

Functions

Writing your own functions

So far we've seen many functions, like `c()`, `class()`, `filter()`, `dim()` ...

Why create your own functions?

- Cut down on repetitive code (easier to fix things!)
- Organize code into manageable chunks
- Avoid running code unintentionally
- Use names that make sense to you

Writing your own functions

The general syntax for a function is:

```
function_name <- function(arg1, arg2, ...) {  
  <function body>  
}
```

Writing your own functions

Here we will write a function that multiplies some number x by 2:

```
times_2 <- function(x) x * 2
```

When you run the line of code above, you make it ready to use (no output yet!).
Let's test it!

```
times_2(x = 10)
```

```
[1] 20
```

Writing your own functions: { }

Adding the curly brackets - {} - allows you to use functions spanning multiple lines:

```
times_2 <- function(x) {  
  x * 2  
}  
times_2(x = 10)
```

```
[1] 20
```

```
is_even <- function(x) {  
  x %% 2 == 0  
}  
is_even(x = 11)
```

```
[1] FALSE
```

```
is_even(x = times_2(x = 10))
```

```
[1] TRUE
```

Writing your own functions: return

If we want something specific for the function's output, we use `return()`:

```
times_2_plus_4 <- function(x) {  
  output_int <- x * 2  
  output <- output_int + 4  
  return(output)  
}  
times_2_plus_4(x = 10)
```

```
[1] 24
```

Writing your own functions: print intermediate steps

- printed results do not stay around but can show what a function is doing
- returned results stay around
- can only return one result but can print many
- if `return` not called, last evaluated expression is returned
- `return` should be the last step (steps after may be skipped)

Adding print

```
times_2_plus_4 <- function(x) {  
  output_int <- x * 2  
  output <- output_int + 4  
  print(paste("times2 result = ", output_int))  
  return(output)  
}
```

```
result <- times_2_plus_4(x = 10)
```

```
[1] "times2 result = 20"
```

```
result
```

```
[1] 24
```


Writing your own functions: multiple inputs

Functions can take multiple inputs:

```
times_2_plus_y <- function(x, y) x * 2 + y  
times_2_plus_y(x = 10, y = 3)
```

```
[1] 23
```

Writing your own functions: multiple outputs

Functions can have one returned result with multiple outputs.

```
x_and_y_plus_2 <- function(x, y) {  
  output1 <- x + 2  
  output2 <- y + 2  
  
  return(c(output1, output2))  
}  
result <- x_and_y_plus_2(x = 10, y = 3)  
result  
  
[1] 12 5
```

Writing your own functions: defaults

Functions can have “default” arguments. This lets us use the function without using an argument later:

```
times_2_plus_y <- function(x = 10, y = 3) x * 2 + y  
times_2_plus_y()
```

```
[1] 23
```

```
times_2_plus_y(x = 11, y = 4)
```

```
[1] 26
```

Writing another simple function

Let's write a function, `sqdif`, that:

1. takes two numbers `x` and `y` with default values of 2 and 3.
2. takes the difference
3. squares this difference
4. then returns the final value

Writing another simple function

```
sqdif <- function(x = 2, y = 3) (x - y)^2
```

```
sqdif()
```

```
[1] 1
```

```
sqdif(x = 10, y = 5)
```

```
[1] 25
```

```
sqdif(10, 5)
```

```
[1] 25
```

```
sqdif(11, 4)
```

```
[1] 49
```

Writing your own functions: characters

Functions can have any kind of input. Here is a function with characters:

```
loud <- function(word) {  
  output <- rep(toupper(word), 5)  
  return(output)  
}  
loud(word = "hooray!")
```

```
[1] "HOORAY!" "HOORAY!" "HOORAY!" "HOORAY!" "HOORAY!"
```

