use interactively.

```
gapminder
```

Skip to main content

A tibble: 1704 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Afghanistan	Asia	1952	28.801	8425333	779.4453
Afghanistan	Asia	1957	30.332	9240934	820.8530
Afghanistan	Asia	1962	31.997	10267083	853.1007
⋮	⋮	⋮	⋮	⋮	⋮
Zimbabwe	Africa	2002	39.989	11926563	672.0386
Zimbabwe	Africa	2007	43.487	12311143	469.7093

with base r:

```
gapminder[gapminder$country == "Canada" & gapminder$year == 1952, ]
```

A tibble: 1 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Canada	Americas	1952	68.75	14785584	11367.16

In the tidyverse:

```
filter(gapminder, country == "Canada", year == 1952)
```

Skip to main content

A tibble: 1 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Canada	Americas	1952	68.75	14785584	11367.16

How does that even work?

- When functions like `filter` are called, there is a delay in evaluation and the data frame is temporarily promoted as first class objects, we say the data masks the workspace
- This is to allow the promotion of the data frame, such that it masks the workspace (global environment)
- When this happens, R can then find the relevant columns for the computation

This is referred to as data masking

Back to our example:

What is going on here?

- code evaluation is delayed
- the `filter` function quotes columns `country` and `year`
- the `filter` function then creates a data mask (to mingle variables from the environment and the data frame)
- the columns `country` and `year` and unquoted and evaluated within the data mask

Skip to main content

A tibble: 1 × 6

country <fct>	continent <fct>	year <int>	lifeExp <dbl>	pop <int>	gdpPercap <dbl>
Canada	Americas	1952	68.75	14785584	11367.16

Trade off of lovely interactivity of tidyverse functions...

programming with them can be more challenging.

Let's try writing a function which wraps filter for gapminder:

```
filter_gap <- function(col, val) {
    filter(gapminder, col == val)
}

filter_gap(country, "Canada")
```

```
Error: object 'country' not found
Traceback:

1. filter_gap(country, "Canada")
2. filter(gapminder, col == val)   # at line 4 of file <text>
3. filter.tbl_df(gapminder, col == val)
4. filter_impl(.data, quo)
```

Why does `filter` work with non-quoted variable names, but our function `filter_gap` fail?

Skip to main content

Defining functions using tidy eval's `enquo` and `!!`:

Use `enquo` to quote the column names, and then `!!` to unquote them in context.

```
filter_gap <- function(col, val) {
    col <- enquo(col)
    filter(gapminder, !!col == val)
}

filter_gap(country, "Canada")
```

A tibble: 12 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Canada	Americas	1952	68.75	14785584	11367.16
Canada	Americas	1957	69.96	17010154	12489.95
Canada	Americas	1962	71.30	18985849	13462.49
⋮	⋮	⋮	⋮	⋮	⋮
Canada	Americas	2002	79.770	31902268	33328.97
Canada	Americas	2007	80.653	33390141	36319.24

Defining functions by embracing column names: `{{ }}`

- In the newest release of `rlang`, there has been the introduction of the `{{` (pronounced "curly curly") operator.

Skip to main content

```r
filter_gap <- function(col, val) {
    filter(gapminder, {{col}} == val)
}

filter_gap(country, "Canada")
```

A tibble: 12 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Canada	Americas	1952	68.75	14785584	11367.16
Canada	Americas	1957	69.96	17010154	12489.95
Canada	Americas	1962	71.30	18985849	13462.49
⋮	⋮	⋮	⋮	⋮	⋮
Canada	Americas	2002	79.770	31902268	33328.97
Canada	Americas	2007	80.653	33390141	36319.24

(OPTIONAL) Creating functions that handle column names as strings:

Sometimes you want to pass a column name into a function as a string (often useful when you are programming and have the column names as a character vector).

