Lecture 13: Writing functions

Managing and Manipulating Data Using R

Introduction

Libraries and data we will use today

Libraries

```
library(tidyverse)
#> -- Attaching packages ---------------------------------------------------------
#> v ggplot2 3.2.1          v purrr   0.3.2
#> v tibble  2.1.3          v dplyr   0.8.3
#> v tidyr   1.0.0.9000     v stringr 1.4.0
#> v readr   1.3.1          v forcats 0.4.0
#> -- Conflicts ---------------------------------------------------------------
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(haven)
library(labelled)
```

Data frame

```
#load dataset with one obs per recruiting event
load(url("https://github.com/ozanj/rclass/raw/master/data/recruiting/recruit_ev
```

Logistics

Remaining lectures

- Lecture 12 (last week): Loops
- Lecture 13 (today): Functions
- Lecture 14: Intro to GitHub

These topics are important, but challenging.

- **Learning goals**: Develop strong conceptual understanding of the basics; some practice applying these skills

Reading to do before next class:

- Grolemund and Wickham chapter 19 (Functions)
- [OPTIONAL] Any slides from lecture we don't cover
  - I wrote this lecture knowing we won't have time to get through all sections
  - Slides we don't cover are mainly for your future reference

To do before next class:

- Set up a GitHub account

What we will do today

1. Introduction
    1.1 What are functions
    1.2 When and why write functions
2. Function basics
    2.1 How to write a function
    2.2 Practice: z_score function
    2.3 Practice: count_events function
    2.4 Student exercise
3. Conditional execution
4. Function arguments
    4.1 Default values
    4.2 Dot-dot-dot (...)
5. Return values
6. Writing functions that humans can understand

What are functions

What are functions

Functions are pre-written bits of code that accomplish some task. Functions generally follow three sequential steps:

1. take in an **input** object(s)
2. **process** the input.
3. **return**. Returns a new object, which may be a vector, data-frame, plot, etc.

We've been working with functions all quarter.

Example: the `select()` function:

```
#type ?select in console
#?select
```

1. **input**. takes in a data frame object as the input
2. **processing**. keeps selected variables that you specify
3. **return**. Returns a new object, which is a data frame containing variables you specify

```
select(df_event,event_type,event_state,zip) %>% str()
df_event %>% select(event_type,event_state,zip) %>% str() # same result
```

What are functions

Functions are pre-written bits of code that accomplish some task. Functions generally follow three sequential steps:

1. take in an **input** object(s)
2. **process** the input.
3. **return**. Returns a new object, which may be a vector, data-frame, plot, etc.

Example: The `sum()` function:

```
#?sum
```

1. **input**. takes in a vector of elements (*class* must be numeric or logical)
2. **processing**. Calculates the sum of elements
3. **return**. Returns numeric vector: length=1; value is sum of input vector

```
sum(c(1,2,3))
#> [1] 6
sum(c(1,2,3)) %>% str()
#>  num 6
```

What are user-written functions

"**user-written functions**" [my term]

- functions you write to perform some specific task, often a data-manipulation or analysis task specific to your project

Like all functions, user-written functions usually follow three steps:

1. take in one or more **inputs**
2. **process** the inputs
    - this may include using pre-written functions like `select()` or `sum()`
3. **return** a new object

Example things we might want to write a function to do:

- Using total population in zip-code and population for each race/ethnicity group, write function that create variables for percent of people in each race/ethnicity group for each zip-code
    - Modify function so it can create zip-code level variables AND statelevel variables
- Write function to read in annual data; call function for each year
- Create interactive maps: NAED_presentation
    - see "deep dive" results

When and why write functions

When should you write a function

Let's introduce task we might want to achieve by writing a function

- Dataset `df_event` : one observation for each university-recruiting_event
    - Variable `event_type` : location type of recruiting event (e.g., public high school)

Task:

- Create the following descriptive statistics tables for each university
    - **Table A**: count of number of recruiting events by event type and the average of median income at each event type
    - **Table B**: same as Table A, but separately for in-state and out-of-state events

Here is some code to create these tables for Stonybrook University in New York

```r
#Table A
df_event %>% filter(univ_id==196097) %>% group_by(event_type) %>%
  summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))
#Table B
df_event %>% filter(univ_id==196097) %>% group_by(event_inst, event_type) %>%
  summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))
```

When should you write a function

To write a function or not

- A function is a self-contained bit of code that performs some specific task; functions allow you to "automate" tasks that you perform more than once
  - e.g., for off-campus recruiting descriptive stats (above): we would write a function, then "call" the function separately for each university
- The alternative to writing a function to perform some specific task is to copy and paste the code each time you want to perform a task
  - e.g., for off-campus recruiting descriptive stats: we would copy above code for each university and change the university ID

Advice about when to write a function from experts

- Grolemund and Wickham chapter 19:
  - "You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)."
- Darin Christenson refers to the programming mantra **DRY**
  - "Do not Repeat Yourself (DRY)"
  - "Functions enable you to perform multiple tasks (that are similar to one another) without copying the same code over and over"

Why write functions

Advantages of writing functions to complete a task compared to the copy-and-paste approach

- As task requirements change (and they always do!), you only need to revise code in one place rather than many places
- Functions give you an opportunity to make continual improvements to completing a task
  - Often, I have two tasks and I write a separate function for each task.
  - Over time, I realize that these two tasks have many things in common and that I can write a single function that completes both tasks.
- Reduce errors that are common in copy-and-paste approach (e.g., forgetting to change variable name or variable value)

Learning how to write functions is a requirement for anybody working on our research projects

- When the RAs move on, we need to be able to efficiently modify tasks they completed. This is only possible when they write functions.