Functions for tibbles

We can use `filter(row_number() == n)` to extract a row of a tibble:

```
get_row <- function(dat, row) dat %>% filter(row_number() == row)
```

```
ces <- calenviroscreen  
ces_1_8 <- ces %>% select(1:8)
```

```
get_row(dat = ces, row = 10)
```

```
# A tibble: 1 × 67
```

```
  CensusTract CaliforniaCounty ZIP Longitude Latitude ApproxLocation  
    <dbl> <chr>           <int>    <dbl>    <dbl> <chr>  
1  6001401000 "Alameda "           94608    -122.     37.8 Oakland  
# 61 more variables: CES4.0Score <dbl>, CES4.0Percentile <dbl>,  
# CES4.0PercRange <chr>, Ozone <dbl>, OzonePctl <dbl>, PM2.5 <dbl>,  
# PM2.5.Pctl <dbl>, DieselPM <dbl>, DieselPMPctl <dbl>, DrinkingWater <dbl>,  
# DrinkingWaterPctl <dbl>, Lead <dbl>, LeadPctl <dbl>, Pesticides <dbl>,  
# PesticidesPctl <dbl>, ToxRelease <dbl>, ToxReleasePctl <dbl>,  
# Traffic <dbl>, TrafficPctl <dbl>, CleanupSites <dbl>,  
# CleanupSitesPctl <dbl>, GroundwaterThreats <dbl>, ...
```

```
get_row(dat = ces, row = 4)
```

```
# A tibble: 1 × 67
```

```
  CensusTract CaliforniaCounty ZIP Longitude Latitude ApproxLocation  
    <dbl> <chr>           <int>    <dbl>    <dbl> <chr>  
1  6001400400 "Alameda "           94609    -122.     37.8 Oakland  
# 61 more variables: CES4.0Score <dbl>, CES4.0Percentile <dbl>,  
# CES4.0PercRange <chr>, Ozone <dbl>, OzonePctl <dbl>, PM2.5 <dbl>,
```

Functions for tibbles

`select(n)` will choose column `n`:

```
get_index <- function(dat, row, col) {  
  dat %>%  
    filter(row_number() == row) %>%  
    select(all_of(col))  
}
```

```
get_index(dat = ces, row = 10, col = 7)
```

```
# A tibble: 1 × 1  
  CES4.0Score  
    <dbl>  
1         43.7
```


Functions for tibbles

Including default values for arguments:

```
get_top <- function(dat, row = 1, col = 1) {  
  dat %>%  
    filter(row_number() == row) %>%  
    select(all_of(col))  
}
```

```
get_top(dat = ces)
```

```
# A tibble: 1 × 1  
  CensusTract  
    <dbl>  
1    6001400100
```

Functions for tibbles

Can create function with an argument that allows inputting a column name for select or other dplyr operation:

```
clean_dataset <- function(dataset, col_name) {  
  my_data_out <- dataset %>% select({{col_name}}) # Note the curly braces  
  write_csv(my_data_out, "clean_data.csv")  
  return(my_data_out)  
}
```

```
clean_dataset(dataset = ces, col_name = "CES4.0Score")
```

```
# A tibble: 8,035 × 1
```

```
  CES4.0Score
```

```
    <dbl>
```

```
1         4.85
```

```
2         4.88
```

```
3        11.2
```

```
4        12.4
```

```
5        16.7
```

```
6        20.0
```

```
7        36.7
```

```
8        37.1
```

```
9        40.7
```

```
10       43.7
```

```
#   8,025 more rows
```

Summary

- Simple functions take the form:
 - `NEW_FUNCTION <- function(x, y){x + y}`
 - Can specify defaults like `function(x = 1, y = 2){x + y}` -return will provide a value as output
 - `print` will simply print the value on the screen but not save it

Lab Part 1

▯ [Class Website](#)

▯ [Lab](#)

Functions on multiple columns

Using your custom functions: **sapply()**- a base R function

Now that you've made a function... You can “apply” functions easily with **sapply()**!

These functions take the form:

```
sapply(<a vector, list, data frame>, some_function)
```

Using your custom functions: `sapply()`

▯ There are no parentheses on the functions! ▯

You can also pipe into your function.

```
er_visits <- CO_heat_ER
```

```
head(er_visits, n = 2)
```

```
# A tibble: 2 × 7
```

	county	rate	lower95cl	upper95cl	visits	year	gender
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>
1	Statewide	5.64	4.70	6.59	140	2011	Female
2	Statewide	7.39	6.30	8.47	183	2011	Male

```
sapply(er_visits, class)
```

county	rate	lower95cl	upper95cl	visits	year
"character"	"numeric"	"numeric"	"numeric"	"numeric"	"numeric"
gender					
"character"					

```
er_visits %>% sapply(class)
```

county	rate	lower95cl	upper95cl	visits	year
"character"	"numeric"	"numeric"	"numeric"	"numeric"	"numeric"
gender					
"character"					

