In [1]:
```
library(tidyverse)
```

```
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
✓ ggplot2 3.3.3     ✓ purrr   0.3.4
✓ tibble  3.0.6     ✓ dplyr   1.0.4
✓ tidyr   1.1.2     ✓ stringr 1.4.0
✓ readr   1.4.0     ✓ forcats 0.5.1
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

Warm-up:

- Write a function with a for-loop inside a conditonal (or vice versa!)

In [21]:
```
x <- c(1,2,3,4,5)
```

In [62]:
```
for (i in x) {
    if (i %% 2 == 0) {
        print(i^2)
    } else {
        print(i)
    }
}
```

```
[1] 1
[1] 4
[1] 3
[1] 16
[1] 5
```

In [111]:
```
square_even_identity_odd <- function (x) {
    for (i in seq_along(x)) {
        if (x[[i]] %% 2 == 0) {
            x[[i]] <- x[[i]]^2
        }
    }
    x
}
```

In [114]:
```
x
square_even_identity_odd(x)
x
```

1 · 2 · 3 · 4 · 5

1 · 4 · 3 · 16 · 5

1 · 2 · 3 · 4 · 5

Global vs. Local Variables

In [115]:
```
x <- 3
plus_one <- function(x) {
    x+1
}
```

What will happen in the following two lines?

In [118]:
```
plus_one(0)
x
```

1

3

- Inside the function, the variable `x` is a **local** variable, with value determined by the input
- Outside the function, the variable `x` is a **global** variable, with value assigned to be `3`

In [126]:
```
y <- 2
add_y <- function(x) {
    x+y
}
```

In [127]:
```
add_y(5)
```

7

- Inside of `add_y()`, the variable `x` is local and `y` is global
- If a function cannot find a variable in its local environment, it will look in the global environment

In [128]:
```
# don't do this
do_not <- function(x) {
    do_not <- function(x) {
        x+100
    }
    do_not(x)
}
do_not(1)
```

101

- Inside of the local function environment for `do_not()` is the function `do_not()` which is only defined inside of the local environment
- Try to avoid:
  - Using global variables inside of functions
  - Naming local variables in function environments the same as other global variables
  - Overwriting function names from base R or tidyverse

In [129]:
```
y <- 2
add_two <- function(x,z) {
    x+z
}
```

Arguments of functions

- We can specify arguments in different ways
  - Position
  - Exact Matching

In [130]:
```
print_three <- function(x,y,z) {
    print(x)
    print(y)
    print(z)
}
```

In [133]:
```
print_three(1,2,3)
```

```
[1] 1
[1] 2
[1] 3
```

In [135]:
```
print_three(x = 1, y = 2, z = 3)
```

```
[1] 1
[1] 2
[1] 3
```

In [136]:
```
print_three(2, 3, x=1)
```

```
[1] 1
[1] 2
[1] 3
```

In [ ]:
```
inner_join(tibl, tib2, by = "var")
```

You can also specify optional arguments in functions

In [150]:
```
print_three <- function(x,y,z, print_last=TRUE) {
    print(x)
    print(y)
    if (print_last) {print(z)}
}
```

In [151]:
```
print_three(1,2)
```

```
[1] 1
[1] 2
Error in print(z): argument "z" is missing, with no default
Traceback:

1. print_three(1, 2)
2. print(z)   # at line 4 of file <text>
```

In [138]:
```
print_three(1,2,3)
```

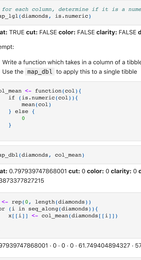
```
[1] 1
[1] 2
[1] 3
```

In [141]:
```
print_three(1,2,3, print_last=FALSE)
print_three(1,2,3,FALSE)
```

```
[1] 1
[1] 2
[1] 1
[1] 2
```

In [ ]:
```
# pseudo-code
# aes <- function(x,y,color=FALSE) {
#             # if (color==FALSE) {
#                 #do nothing
#             } else { graph by color}
#}
```

In [49]:
```
ggplot(mpg, aes(x=cty, y=hwy, color=class, by=, alpha= )) + geom_point()
```



In [154]:
```
# possible, but not great practice
# try to use exact matching with optional arguments
print_three(1,2,3,FALSE)
```

```
[1] 1
[1] 2
```

You can create a function with arbitrarily many arguments

In [155]:
```
sum_three_one <- function(...){
    sum(...)+1
}
```

In [157]:
```
sum_three_one(1,2)
sum_three_one(1,2,3)
sum_three_one(1,2,100,100,100)
sum_three_one()
```

4

7

304

1

You can also "end" functions early with `return()`

In [160]:
```
# useful for special cases when the rest of the function is too loog for a simple if...else
random_calc <- function(x) {

    if (identical(x, 0)) return("Cannot divide by zero.")

    y <- 1/x
    1/y + 1
}
```

In [162]:
```
# this is an equivalent function
random_calc <- function(x) {

    if (identical(x, 0)){
      "Cannot divide by zero."
    } else {
      y <- 1/x
      1/y + 1
    }
}
```

In [163]:
```
random_calc(0)
random_calc(10)
```

'Cannot divide by zero.'

11

Functionals: Map

- A functional is a "function" that takes as input another function
- A common for-loop is to iterate over a vector/list, change each element, and return the altered vector/list
- The functions below streamline this common for-loop
  - `map()` outputs a list
  - `map_dbl()` outputs a double vector
  - `map_int()` outputs an integer vector
  - `map_lgl()` outputs a logical vector
  - `map_chr()` outputs a character vector

In [164]:
```
plus_one <- function(x) {x+1}
```

In [165]:
```
# I can use a for loop
plus_one_vec <- function(v) {
    for (i in seq_along(v)) {
        v[[i]] <- plus_one(v[[i]])
    }
    v
}

x <- c(1,2,3,4,5)
plus_one_vec(x)
```

2 · 3 · 4 · 5 · 6

In [166]:
```
# or I can use the map function
# this applies the function "plus_one" to each element of x
map_dbl(x, plus_one)
```

2 · 3 · 4 · 5 · 6

In [168]:
```
# for each column, determine if it is a numeric vector
map_lgl(diamonds, is.numeric)
```

carat: TRUE **cut:** FALSE **color:** FALSE **clarity:** FALSE **depth:** TRUE **table:** TRUE **price:** TRUE **x:** TRUE **y:** TRUE **z:** TRUE

Attempt:

- Write a function which takes in a column of a tibble and outputs the mean if it is numeric and 0 otherwise
- Use the `map_dbl` to apply this to a single tibble

In [169]:
```
col_mean <- function(col){
    if (is.numeric(col)){
        mean(col)
    } else {
        0
    }
}
```

In [172]:
```
map_dbl(diamonds, col_mean)
```

carat: 0.797939747868001 **cut:** 0 **color:** 0 **clarity:** 0 **depth:** 61.749404894327 **table:** 57.457183908046 **price:** 3932.79972191324 **x:** 5.73115721171672 **y:** 5.73452595476455 **z:** 3.53873737782715

In [173]:
```
x <- rep(0, length(diamonds))
for (i in seq_along(diamonds)){
    x[[i]] <- col_mean(diamonds[[i]])
}
x
```

0.797939747868001 · 0 · 0 · 0 · 61.749404894327 · 57.457183908046 · 3932.79972191324 · 5.73115721171672 · 5.73452595476455 · 3.53873737782715

In [ ]:
```

```