Why write functions (my own experience)

How I use functions in my research (acquiring, processing, and analyzing data)

1. **Acquiring data**. Since I often create longitudinal datasets from annual "input data," I usually write a function or loop to read-in the data and do initial processing
   - After writing a function for a specific data source, I generalize the function to read-in other data sources that share commonalities
     - e.g., US Census Bureau ACS Data; IPEDS
2. **Processing data** (the big step between acquiring data and analyzing data). Write functions for data processing steps:
   - sometimes these are small/quick steps that I do over and over
     - e.g., cleaning a "string" ID variable
   - sometimes these are big/multi-step processes
     - e.g., write function that takes-in longitudinal data on number of degrees awarded by field and award-level for each university, and creates measures of "degree adoption"
3. **Analyzing data** (after creating analysis datasets). I **ALWAYS** write functions to automate analyses and the creation of tables/graphs
   - As a young research assistant, bosses were always asking me to change the variables and then recreate the regression tables (same scenario for dissertation analyses + chairs, R&R manuscripts + reviewers)
   - Write functions that allow me to specify which models to run, which variables to include, etc and then "spit out" polished, publication-ready tables

Function basics

Strategy for learning to write functions

How I'll approach teaching you how to write functions

1. Introduce the basic components of a function
2. Non-practical example:
    - start by writing a function that simply prints "hello"
    - then, we'll make iterative improvements to this function
3. Practical example: create descriptive tables for off-campus recruiting project
    - start by writing simple version of this function
    - then, we'll make iterative improvements to this function
4. student tasks: practice writing functions with a partner
5. Then, we'll introduce more complicated elements of writing a function (e.g., conditional execution)

Central theme is the importance of continually revising your functions

How to write a function

Three components of a function

The `function()` function tells R that you are writing a function

```
function_name <- function(x,y,z) {
  #function body
}
#?base #for help on `function()` type "`?base`" in console, click on "index," an
```

Three components of a function:

1. **function name**
   - specify function name before the assignment operator `<-`
2. **function arguments** (sometimes called “inputs” or “arguments”)
   - Inputs that the function takes
     - can be vectors, data frames, logical statements, strings, etc.
   - in above hypothetical code, the function took three inputs `x` , `y` , `z`
     - we could have written this instead: `function(Larry,Curly,Moe)`
   - In “function call,” you specify values to assign to these function arguments
3. **function body**
   - What the function does to the inputs
   - Above hypothetical function doesn’t do anything

Hello function

Task: First example is to write a function that simply prints "Hello!"

Perform task outside of function

- First step in writing a function to perform a task is always to perform the task outside of a function

```
"Hello!"
#> [1] "Hello!"
```

Create the function

```
print_hello <- function() {
  "Hello!"
}
```

1. **function name**
   - function name is `print_hello`
2. **function arguments** (sometimes called "inputs")
   - the `print_hello` function doesn't take any arguments
3. **function body** (what the function does to the inputs)
   - body of `print_hello` simply prints "Hello!"

Call the function

```
print_hello()
#> [1] "Hello!"
```

Hello function

Task: modify `print_hello` function so it also prints our name, which we specify as an input.

First, perform task outside a function. A few approaches we could take:

1. This seems wrong because my name is not an input

```
"Hello! My name is Karina Salazar"
#> [1] "Hello! My name is Karina Salazar"
```

2. Why doesn't this approach work?

```
x <- "Karina Salazar"
x
#> [1] "Karina Salazar"
"Hello! My name is x"
#> [1] "Hello! My name is x"
```

3. Why doesn't this approach work?

```
"Hello! My name is " x
```

4. This approach sort of works

```
"Hello! My name is "
#> [1] "Hello! My name is "
x
#> [1] "Karina Salazar"
```

Hello function

Task: modify `print_hello` function so it also prints our name, which we specify as an input.

First, perform task outside a function.

- Let's take another approach. Experiment with the `print()` function

```
#?print
print("Hello! My name is")
#> [1] "Hello! My name is"
print(x)
#> [1] "Karina Salazar"
```

- Want our `print_hello` function to print everything on one line. Why doesn't this work?

```
print("Hello! My name is") print(x)
print("Hello! My name is"), print(x)
```

What went wrong? seems like `print()` function:

- Can only print one object at a time
- Can't put two instances of `print()` on same line of code
- Each instance of `print()` will be printed on separate line

Hello function

Task: modify `print_hello` function so it also prints our name, which we specify as an input.

First, perform task outside a function.

