

# Intro to R

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

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

The general syntax for a function is:

```
functionName <- function(inputs) {  
  <function body>  
  return(value)  
}
```

## 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

## 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)
```

```
cars <- read_kaggle()
cars_1_8 <- cars %>% select(1:8)
```

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

```
# A tibble: 1 × 34
```

```
  RefId IsBadBuy PurchDate Auction VehYear VehicleAge Make Model Trim SubMod
  <dbl>   <dbl> <chr>      <chr>   <dbl>      <dbl> <chr> <chr> <chr> <chr>
1    10         0 12/7/2009 ADESA      2007          2 FORD FIVE... SEL  4D SED
# ... with 24 more variables: Color <chr>, Transmission <chr>, WheelTypeID <chr>,
# WheelType <chr>, VehOdo <dbl>, Nationality <chr>, Size <chr>,
# TopThreeAmericanName <chr>, MMRAcquisitionAuctionAveragePrice <chr>,
# MMRAcquisitionAuctionCleanPrice <chr>,
# MMRAcquisitionRetailAveragePrice <chr>,
# MMRAcquisitionRetailCleanPrice <chr>, MMRCurrentAuctionAveragePrice <chr>,
# MMRCurrentAuctionCleanPrice <chr>, MMRCurrentRetailAveragePrice <chr>, ...
```

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

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           4.6         3.1         1.5         0.2   setosa
```

## 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 = cars, row = 10, col = 8)
```

```
# A tibble: 1 × 1  
  Model  
  <chr>  
1 FIVE HUNDRED
```



# 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 = cars)
```

```
# A tibble: 1 × 1  
  RefId  
  <dbl>  
1      1
```

# 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.

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

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa

```
sapply(iris, class)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
"numeric"	"numeric"	"numeric"	"numeric"	"factor"

```
iris %>% sapply(class)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
"numeric"	"numeric"	"numeric"	"numeric"	"factor"

## Using your custom functions: `sapply()`

```
select(cars, VehYear:VehicleAge) %>% head()
```

```
# A tibble: 6 × 2
  VehYear VehicleAge
  <dbl>      <dbl>
1    2006           3
2    2004           5
3    2005           4
4    2004           5
5    2005           4
6    2004           5
```

```
select(cars, VehYear:VehicleAge) %>%
  sapply(times_2) %>%
  head()
```

```
      VehYear VehicleAge
[1, ]    4012           6
[2, ]    4008          10
[3, ]    4010           8
[4, ]    4008          10
[5, ]    4010           8
[6, ]    4008          10
```

## Using your custom functions “on the fly” to iterate

```
select(cars, VehYear:VehicleAge) %>%  
  sapply(function(x) x / 1000) %>%  
  head()
```

	VehYear	VehicleAge
[1, ]	2.006	0.003
[2, ]	2.004	0.005
[3, ]	2.005	0.004
[4, ]	2.004	0.005
[5, ]	2.005	0.004
[6, ]	2.004	0.005



across

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

`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 =`
- Then list any arguments for the function (e.g., `na.rm = TRUE`)

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

Combining with `summarize()`

```
cars_dbl <- cars %>% select(Make, starts_with("Veh"))  
  
cars_dbl %>%  
  summarize(across(.cols = everything(), .fns = mean))
```

```
# A tibble: 1 × 5  
  Make VehYear VehicleAge VehOdo VehBCost  
  <dbl>   <dbl>      <dbl>  <dbl>   <dbl>  
1    NA   2005.        4.18 71500.   6731.
```

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

Can use with other tidyverse functions like `group_by`!

```
cars_dbl %>%  
  group_by(Make) %>%  
  summarize(across(.cols = everything(), .fns = mean))
```

```
# A tibble: 33 × 5  
  Make      VehYear VehicleAge VehOdo VehBCost  
  <chr>      <dbl>      <dbl>   <dbl>   <dbl>  
1 ACURA      2003.        6.52  81732.   9039.  
2 BUICK       2004.        5.65  76238.   6169.  
3 CADILLAC    2004.        5.24  73770.  10958.  
4 CHEVROLET   2006.        3.97  73390.   6835.  
5 CHRYSLER    2006.        3.65  66814.   6507.  
6 DODGE       2006.        3.75  68261.   7047.  
7 FORD        2005.        4.75  76749.   6403.  
8 GMC         2004.        5.61  79273.   8342.  
9 HONDA       2004.        5.33  77877.   8350.  
10 HUMMER     2006         3      70809  11920  
# ... with 23 more rows
```