You can do this by using symbols + unquoting with `sym` + `!!` :

Skip to main content

```
# example of what we want to wrap: filter(gapminder, country == "Canada")
filter_gap <- function(col, val) {
    col <- sym(col)
    filter(gapminder, !!col == val)
}

filter_gap("country", "Canada")
```

A tibble: 12 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Canada	Americas	1952	68.75	14785584	11367.16
Canada	Americas	1957	69.96	17010154	12489.95
Canada	Americas	1962	71.30	18985849	13462.49
⋮	⋮	⋮	⋮	⋮	⋮
Canada	Americas	2002	79.770	31902268	33328.97
Canada	Americas	2007	80.653	33390141	36319.24

The walrus operator `:=` is needed when assigning values

- `:=` is needed when addinging values with tidyevaluation

```
group_summary <- function(data, group, col, fun) {
    data %>%
        group_by({{ group }}) %>%
```

Skip to main content

```
group_summary(gapminder, continent, gdpPercap, mean)
```

A tibble: 5 × 2

continent	gdpPercap
<fct>	<dbl>
Africa	2193.755
Americas	7136.110
Asia	7902.150
Europe	14469.476
Oceania	18621.609

Pass the dots when you can

If you are only passing on variable to a tidyverse function, and that variable is not used in logical comparisons, or in variable assignment, you can get away with passing the dots:

```
sort_gap <- function(...) {
    arrange(gapminder, ...)
}

sort_gap(year)
```

Skip to main content

A tibble: 1704 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Afghanistan	Asia	1952	28.801	8425333	779.4453
Albania	Europe	1952	55.230	1282697	1601.0561
Algeria	Africa	1952	43.077	9279525	2449.0082
⋮	⋮	⋮	⋮	⋮	⋮
Zambia	Africa	2007	42.384	11746035	1271.2116
Zimbabwe	Africa	2007	43.487	12311143	469.7093

Notes on passing the dots

- the dots should be the last function argument (or you will not be able to use positional arguments)
- they are useful because you can add multiple arguments

For example:

```
sort_gap <- function(..., x) {
    print(x + 1)
    arrange(gapminder, ...)
}

sort_gap(year, continent, country, 2)
```

Skip to main content

```
1. sort_gap(year, continent, country, 2)
2. print(x + 1)   # at line 2 of file <text>
```

```
sort_gap <- function(x, ...) {
    print(x + 1)
    arrange(gapminder, ...)
}

sort_gap(1, year, continent, country)
```

```
[1] 2
```

A tibble: 1704 × 6

country	continent	year	lifeExp	pop	gdpPercap
<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
Algeria	Africa	1952	43.077	9279525	2449.008
Angola	Africa	1952	30.015	4232095	3520.610
Benin	Africa	1952	38.223	1738315	1062.752
⋮	⋮	⋮	⋮	⋮	⋮
Australia	Oceania	2007	81.235	20434176	34435.37
New Zealand	Oceania	2007	80.204	4115771	25185.01

Skip to main content

Pass the dots is not always the solution...

```
square_diff_n_select <- function(data, ...) {
    data %>%
        mutate(... := (... - mean(...))^2) %>%
        select(...)
}

square_diff_n_select(mtcars, mpg, mpg:hp)
```

```
Error: Problem with `mutate()` input `...`.
✖ object 'hp' not found
ℹ Input `...` is `(... - mean(...))^2`.
Traceback:

1. square_diff_n_select(mtcars, mpg, mpg:hp)
2. data %>% mutate(`:=`(..., (... - mean(...))^2)) %>% select(...)   # at line 2-4 of file <text>
3. withVisible(eval(quote(`_fseq`(`_lhs`)), env, env))
4. eval(quote(`_fseq`(`_lhs`)), env, env)
5. eval(quote(`_fseq`(`_lhs`)), env, env)
6. `_fseq`(`_lhs`)
7. freduce(value, `_function_list`)
8. function_list[[i]](value)
9. mutate(., `:=`(..., (... - mean(...))^2))
10. mutate.data.frame(., `:=`(..., (... - mean(...))^2))
11. mutate_cols(.data, ...)
12. withCallingHandlers({
  .     for (i in seq_along(dots)) {
  .         not_named <- (is.null(dots_names) || dots_names[i] ==
  .             "")
  ...
```