- We need to find an alternative to `print()` that can print multiple objects on the same line
- Let's use `cat()` function [we used `cat()` in regular expressions lecture!]

```
#?cat
cat("Hello! My name is ")
#> Hello! My name is
cat(x)
#> Karina Salazar

cat("Hello! My name is",x)
#> Hello! My name is Karina Salazar
```

Success! Now we can write a function for this task

Hello function

Task: modify `print_hello` function so that it also prints our name

Perform task outside a function.

```r
x <- "Karina Salazar"
cat("Hello! My name is",x)
#> Hello! My name is Karina Salazar
```

Create function

```r
print_hello <- function(name) {
  cat("Hello! My name is",name)
}
```

1. **function name** is `print_hello`
2. **function arguments**. "inputs" to the function
   - takes one argument, `name`; could have named this argument `x` or `Ralph`
3. **function body**. What function does to the inputs
   - `cat("Hello! My name is",name)`

Call function

```r
print_hello("Patricia Martin")
#> Hello! My name is Patricia Martin

#print_hello(Patricia Martin) #note: this doesn't work
```

Hello function

Task: modify `print_hello` function so that it also takes our year of birth as an input and states our age

Perform task outside of function

```
x <- "Ozan Jaquette"
y <- 1979
z <- 2019 - 1979
cat("Hello! My name is",x,". In 2019 I will turn",z,"years old")
#> Hello! My name is Ozan Jaquette . In 2019 I will turn 40 years old
```

Improvements we could make (before writing function):

1. Remove extra space between name and the period
   - `sep` argument of `cat()` defines what to put after each element
   - default is `sep = " "`; change to `sep=""` and specify spaces manually

```
#?cat
cat("Hello! My name is ",x,". In 2019 I will turn ",z," years old", sep="")
#> Hello! My name is Ozan Jaquette. In 2019 I will turn 40 years old
```

2. use **date functions** to:
   - specify current date (rather than manually typing "2018")
   - calculate age exactly (rather than as current year minus birth year)
   - But we haven't learned date functions, so hold off

Hello function

Task: modify `print_hello` so it takes year of birth as input and states our age

Perform task outside of function

```
cat("Hello! My name is ",x,". In 2019 I will turn ",z," years old", sep="")
#> Hello! My name is Ozan Jaquette. In 2019 I will turn 40 years old
```

Create function

```
print_hello <- function(name,birth_year) {
  age <- 2019 - birth_year
  cat("Hello! My name is ",name,". In 2019 I will turn ",age," years old", sep=
}
```

1. **function name** is `print_hello`
2. **function arguments**. "inputs" to the function
   - `print_hello` function takes two arguments, `name` and `birth_year`
3. **function body**.What function does to the inputs
   - `age <- 2019 - birth_year`
   - `cat("Hello! My name is",name,"and in 2019 I will turn",age,"years old")`

Call function

```
print_hello("Karina Salazar",1989)
#> Hello! My name is Karina Salazar. In 2019 I will turn 30 years old
```

Recipe for writing a function

Recipe for first version of a function:

1. Experiment with performing the task outside of a function
    - experiment with performing task with different sets of inputs
    - sometimes you will have to revise this code, when an approach that worked outside a function does not work within a function
2. Write the function
3. Test the function; try to "break" it

As you use this function, make continual improvements going back-and-forth between steps 1-3

Practice: z_score function

`z_score` function

Task: Write function that calculates `z-score` for each element of a vector

- Z-score for observation *i* `=` number of standard deviations from mean
- $z_i = \frac{x_i - \bar{x}}{sd(x)}$

Create a vector of numbers we'll use to develop `z_score` function

```
v=c(seq(5,15))
v
#>  [1]  5  6  7  8  9 10 11 12 13 14 15
typeof(v) # type==integer vector
#> [1] "integer"
class(v) # class == integer
#> [1] "integer"
length(v) # number of elements in object v
#> [1] 11
v[1] # 1st element of v
#> [1] 5
v[10] # 10th element of v
#> [1] 14
```

Components of z-score using `mean()` and `sd()` functions

```
mean(v)
#> [1] 10
sd(v)
#> [1] 3.316625
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$

First experiment calculating z-score without writing function

- Calculate z-score for some value

```
(5-mean(v))/sd(v)
#> [1] -1.507557
(10-mean(v))/sd(v)
#> [1] 0
```

- Calculate z-score for particular elements of vector `v`

```
v[1]
#> [1] 5
(v[1]-mean(v))/sd(v)
#> [1] -1.507557
v[8]
#> [1] 12
(v[8]-mean(v))/sd(v)
#> [1] 0.6030227
```

- Calculate `z_i` for multiple elements of vector `v`

```
c(v[1],v[8],v[11])
#> [1]  5 12 15
c((v[1]-mean(v))/sd(v),(v[8]-mean(v))/sd(v),(v[11]-mean(v))/sd(v))
#> [1] -1.5075567  0.6030227  1.5075567
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$

Next, write function to calculate z_score for each element of vector

```
z_score <- function(x) {
  (x - mean(x))/sd(x)
}

#test function
#note use of c() function to indicate individual arguments for multiple calls
z_score(c(5,6,7,8,9,10,11,12,13,14,15))
#>  [1] -1.5075567 -1.2060454 -0.9045340 -0.6030227 -0.3015113  0.0000000
#>  [7]  0.3015113  0.6030227  0.9045340  1.2060454  1.5075567
v=c(seq(5,15))
z_score(v)
#>  [1] -1.5075567 -1.2060454 -0.9045340 -0.6030227 -0.3015113  0.0000000
#>  [7]  0.3015113  0.6030227  0.9045340  1.2060454  1.5075567
z_score(c(seq(20,25)))
#> [1] -1.3363062 -0.8017837 -0.2672612  0.2672612  0.8017837  1.3363062
```

Components of function

1. **function name** is `z_score`
2. **function arguments**. Takes one input, which we named `x`
   - inputs can be vectors, dataframes, logical statements, etc.
3. **function body**.What function does to the inputs
   - for each element of `x`, calculate difference between value of element and mean value of elements, then divide by standard deviation of elements

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$

Improve our function by trying to break it

```
w=c(NA,seq(1:5),NA)
w
#> [1] NA  1  2  3  4  5 NA
z_score(w)
#> [1] NA NA NA NA NA NA NA
```

- What went wrong?