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

Combining with `summarize()`:

```
# Adding arguments to the end!
#
cars_dbl %>%
  group_by(Make) %>%
  summarize(across(.cols = everything(), .fns = mean, na.rm = TRUE))
```

```
# A tibble: 33 × 5
  Make      VehYear VehicleAge VehOdo VehBCost
  <chr>      <dbl>      <dbl>   <dbl>   <dbl>
1 ACURA    2003.        6.52 81732.   9039.
2 BUICK     2004.        5.65 76238.   6169.
3 CADILLAC  2004.        5.24 73770.  10958.
4 CHEVROLET 2006.        3.97 73390.   6835.
5 CHRYSLER  2006.        3.65 66814.   6507.
6 DODGE     2006.        3.75 68261.   7047.
7 FORD      2005.        4.75 76749.   6403.
8 GMC       2004.        5.61 79273.   8342.
9 HONDA     2004.        5.33 77877.   8350.
10 HUMMER   2006         3    70809  11920
# ... with 23 more rows
```

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

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

```
cars_dbl %>%  
  group_by(Make) %>%  
  summarize(across(.cols = starts_with("Veh"), .fns = mean))
```

```
# A tibble: 33 × 5  
  Make      VehYear VehicleAge VehOdo VehBCost  
  <chr>      <dbl>      <dbl>   <dbl>   <dbl>  
1 ACURA      2003.        6.52 81732.    9039.  
2 BUICK       2004.        5.65 76238.    6169.  
3 CADILLAC    2004.        5.24 73770.   10958.  
4 CHEVROLET   2006.        3.97 73390.    6835.  
5 CHRYSLER    2006.        3.65 66814.    6507.  
6 DODGE       2006.        3.75 68261.    7047.  
7 FORD        2005.        4.75 76749.    6403.  
8 GMC         2004.        5.61 79273.    8342.  
9 HONDA       2004.        5.33 77877.    8350.  
10 HUMMER     2006         3      70809   11920  
# ... with 23 more rows
```

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

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

```
cars_dbl %>%  
  mutate(across(  
    .cols = starts_with("Veh"),  
    .fns = round,  
    digits = -3  
  ))
```

```
# A tibble: 72,983 × 5
```

	Make	VehYear	VehicleAge	Veh0do	VehBCost
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	MAZDA	2000	0	89000	7000
2	DODGE	2000	0	94000	8000
3	DODGE	2000	0	74000	5000
4	DODGE	2000	0	66000	4000
5	FORD	2000	0	69000	4000
6	MITSUBISHI	2000	0	81000	6000
7	KIA	2000	0	65000	4000
8	FORD	2000	0	66000	4000
9	KIA	2000	0	50000	6000
10	FORD	2000	0	85000	8000

```
# ... with 72,973 more rows
```

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

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

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

```
# Child mortality data
mort <- read_mortality() %>% rename(country = `...1`)
```

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

```
# A tibble: 197 × 11
```

	country	`1940`	`1941`	`1942`	`1943`	`1944`	`1945`	`1946`	`1947`	`1948`	`1949`
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Afghan...	NA	NA	NA	0	0	0	NA	NA	NA	NA
2	Albania	1.53	1.31	1.48	1.46	1.43	1.40	1.37	1.41	1.37	1.34
3	Algeria	NA	NA	NA	0	0	0	NA	NA	NA	NA
4	Angola	4.46	4.46	4.46	4.34	4.34	4.34	4.33	4.22	4.22	4.21
5	Argent...	0.641	0.603	0.602	0.558	0.551	0.510	0.503	0.496	0.494	0.492
6	Armenia	NA	NA	NA	0	0	0	NA	NA	NA	NA
7	Aruba	NA	NA	NA	0	0	0	NA	NA	NA	NA
8	Austra...	0.263	0.275	0.276	0.299	0.260	0.271	0.295	0.279	0.271	0.271
9	Austria	0.504	0.474	0.417	0.389	0.360	0.311	0.311	0.312	0.274	0.274
10	Azerba...	NA	NA	NA	0	0	0	NA	NA	NA	NA

```
# ... with 187 more rows
```



## Use custom functions within `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.

## map\_df

```
library(purrr)
airquality %>% map_df(replace_na, replace = 0)
```

```
# A tibble: 153 × 6
   Ozone Solar.R Wind Temp Month Day
  <int>   <int> <dbl> <int> <int> <int>
1     41     190   7.4     67     5     1
2     36     118    8      72     5     2
3     12     149  12.6     74     5     3
4     18     313  11.5     62     5     4
5      0      0  14.3     56     5     5
6     28      0  14.9     66     5     6
7     23     299   8.6     65     5     7
8     19      99  13.8     59     5     8
9      8      19  20.1     61     5     9
10     0     194   8.6     69     5    10
# ... with 143 more rows
```

# 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()`
- `purrr` is a package that you can use to do more iterative work easily
- 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](#)