Using your custom functions: `sapply()`

```
select(er_visits, rate:upper95cl) %>% head()
```

```
# A tibble: 6 × 3
```

	rate	lower95cl	upper95cl
	<dbl>	<dbl>	<dbl>
1	5.64	4.70	6.59
2	7.39	6.30	8.47
3	6.51	5.80	7.23
4	5.64	4.72	6.57
5	7.56	6.48	8.65
6	6.58	5.88	7.29

```
select(er_visits, rate:upper95cl) %>%  
  sapply(times_2) %>%  
  head()
```

	rate	lower95cl	upper95cl
[1,]	11.28546	9.395283	13.17564
[2,]	14.77374	12.597645	16.94983
[3,]	13.02989	11.593179	14.46660
[4,]	11.28268	9.430621	13.13474
[5,]	15.12880	12.959418	17.29817
[6,]	13.16714	11.750214	14.58407

Using your custom functions “on the fly” to iterate

```
select(er_visits, rate:upper95cl) %>%  
  apply(function(x) x / 1000) %>%  
  head()
```

	rate	lower95cl	upper95cl
[1,]	0.005642730	0.004697642	0.006587819
[2,]	0.007386868	0.006298822	0.008474914
[3,]	0.006514945	0.005796590	0.007233300
[4,]	0.005641341	0.004715311	0.006567371
[5,]	0.007564398	0.006479709	0.008649086
[6,]	0.006583570	0.005875107	0.007292033

across

Using functions in `mutate()` and `summarize()`

Already know how to use functions to modify columns using `mutate()` or calculate summary statistics using `summarize()`.

```
er_visits %>%  
  mutate(rate_round = round(rate, 2)) %>%  
  summarize(max_rate_round = max(rate_round, na.rm = T),  
            max_rate = max(rate, na.rm = T))
```

```
# A tibble: 1 × 2  
  max_rate_round max_rate  
    <dbl>         <dbl>  
1      89.3      89.3
```

Applying functions with **across** from **dp1yr**

`across()` makes it easy to apply the same transformation to multiple columns. Usually used with `summarize()` or `mutate()`.

```
summarize(across( .cols = <columns>, .fns = function))
```

or

```
mutate(across(.cols = <columns>, .fns = function))
```

- List columns first: `.cols =`
- List function next: `.fns =`
- If there are arguments to a function (e.g., `na.rm = TRUE`), the function may need to be modified to an anonymous function, e.g., `\(x) mean(x, na.rm = TRUE)`

Applying functions with **across** from **dplyr**

Combining with `summarize()`

```
ces_dbl <- ces %>% select(CaliforniaCounty, CES4.0Score, CES4.0Percentile)
```

```
ces_dbl %>%  
  summarize(across(.cols = everything(), .fns = mean, na.rm=T))
```

```
# A tibble: 1 × 3  
  CaliforniaCounty CES4.0Score CES4.0Percentile  
      <dbl>          <dbl>          <dbl>  
1           NA          28.3          50.0
```

Applying functions with **across** from **dplyr**

Can use with other tidyverse functions like `group_by`!

```
ces_dbl %>%  
  group_by(CaliforniaCounty) %>%  
  summarize(across(.cols = everything(), .fns = mean, na.rm=T))
```

```
# A tibble: 58 × 3  
  CaliforniaCounty CES4.0Score CES4.0Percentile  
    <chr>           <dbl>         <dbl>  
1 "Alameda "      22.9          41.3  
2 "Alpine "       13.6          22  
3 "Amador "       20.7          38.8  
4 "Butte "        21.7          39.8  
5 "Calaveras "    16.1          28.0  
6 "Colusa "       27.0          52.2  
7 "Contra Costa"  21.0          36.7  
8 "Del Norte"     21.4          40.3  
9 "El Dorado"     10.2          14.6  
10 "Fresno "      40.9          69.5  
#   48 more rows
```