Let's revise our function

```
z_score <- function(x) {
  (x - mean(x, na.rm=TRUE))/sd(x, na.rm=TRUE)
}
z_score(w)
#> [1]         NA -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
#> [7]         NA
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$ [STUDENTS WORK ON THEIR OWN]

Does our `z_score` function work when applied to variables from a data frame?

- Create data frame called `df`

```r
set.seed(12345) # set "seed" so we all get the same "random" numbers
df <- tibble(
  a = c(NA,rnorm(9)),
  b = c(NA,rnorm(9)),
  c = c(NA,rnorm(9))
)
class(df) # class of object df
df # print data frame
df$a # print element "a" (i.e., variable "a") of object df (data frame)
str(df$a) # structure of element "a" of df: a numeric vector
```

- Apply `z_score` function to variables in data frame

```r
mean(df$a, na.rm=TRUE) # mean of variable "a"
sd(df$a, na.rm=TRUE) # std dev of variable "a"

df$a # print variable "a"
z_score(df$a) # z_score function to calculate z-score for each obs of variable "
(df$a[2] - mean(df$a, na.rm=TRUE))/sd(df$a, na.rm=TRUE) # check result

z_score(df$b) # z-score for each obs of variable "b"
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$ [STUDENTS WORK ON THEIR OWN]

Task

- Use our `z_score` function to create a new variable that is the z-score version of a variable

Base R approach

Why learn "Base R" approach?

- For some tasks, using Tidyverse functions within a user-written function or within a loop requires more advanced programming skills

Show how to create and delete variables using "Base R" approach

```
df # print data frame df

df$one <- 1 # create variable "one" that always equals 1
df # print data frame df
df$one <- NULL # remove variable "one"
df

df$c_plus2 <- df$c+2 #create variable equal to "c" plus 2
df
df$c_plus2 <- NULL # remove variable "c_plus2"
df
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$ [STUDENTS WORK ON THEIR OWN]

Task

- Use our `z_score` function to create a new variable that is the z-score version of a variable

Base R approach

```
z_score <- function(x) { # note: same function as before
  (x - mean(x, na.rm=TRUE))/sd(x, na.rm=TRUE)
}

#Simply calling function doesn't create new variable
z_score(df$c)
```

Assign new variable, using z_score function to create variable values

- **Note**: Preferred approach is to not create new variable within the function

```
df$c_z <- z_score(df$c)
df$c_z
#>  [1]         NA  0.7824644  0.1323014  0.5126742  1.0474944 -0.6136182
#>  [7] -1.3324528 -1.3677077  1.3237878 -0.4849435
```

Examine data frame

```
df
df$c_z <- NULL # remove variable
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$

Task

- Use our `z_score` function to create a new variable that is the z-score version of a variable

Tidyverse approach

```
z_score <- function(x) { #same function as before
  (x - mean(x, na.rm=TRUE))/sd(x, na.rm=TRUE)
}

df %>% mutate(
  a_z = z_score(a),
  c_z = z_score(c)
  ) %>% select(a_z,c_z) %>% str()
#> Classes 'tbl_df', 'tbl' and 'data.frame':    10 obs. of  2 variables:
#>  $ a_z: num  NA 0.7775 0.9302 -0.0785 -0.5025 ...
#>  $ c_z: num  NA 0.782 0.132 0.513 1.047 ...
```

Changes not retained unless we assign

```
names(df)
df <- df %>% mutate(
  a_z = z_score(a),
  c_z = z_score(c)
  )
df
```

`z_score` function, $z_i = \frac{x_i - \bar{x}}{sd(x)}$ [STUDENTS WORK ON THEIR OWN]

We can apply our function to a "real" dataset too

```
df_event_small <- df_event[1:10,] %>% # keep first 10 observations
  select(instnm,univ_id,event_type,med_inc) # keep 4 vars
#df_event_small

#show mean, std dev for variable med_inc
df_event_small %>% summarise_at(
  .vars = vars(med_inc),
  .funs = funs(mean, sd, .args=list(na.rm=TRUE)))
#> Warning: funs() is soft deprecated as of dplyr 0.8.0
#> Please use a list of either functions or lambdas:
#>
#>   # Simple named list:
#>   list(mean = mean, median = median)
#>
#>   # Auto named with `tibble::lst()`:
#>   tibble::lst(mean, median)
#>
#>   # Using lambdas
#>   list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
#> This warning is displayed once per session.
#> # A tibble: 1 x 2
#>     mean     sd
#>    <dbl>  <dbl>
#> 1 78643. 11392.
```

Practice: count_events function

`count_events` function

Let's write a function for a practical data analysis task

- Dataset `df_event`: one observation for each university-recruiting_event
- Variable `event_type`: location type of recruiting event (e.g., public HS)

Task: Create the following descriptive statistics table for each university

- **Table A**: count of number of recruiting events by event type and in-state/out-of-state and the average of median income at each event type

Before writing function, we perform task outside a function.