Skip to main content

When passing in different column names to different functions, embrace multiple column names

```r
square_diff_n_select <- function(data, col_to_change, col_range) {
    data %>%
        mutate({{ col_to_change }} := ({{ col_to_change }} - mean({{ col_to_change }}))^2) %>%
        select({{col_range}})
}

square_diff_n_select(mtcars, mpg, mpg:hp)
```

A data.frame: 32 × 4

	mpg	cyl	disp	hp
	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	0.8269629	6	160	110
Mazda RX4 Wag	0.8269629	6	160	110
Datsun 710	7.3407129	4	108	93
⋮	⋮	⋮	⋮	⋮
Maserati Bora	25.914463	8	301	335
Volvo 142E	1.714463	4	121	109

Combining embracing with pass the dots:

```r
square_diff_n_select <- function(data, col_to_change, ...) {
```

Skip to main content

```
        select(..., {{ col_to_change }})
}

square_diff_n_select(mtcars, mpg, drat, carb)
```

A data.frame: 32 × 3

	drat	carb	mpg
	<dbl>	<dbl>	<dbl>
Mazda RX4	3.90	4	0.8269629
Mazda RX4 Wag	3.90	4	0.8269629
Datsun 710	3.85	1	7.3407129
⋮	⋮	⋮	⋮
Maserati Bora	3.54	8	25.914463
Volvo 142E	4.11	2	1.714463

Programming defensively with tidy evaluation

You can embrace `{{` the column names in an `if` + `stop` statement to check user input when unquoted column names are used as function arguments.

First, we demonstrate how to check if a column is numeric using `DATA_FRAME %>% pull({{ COLUMN_NAME }})` to access the column:

```
check_if_numeric <- function(data, col) {
```

Skip to main content

```
check_if_numeric(gapminder, pop)
```

TRUE

Next, we add a `if` + `stop` to this, to throw an error in our `square_diff_n_select` function when the column type is not what our function is designed to handle. Here our function works, as the `lifeExp` column in the `gapminder` data set is numeric.

```
square_diff_n_select <- function(data, col_to_change, ...) {
    if (!is.numeric(data %>% pull({{ col_to_change }}))) {
        stop('col_to_change must be numeric')
    }

    data %>%
        mutate({{ col_to_change }} := ({{ col_to_change }} - mean({{ col_to_change }}))^2) %>%
        select(..., {{ col_to_change }})
}

square_diff_n_select(gapminder, lifeExp, country, year)
```

Skip to main content

A tibble: 1704 × 3

country	year	lifeExp
<fct>	<int>	<dbl>
Afghanistan	1952	940.8599
Afghanistan	1957	849.2818
Afghanistan	1962	755.0097
⋮	⋮	⋮
Zimbabwe	2002	379.6823
Zimbabwe	2007	255.5982

Here our function throws an error, as the `continent` column in the `gapminder` data set is **not** numeric.

```
square_diff_n_select(gapminder, continent, country, year)
```

```
Error in square_diff_n_select(gapminder, continent, country, year): col_to_change must be numeric
Traceback:

1. square_diff_n_select(gapminder, continent, country, year)
2. stop("col_to_change must be numeric")   # at line 3 of file <text>
```

What did we learn?

- data masking and its role in tidy evaluation

Skip to main content

- the walrus `:=` operator for assignment when programming with tidy-evaluated functions
- more useful examples of pass the dots `...`

Attribution:

- Tidy evaluation by Lionel Henry & Hadley Wickham
- Tidy eval in context talk by Jenny Bryan
- Programming in the tidyverse
- Advanced R by Hadley Wickham

Previous
Lecture 7: Mapping and nested data frames