Applying functions with **across** from **dp1yr**

To add arguments to functions, may need to use anonymous function. In this syntax, the shorthand `\(x)` is equivalent to `function(x)`.

```
ces_dbl %>%  
  group_by(CaliforniaCounty) %>%  
  summarize(across(.cols = everything(), .fns = \(x) mean(x, na.rm = TRUE)))
```

```
# A tibble: 58 × 3  
  CaliforniaCounty CES4.0Score CES4.0Percentile  
    <chr>             <dbl>         <dbl>  
1 "Alameda "        22.9          41.3  
2 "Alpine "         13.6           22  
3 "Amador "         20.7          38.8  
4 "Butte "          21.7          39.8  
5 "Calaveras "      16.1          28.0  
6 "Colusa "         27.0          52.2  
7 "Contra Costa"    21.0          36.7  
8 "Del Norte"       21.4          40.3  
9 "El Dorado"       10.2          14.6  
10 "Fresno "        40.9          69.5  
#   48 more rows
```

Applying functions with **across** from **dplyr**

Using different `tidyselect()` options (e.g., `starts_with()`, `ends_with()`, `contains()`)

```
ces_dbl %>%  
  group_by(CaliforniaCounty) %>%  
  summarize(across(.cols = contains("Perc"), .fns = mean))
```

```
# A tibble: 58 × 2  
  CaliforniaCounty CES4.0Percentile  
    <chr>           <dbl>  
1 "Alameda "      NA  
2 "Alpine "       22  
3 "Amador "      38.8  
4 "Butte "       39.8  
5 "Calaveras "   NA  
6 "Colusa "      52.2  
7 "Contra Costa" 36.7  
8 "Del Norte"    40.3  
9 "El Dorado"    14.6  
10 "Fresno "     NA  
#   48 more rows
```


Applying functions with **across** from **dp1yr**

Combining with `mutate()`: rounding to the nearest power of 10 (with negative digits value)

```
ces_dbl %>%  
  mutate(across(  
    .cols = starts_with("CES"),  
    .fns = round,  
    digits = 3  
  ))
```

```
# A tibble: 8,035 × 3  
  CaliforniaCounty CES4.0Score CES4.0Percentile  
    <chr>          <dbl>          <dbl>  
1 "Alameda "      4.85            2.8  
2 "Alameda "      4.88            2.87  
3 "Alameda "     11.2           15.9  
4 "Alameda "     12.4           19.0  
5 "Alameda "     16.7           29.7  
6 "Alameda "     20.0           37.6  
7 "Alameda "     36.7           70.1  
8 "Alameda "     37.1           70.7  
9 "Alameda "     40.7           76.2  
10 "Alameda "     43.7           80.4  
#   8,025 more rows
```

Applying functions with **across** from **dp1yr**

Combining with `mutate()` - the `replace_na` function

`replace_na({data frame}, {list of values})` or `replace_na({vector}, {single value})`

```
yearly_co2_emissions %>%  
  select(country, starts_with("194")) %>%  
  mutate(across(  
    .cols = c(`1943`, `1944`, `1945`),  
    .fns = replace_na,  
    replace = 0  
  ))
```

A tibble: 192 × 11

	country <chr>	`1940` <dbl>	`1941` <dbl>	`1942` <dbl>	`1943` <dbl>	`1944` <dbl>	`1945` <dbl>	`1946` <dbl>	`1947` <dbl>	`1948` <dbl>
1	Afghanistan	NA	NA	NA	0	0	0	NA	NA	NA
2	Albania	693	627	744	462	154	121	484	928	704
3	Algeria	238	312	499	469	499	616	763	744	803
4	Andorra	NA	NA	NA	0	0	0	NA	NA	NA
5	Angola	NA	NA	NA	0	0	0	NA	NA	NA
6	Antigua and B...	NA	NA	NA	0	0	0	NA	NA	NA
7	Argentina	15900	14000	13500	14100	14000	13700	13700	14500	17400
8	Armenia	848	745	513	655	613	649	730	878	935
9	Australia	29100	34600	36500	35000	34200	32700	35500	38000	38500
10	Austria	7350	7980	8560	9620	9400	4570	12800	17600	24500