First, identify value of ID variable for each university (this took me some time)

```r
names(df_event)
df_event %>% count(univ_id) # "univ_id" is id var for each university
df_event %>% count(instnm) #"instnm" is the name for university

#identify univ_id value assoicated with each university name
df_event %>% select(instnm,univ_id) %>%
  group_by(univ_id) %>% # group by university ID
  filter(row_number()==1) %>% # grab first row for each group (univ_id)
  arrange(univ_id) # sort by univ_id
```

`count_events` function

Task: calculate number of events and avg. household income by:

1. event-type and whether event is in-state/out-of-state

Create "by event-type & in-state/out-state" table outside function (Table A)

- Number of events by type

```
df_event %>% count(event_inst, event_type)
```

- Number of events by type and in-state/out-of-state and avg. income for all public universities

```
df_event %>% group_by(event_inst, event_type) %>% summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))
```

- Number of events by type and avg. income for particular university
  - e.g., U. Arkansas univ_id==106397

```
df_event %>% filter(univ_id==106397) %>% group_by(event_inst,event_type) %>%
  summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))
```

`count_events` function

Task: calculate number of events and avg. household income by:

1. event-type and whether event is in-state/out-of-state (Table A)

Create function

```
count_events <- function(id) {
#by event-type and in/out state
df_event %>% filter(univ_id==id) %>% group_by(event_inst,event_type) %>%
  summarise(n_events=n(), mean_inc=mean(med_inc, na.rm = TRUE))
}
```

1. **function name**: `count_events`
2. **function arguments**: Takes one input, which we named `id`
3. **function body**. What function does to the inputs

Call function

```
count_events(106397) # U. Arkansas
count_events(215293) # U. of Pittsburgh
```

Student exercise

Student exercise: `num_negative` function

Adapted from Ben Skinner's *programming 1* R Workshop HERE

Before presenting task, we'll create a sample dataset `df` that contains some negative values

- code omitted; don't worry about understanding this code

```
#> # A tibble: 100 x 4
#>        id   age sibage parage
#>     <int> <dbl>  <dbl>  <dbl>
#>  1      1    17      8     49
#>  2      2    15    -97     46
#>  3      3   -97    -97     53
#>  4      4    13     12     -4
#>  5      5   -97     10     47
#>  6      6    12     10     52
#>  7      7   -99      5     51
#>  8      8   -97     10     55
#>  9      9    16      6     51
#> 10     10    16    -99     -8
#> # ... with 90 more rows
```

Student exercise: `num_negative` function

Some common tasks when working with survey data:

- identify number of observations with `NA` values for a specific variable
- identify number of observations with negative values for a specific variable
- Replace negative values with `NA` for a specific variable

Your task for student exercise:

- Write a function that counts the number of observations with negative values for a specific variable
- Apply this function to variables from dataframe `df`

Recommended steps:

- Perform task outside of function
  - You use "Base R" or Tidyverse approach to counting negative values
  - Base R HINT: `sum(data_frame_name$var_name<0)`
  - Tidyverse HINT: `filter(var_name<0)` + `nrow()`
- Write function
- Apply/test function on variables

Student exercise: `num_negative` function [SOLUTION]

Task:

- count number of observations with negative values for specific variable

Step 1: Perform task outside of function [output omitted]

```
names(df) # identify variable names
df$age # print observations for a variable

#BaseR
sum(df$age<0) # count number of obs w/ negative values for variable "age"

#Tidyverse
df %>%
  filter(age<0) %>%
  nrow() # count number of obs w/ negative values for variable "age"
```

Step 2: Write function

```
num_missing <- function(x){
  sum(x<0)
}
```

Step 3: apply function

```
num_missing(df$age)
#> [1] 27
num_missing(df$sibage)
#> [1] 22
```

OPTIONAL Student exercise: `num_missing` function

In survey data, negative values often refer to reason for missing values

- e.g., `-8` refers to "didn't take survey"
- e.g., `-7` refers to "took survey, but didn't answer this question"

Your task: Write function `num_mising` that counts number of missing observations for a variable and allows you to specify which values are associated with missing for that variable. This function will take two arguments:

- `x` : the variable (e.g., `df$sibage`)
- `miss_vals` : vector of values you want to associate with "missing" variable
  - Values to associate with missing for `df$age` : `-97,-98,-99`
  - Values to associate with missing for `df$sibage` : `-97,-98,-99`
  - Values to associate with missing for `df$parage` : `-4,-7,-8`

Recommended steps:

- Perform task outside of function (recommend "Base R" approach)
  - HINT: `sum(data_frame_name$var_name< %in% c(-4,-5))`
- Write function
- Apply/test function on variables

OPTIONAL Student exercise: `num_missing` function [SOLUTION]

Perform task outside of function

```
sum(df$age %in% c(-97,-98,-99))
#> [1] 27
```

Write function

```
num_missing <- function(x, miss_vals){

  sum(x %in% miss_vals)
}
```

Call function

```
num_missing(df$age,c(-97,-98,-99))
#> [1] 27
num_missing(df$sibage,c(-97,-98,-99))
#> [1] 22
num_missing(df$parage,c(-4,-7,-8))
#> [1] 17
```

Conditional execution

Conditional execution

`if` statements allow you to conditionally execute certain blocks of code depending on whether some condition is satisfied

- From (http://r4ds.had.co.nz/functions.html#conditional-execution)

```
if (TRUE/FALSE condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

Review `TRUE` / `FALSE` conditions and `type==logical`

- Examples of `TRUE` / `FALSE` conditions

```
(2+2==4)
#> [1] TRUE
(2+2==5)
#> [1] FALSE
```

- How do you know if "condition" is `TRUE` / `FALSE` ? It has `type==logical`

```
typeof(2+2==4)
#> [1] "logical"
typeof(2+2==5)
#> [1] "logical"
typeof(2+2)
#> [1] "double"
```

Conditional execution, simple example

Task

- Imagine you are developing an administrative software program that sends students an email about whether they are on academic probation
- Write a function that takes `gpa` as an input and does the following:
  - if `gpa` is less than `2`, function prints GPA and says they are on probation;
  - otherwise, function prints GPA and says they are not on probation

```r
email_gpa <- function(gpa) {
  if (gpa<2) {

    cat("Students with a GPA below 2.0 are on academic probation. Your GPA is",
        gpa,"and you are on academic probation. You must follow these steps..."

  } else {
    cat("Your GPA is",gpa,"and you are not on academic probation.")
  }
}
email_gpa(1.9)
#> Students with a GPA below 2.0 are on academic probation. Your GPA is 1.9 and
email_gpa(3)
#> Your GPA is 3 and you are not on academic probation.
```

`condition` must evaluate to either `TRUE` or `FALSE`

The condition must evaluate to either `TRUE` or `FALSE`. This means:

1. condition must evaluate to `type==logical`
2. condition must have `length==1`

To demonstrate, we write function that takes one input, `x`, and does this:

- prints the `type` and `length` of `x`
- if condition `x` evaluates to `TRUE`, prints: “condition is true”
- otherwise, prints: “condition is not true”

```
eval_condition <- function(x) {
  cat("condition type is:",typeof(x), fill=TRUE)
  cat("condition length is:",length(x), fill=TRUE)
  if (x) {
    "condition is true"
  } else {
    "condition is not true"
  }
}
eval_condition(TRUE)
eval_condition(4==4)
eval_condition(4==3)
eval_condition("hello")
eval_condition(NA)
eval_condition(c(4==4))
eval_condition(c(4==4,4==3))
```

Conditions with multiple logical expressions

A `condition` can have multiple logical expressions as long as the `condition` evaluates to `TRUE` / `FALSE`

- Use `||` (or) and `&&` (and) to combine multiple logical expressions
- GW: "Never use | or & in an if statement: these are **vectorised** operations that apply to multiple values (that's why you use them in filter())"

Task. Write function `go_to_daycare` that takes two inputs: `weekday` (0/1 indicator); and `temp`

- if `weekday==1` **and** `temp` less than `99`, print: "Kid goes to daycare!"
- otherwise, print: "Kid stays home"

```
go_to_daycare <- function(weekday,temp) {
   if (weekday==1 && temp<99) {
    "Kid goes to daycare!"
  } else {
    "Kid stays home"
  }
}
go_to_daycare(1,98)
go_to_daycare(1,101)
go_to_daycare(1,99)
go_to_daycare(0,98)
```

Multiple conditions

Can use multiple `if` statements together if you have two or more conditions you want to specify in the code

```
if (condition) {
  # run this code if condition TRUE
} else if (condition) {
  # run this code if previous condition FALSE and this condition TRUE
} else {
  # run this code if all previous conditions FALSE
}
```

Student exercise. Write function `email_gpa` that takes one input, `gpa`, and prints the following text based on `gpa` (text would go in email to student):

- if `gpa` less than 2, prints: "Your GPA is [INSERT `gpa`]. You are on academic probation."
- else if `gpa` is greater than or equal to 3.5, prints: "Your GPA is [INSERT `gpa`]. You made the Dean's list. Congratulations!"
- otherwise, prints: "Your GPA is [INSERT `gpa`]"

SOLUTION ON NEXT SLIDE

Multiple conditions

Student exercise. Write function `email_gpa` that takes one input, `gpa`, and prints the following text based on `gpa` (text would go in email to student):

- if `gpa` less than 2, prints: "Your GPA is [INSERT `gpa`]. You are on academic probation."
- else if `gpa` is greater than or equal to 3.5, prints: "Your GPA is [INSERT `gpa`]. You made the Dean's list. Congratulations!"
- otherwise, prints: "Your GPA is [INSERT `gpa`]"

Solution

```
email_gpa <- function(gpa) {
  if (gpa<2) {
    cat("Your GPA is ",gpa,". You are on academic probation.", sep="")
  } else if (gpa>=3.5) {
    cat("Your GPA is ",gpa,". You made the Dean's list. Congratulations!", sep=
  } else {
    cat("Your GPA is ",gpa,".", sep="")
  }
}
email_gpa(1.9)
email_gpa(3.5)
email_gpa(3)
```

Conditional execution: coding style

See Grolemund and Wickham 19.4.3 for recommendations about coding style

Function arguments

Types of function arguments

Recall that user-written functions have three components

1. **function name**
2. **function arguments** (sometimes called “inputs”)
    - Inputs that the function takes
    - In “function call,” you specify values to assign to these function arguments
3. **function body**
    - What the function does to the inputs

Two broad types of arguments (according to Grolemund and Wickham):

1. **Data arguments**. Arguments that supply the data that will be processed by the function
2. **Detail arguments**. Arguments that control details of the computation

Recommended order of arguments (according to Grolemund and Wickham):

- **data arguments** come first
- **detail arguments** should come at the end and should often have a `default` value

Default values

Default values for arguments

A **default value** is the value that will be assigned to a function argument if the function call does not explicitly assign a value to that argument

- Most functions we have been working with have default values

Example: help file for the `mean()` function shows the default values

- `mean(x, trim = 0, na.rm = FALSE, ...)`
- `na.rm` is an argument of `mean()`
  - default value of `na.rm` is `FALSE`, meaning that missing values will not be removed prior to calculating mean