182 more rows

1 more variable: `1949` <dbl>

Use custom functions within `mutate` and `across`

If your function needs to span more than one line, better to define it first before using inside `mutate()` and `across()`.

```
times1000 <- function(x) x * 1000
```

```
airquality %>%  
  mutate(across(  
    .cols = everything(),  
    .fns = times1000  
  )) %>%  
  head(n = 2)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41000	190000	7400	67000	5000	1000
2	36000	118000	8000	72000	5000	2000

```
airquality %>%  
  mutate(across(  
    .cols = everything(),  
    .fns = function(x) x * 1000  
  )) %>%  
  head(n = 2)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41000	190000	7400	67000	5000	1000
2	36000	118000	8000	72000	5000	2000

purrr package

Similar to `across`, `purrr` is a package that allows you to apply a function to multiple columns in a data frame or multiple data objects in a list.

While we won't get into `purrr` too much in this class, its a handy package for you to know about should you get into a situation where you have an irregular list you need to handle!

Multiple Data Frames

Multiple data frames

Lists help us work with multiple data frames

```
AQ_list <- list(AQ1 = airquality, AQ2 = airquality, AQ3 = airquality)
str(AQ_list)
```

List of 3

```
$ AQ1:'data.frame':    153 obs. of  6 variables:
 ..$ Ozone   : int [1:153] 41 36 12 18 NA 28 23 19 8 NA ...
 ..$ Solar.R: int [1:153] 190 118 149 313 NA NA 299 99 19 194 ...
 ..$ Wind    : num [1:153] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 ..$ Temp    : int [1:153] 67 72 74 62 56 66 65 59 61 69 ...
 ..$ Month   : int [1:153] 5 5 5 5 5 5 5 5 5 5 ...
 ..$ Day     : int [1:153] 1 2 3 4 5 6 7 8 9 10 ...
$ AQ2:'data.frame':    153 obs. of  6 variables:
 ..$ Ozone   : int [1:153] 41 36 12 18 NA 28 23 19 8 NA ...
 ..$ Solar.R: int [1:153] 190 118 149 313 NA NA 299 99 19 194 ...
 ..$ Wind    : num [1:153] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 ..$ Temp    : int [1:153] 67 72 74 62 56 66 65 59 61 69 ...
 ..$ Month   : int [1:153] 5 5 5 5 5 5 5 5 5 5 ...
 ..$ Day     : int [1:153] 1 2 3 4 5 6 7 8 9 10 ...
$ AQ3:'data.frame':    153 obs. of  6 variables:
 ..$ Ozone   : int [1:153] 41 36 12 18 NA 28 23 19 8 NA ...
 ..$ Solar.R: int [1:153] 190 118 149 313 NA NA 299 99 19 194 ...
 ..$ Wind    : num [1:153] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 ..$ Temp    : int [1:153] 67 72 74 62 56 66 65 59 61 69 ...
 ..$ Month   : int [1:153] 5 5 5 5 5 5 5 5 5 5 ...
 ..$ Day     : int [1:153] 1 2 3 4 5 6 7 8 9 10 ...
```

Multiple data frames: **sapply**

```
AQ_list %>% sapply(class)
```

```
      AQ1      AQ2      AQ3  
"data.frame" "data.frame" "data.frame"
```

```
AQ_list %>% sapply(nrow)
```

```
AQ1 AQ2 AQ3  
153 153 153
```

```
AQ_list %>% sapply(colMeans, na.rm = TRUE)
```

```
      AQ1      AQ2      AQ3  
Ozone  42.129310  42.129310  42.129310  
Solar.R 185.931507 185.931507 185.931507  
Wind    9.957516  9.957516  9.957516  
Temp   77.882353  77.882353  77.882353  
Month   6.993464  6.993464  6.993464  
Day    15.803922  15.803922  15.803922
```

Summary

- Apply your functions with `sapply(<a vector or list>, some_function)`
- Use `across()` to apply functions across multiple columns of data
- Need to use `across` within `summarize()` or `mutate()`
- Can use `sapply` or `purrr` to work with multiple data frames within lists simultaneously

Lab Part 2

▮ [Class Website](#)

▮ [Lab](#)



Image by [Gerd Altmann](#) from [Pixabay](#)