```
#?mean
mean(c(2,4,6,NA))
#> [1] NA
mean(c(2,4,6,NA), na.rm=FALSE) # same as default
#> [1] NA
mean(c(2,4,6,NA), na.rm=TRUE)
#> [1] 4
```

Default values for arguments

When writing function, specify **default values** for an argument the same way you would specify values for that argument when calling the function

Task. Modify `go_to_daycare` function that says whether kid goes to daycare

- Replace input `temp` with input `fever` (0/1 indicator)
- `fever` should have default value of `0`

```
go_to_daycare <- function(weekday,fever = 0) {
  cat("weekday==",weekday,"; fever==",fever,sep="", fill=TRUE)
  if (weekday==1 && fever==0) {
    "Kid goes to daycare!"
  } else {
    "Kid stays home"
  }
}
go_to_daycare(1,0)
#> weekday==1; fever==0
#> [1] "Kid goes to daycare!"
go_to_daycare(weekday=1,fever=0)
#> weekday==1; fever==0
#> [1] "Kid goes to daycare!"
go_to_daycare(weekday=1,fever=1)
#> weekday==1; fever==1
#> [1] "Kid stays home"
go_to_daycare(weekday=1)
#> weekday==1; fever==0
#> [1] "Kid goes to daycare!"
```

Dot-dot-dot (...)

Dot-dot-dot (`...`)

Many functions take an arbitrary number of arguments/inputs, e.g. `select()`

```
select(df_event,instnm,univ_id,event_type,med_inc) %>% names()
#> [1] "instnm"     "univ_id"    "event_type" "med_inc"
```

These functions rely on a special argument `...` (pronounced dot-dot-dot)

- the `...` argument captures any number of arguments that aren't otherwise matched

`filter()` function also uses `...`:

- syntax: `filter(.data,...)`
- First argument is data frame; remaining arguments are any number of filters you apply to the data

```
filter(df_event, event_state=="CA",med_inc>100000,event_type=="public hs") %>%
  count()
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   504
```

Dot-dot-dot (`...`) example:count_events function revisited

Recall our simple `count_events` function to produce descriptive table:

```
count_events <- function(id) {
  df_event %>% filter(univ_id==id) %>% group_by(event_inst,event_type) %>%
    summarise(n_events=n(), mean_inc=mean(med_inc, na.rm = TRUE))
}
count_events(106397)
```

Task

- Revise `count_events` function so that `group_by()` variables can be specified at program call

Challenge in completing this task:

- number of `group_by()` variables indeterminate
  - e.g., `group_by(event_type)` or `group_by(event_inst,event_type)`

Dot-dot-dot (`...`) example:count_events function revisited

Task. Revise `count_events()` so `group_by()` vars specified at program call

As a first step, revise `count_event` function so that we specify **one** `group_by()` variable at program call

```
count_events <- function(id, group_by_var) {
  df_event %>% filter(univ_id==id) %>% group_by(group_by_var) %>%
    summarise(n_events=n(), mean_inc=mean(med_inc, na.rm = TRUE))
}

count_events(id=106397, "event_type")
count_events(id=106397, event_type)
```

Why didn't this work?

- Answer is complicated
  - basically `group_by()` wants variable names [without quotes] listed within `group_by()`
  - but our function passes `group_by_var` (Here `event_type`) as a string
- More complete explanation HERE

Dot-dot-dot (`...`) example: count_events function revisited

Task. As a first step, revise `count_event` function so that we specify **one** `group_by()` variable at program call

Solution:

- use `group_by_()` within your function instead of `group_by`
  - `group_by_()` uses "standard evaluation" [Google it later]

```r
count_events <- function(id, group_by_var) {
  df_event %>% filter(univ_id==id) %>% group_by_(group_by_var) %>%
    summarise(n_events=n(), mean_inc=mean(med_inc, na.rm = TRUE))
}

count_events(id=106397, "event_type")
```

Note: when writing functions, this approach works for all `dplyr` functions

- e.g., when writing function, use `summarise_()` rather than `summarise()`
- e.g., when writing function, use `filter_()` rather than `filter()`

Dot-dot-dot (`...`) example: count_events function revisited

Now, we can complete our **Task**.

- Revise `count_events()` so `group_by()` vars specified at program call

```
count_events <- function(id, ...) {
  df_event %>% filter(univ_id==id) %>% group_by_(...) %>%
    summarise(n_events=n(), mean_inc=mean(med_inc, na.rm = TRUE))
}
count_events(id=106397, "event_type")
#> Warning: group_by_() is deprecated.
#> Please use group_by() instead
#>
#> The 'programming' vignette or the tidyeval book can help you
#> to program with group_by() : https://tidyeval.tidyverse.org
#> This warning is displayed once per session.
count_events(id=106397, "event_inst", "event_type")
count_events(id=106397, "event_state")
```

2. **function arguments/inputs**
   - `function(id, ...)` states the first argument is named `id` and the function will additionally take any number of un-named arguments
3. **function body**
   - `%>% group_by_(...)` means substitute the un-named arguments (which you specify in function call) as inputs to `group_by_()` function
4. **function call**
   - `count_events(id=106397, "event_inst", "event_type")`: insert `"event_inst"` and `"event_type"` as values for unnamed arguments

Return values

Return values [in functions written by others]

Return value of a function is object created ("returned") after function runs

- this could be a vector, a list, a data frame, etc
- In help-file for any function, the section **Value** describes return value

Example: `sum()` (a "Base R" function)

- syntax: `sum(..., na.rm=TRUE/FALSE)`
- returns numeric vector: length== `1`; value is sum of all values within `...`

```
#?sum
sum(df_event$fr_lunch, na.rm = TRUE) # number of free/reduced lunch students
#can use str(), length(), etc to examine what is returned by function
str(sum(df_event$fr_lunch, na.rm = TRUE)) # numeric vector
length(sum(df_event$fr_lunch, na.rm = TRUE)) # length=1
```

Example: `select()` (a Tidyverse function from `dplyr` package)

- syntax: `select(data_frame_name, ...)`
- returns: a data frame containing variables selected within `...`

```
#?select()
select(df_event,instnm,univ_id,event_date) %>% head(n=5)
#Use str() to examine object returned by select function
select(df_event,instnm,univ_id,event_date) %>% str()
```

Return values in functions you write

By default, the value returned by a user-written function is the last statement evaluated by the function

- e.g., our `z_score` function **returns** a numeric vector with length equal to length of its input

```r
#create some vector named w
(w=c(NA,seq(1:4),NA))
#> [1] NA  1  2  3  4 NA

#create z-score function
z_score <- function(x) {
  (x - mean(x, na.rm=TRUE))/sd(x, na.rm=TRUE)
}

#call function
z_score(w)
#> [1]         NA -1.1618950 -0.3872983  0.3872983  1.1618950         NA

#use str() to describe the object returned
str(z_score(w))
#>  num [1:6] NA -1.162 -0.387 0.387 1.162 ...
```

Return values in functions you write

By default, the value returned by a user-written function is the last statement evaluated by the function

Let's apply `z_score()` function to variable `med_inc` within `df_event`

```
#apply z_score() to first 5 observations
z_score(df_event$med_inc[1:5])
#> [1] -0.3291617  1.7830646 -0.5205094 -0.5205094 -0.4128839

#apply z_score() to all obs; use str() to examine what is returned
str(z_score(df_event$med_inc))
#>  num [1:18680] -0.4425 0.000821 -0.48266 -0.48266 -0.460071 ...
```

Note: even though `z_score` function returns a numeric vector, data frame is unchanged unless we **assign** new variable

```
#without assignment
df_event %>% mutate(med_inc_z=z_score(med_inc)) %>%
  select(med_inc, med_inc_z) %>% head(n=5)
names(df_event)

#with assignment
df_event <- df_event %>% mutate(med_inc_z=z_score(med_inc))
str(df_event$med_inc_z)
```

Return values in functions you write

By default, the value returned by a user-written function is the last statement evaluated by the function

You can override this default behavior – that is, “choose to return early” – by using the `return()` function

- see Grolemund and Wickham 19.6 for details

Return value and writing “pipeable functions”

“Pipeable functions”

- Functions that can be used within a pipe

GW (chapter 19.6.2) identify two types of pipeable functions:

1. **transformations** “an object is passed to the function’s first argument and a modified object is returned”
    - e.g., all functions from `dplyr` package – `select()`, `filter()`, etc. – are “transformation functions”
2. **side effects** “the passed object is not transformed. Instead, the function performs an action on the object, like drawing a plot”
    - e.g., the `cat()` function

GW recommendation for writing “side effect” type functions:

- you should “invisibly” return the first argument (i.e., input object)
    - “invisibly” means object will not be printed
- Why? input object can still be used within a pipe
- do this using the `invisible()` function

Return value and writing "pipeable functions"

When writing "side effect" functions, which do not create object, use `invisible()` to "invisibly" return an object that is an input to function

- syntax: `invisible(x)` ; where `x` is some object

GW Example: create function that prints number of `NAs` in data frame

```r
mtcars # example data frame included when you install R
str(mtcars)

show_missings <- function(df) {
  n <- sum(is.na(df)) # var that equals sum of NAs in data frame
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df) # returns object associated with argument df
}
show_missings(mtcars) # call function
str(show_missings(mtcars)) # what function returns
```

Because `show_missings` function used `invisible()` to return input data frame, we can use this function in a pipe

```r
mtcars %>% show_missings() %>%
  mutate(mpg_v2 = ifelse(mpg < 20, NA, mpg)) %>% # create var that has NAs
  show_missings()
#> Missing values: 0
#> Missing values: 18
```

Writing functions that humans can understand

Functions are for humans and computers

From Grolemund and Wickham
(http://r4ds.had.co.nz/functions.html#functions-are-for-humans-and-computers)

Functions you write are processed by computers, but important for humans to be able to understand your function too.

Be thoughtful about

- function names
- names of arguments/inputs
- commenting your code
- coding style

Function names

Grolemund and Wickham recommendations:

- functions **perform actions** on **inputs**, so name of function should be verbs name and inputs/arguments should be nouns
  - e.g, we named functions `print_hello` and `count_events`
- But better to name the function a noun if the verb that comes to mind feels too generic
  - e.g., the name `z_score` is better than `calculate_z_score`
- Recommend using "snake_case" to separate words
  - e.g., `print_hello` rather than `print.hello` or `PrintHello`

Commenting code

Grolemund and Wickham recommendations:

> *"Use comments, lines starting with #, to explain the "why" of your code. You generally should avoid comments that explain the "what" or the "how". If you can't understand what the code does from reading it, you should think about how to rewrite it to be more clear"*

Ozan recommendations

- I use comments to explain why
- I also use comments to explain what the code does and/or how it works
    - Writing these comments help me work through each step of a problem
    - These comments help me/others understand code when I return to it after several months