Lecture 9: Writing functions

EDUC 263: Managing and Manipulating Data Using R

Ozan Jaquette

1. Introduction
2. Why and when to write a function
3. Function basics
4. Functions are for humans and computers
5. Conditional execution
6. Function arguments
7. Return values

Introduction

Libraries

```
library(tidyverse)
#> -- Attaching packages ---------------------------------------
#> v ggplot2 3.0.0     v purrr   0.2.5
#> v tibble  1.4.2     v dplyr   0.7.6
#> v tidyr   0.8.1     v stringr 1.3.1
#> v readr   1.1.1     v forcats 0.3.0
#> -- Conflicts ------------------------------------------------
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(haven)
```

Data we will work with

```
#load dataset with one obs per recruiting event
load("../../data/recruiting/recruit_event_somevars.Rdata")

#load dataset with one obs per high school
load("../../data/recruiting/recruit_school_somevars.Rdata")
```

Why and when to write a function

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** (A) a new object or (B) a visualizatoin (e.g., plot)

We've been working with functions all quarter. For example, the `select()` function (type `?select` in console):

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 may be a vector, a data-frame, a plot, etc.

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

The `sum()` function (`?sum` in console):

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

```
sum(c(1,2,3))
sum(c(1,2,3)) %>% str()
```

What are user-written functions

user-written functions [my term] are functions you write to perform some specific task, usually a data-manipulation or analysis task specific to your project

Like all functions, user-written functions 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

Before showing you how to write your own functions, let's see an example of a function in action [FIX]

POTENTIAL EXAMPLES:

- SOME SPECIFIC TABULATION LIKE COUNT OF EVENT BY TYPE FOR EACH UNIVERSITY
- READ IN DATA
- FIX MISSING VALUES
  https://www.btskinner.me/rworkshop/modules/programming_one.html

When should you write a function

Wickham chapter 19 has some practical advice about when to write your own function.

Before stating this, let's introduce a task from the "off-campus recruiting project" that we might want to achieve by writing a function

Task:

- Dataset `df_event` has one observation for each university-recruiting_event for several public universities
  - Variable `event_type` identifies location type of recruiting event (e.g., public high school, private high school)
- We want to 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

```
df_event %>% filter(univ_id==196097) %>% group_by(event_type) %>%
  summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))

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

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

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 the off-campus recruiting descriptive stats, we would copy above code for each university and change the university ID

Grolemund and Wickham say:

> *"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 the way you complete 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 and 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)

Why write functions

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
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., writing a general function that takes-in longitudinal data on number of degrees degrees awarded by field an 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
   - Functions flexible in terms of which models to run, variables to include, etc.

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

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

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

```
#to get help type "?base" in console and scroll index for "function", but help f
function_name <- function(x,y,z) {
  #function body
}
```

Three components of a function:

1. **function name**
   - specify function name before the assignment operator <-
2. **function arguments** (sometimes called "inputs")
   - Inputs that the function takes
     - can be vectors, data frames, logical statements, 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

Our first example is to write a function that simply prints “Hello!”

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 the `print_hello` function so that it also prints our name, which we specify as an input.
- first, perform task outside a function. A few approaches we could take

This seems wrong because my name is not an input

```
"Hello! My name is Ozan Jaquette"
#> [1] "Hello! My name is Ozan Jaquette"
```

Why doesn't this work?

```
x <- "Ozan Jaquette"
x
#> [1] "Ozan Jaquette"
"Hello! My name is x"
#> [1] "Hello! My name is x"
```

Why doesn't this work?

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

This sort of works

```
"Hello! My name is "
#> [1] "Hello! My name is "
x
#> [1] "Ozan Jaquette"
```

Hello function

Task:

- modify the `print_hello` function so that it also prints our name, which we specify as an input.
- first, perform task outside a function. A few approaches we could take

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

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

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 the `print_hello` function so that it also prints our name, which we specify as an input.
- first, perform task outside a function. A few approaches we could take

We need to find an alternative to `print()` that can print multiple objects on the same line

Let's experiment with `cat()` function [I had to Google this]

```
#?cat
cat("Hello! My name is ")
#> Hello! My name is
cat(x)
#> Ozan Jaquette

cat("Hello! My name is ",x)
#> Hello! My name is  Ozan Jaquette
cat("Hello! My name is",x)
#> Hello! My name is Ozan Jaquette
```

Success! Now we can write a function for this task

Hello function

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

Task outside of function

```
x <- "Ozan Jaquette"
cat("Hello! My name is",x)
#> Hello! My name is Ozan Jaquette
```

Create function

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

1. **function name** is `print_hello`
2. **function arguments**. "inputs" to the function
   - `print_hello` function takes one argument, `name`
   - Instead of `name`, we 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

```
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 <- 2018 - 1979
z
#> [1] 39
cat("Hello! My name is",x,". In 2018 I will turn",z,"years old")
#> Hello! My name is Ozan Jaquette . In 2018 I will turn 39 years old
```

Improvements we could make:

- use **date functions** to:
  1. specify current date (rather than manually typing "2018")
  2. calculate age exactly (rather than as current year minus birth year)

  - But we haven't learned date functions, so hold off
- use **string functions** to:
  - remove extra space between name and the period
  - but we haven't learned string functions, so hold off

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

```
cat("Hello! My name is",x,"and in 2018 I will turn",z,"years old")
#> Hello! My name is Ozan Jaquette and in 2018 I will turn 39 years old
```

Create function

```
print_hello <- function(name,birth_year) {
  age <- 2018 - birth_year
  cat("Hello! My name is",name,"and in 2018 I will turn",age,"years old")
}
```

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 <- 2018 - birth_year`
   - `cat("Hello! My name is",name,"and in 2018 I will turn",age,"years old")`

Call function

```
print_hello("Ozan Jaquette",1979)
#> Hello! My name is Ozan Jaquette and in 2018 I will turn 39 years old
```

Recipe for writing 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

Practice: the `z_score` function

`z_score` function

The z-score for observation *i* is number of standard deviations from mean:

$z_i = \frac{x_i - \bar{x}}{sd(x)}$

Task:

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

Let's create a vector of numbers

```
v=c(seq(5,15))
v
length(v)
v[1]
v[10]
```

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

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

Create data frame

```
set.seed(12345) # 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)),
  d = c(NA,rnorm(9))
)
df
df
class(df)

df$a
str(df$a)
```

Apply `z_score` function to variables in data frame

```
mean(df$a, na.rm=TRUE)
df$a
z_score(df$a)

z_score(df$b)
```

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

We can use our function to create a new variable that is the z-score version of a variable

Base R approach

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

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

#assign new variable [base R approach to creating new variables]
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
```

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

We can use our function to create a new variable that is the z-score version of a variable

Tidyverse approach

```
#same function as before
z_score <- function(x) {
  (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
df <- df %>% mutate(
  a_z = z_score(a),
  c_z = z_score(c)
  )
names(df)
#> [1] "a"   "b"   "c"   "d"   "c_z" "a_z"
```

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

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

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

df_event_small %>% mutate(
    med_inc_z = z_score(med_inc))
#> # A tibble: 10 x 5
#>    instnm      univ_id event_type med_inc med_inc_z
#>    <chr>         <int> <fct>        <dbl>     <dbl>
#>  1 UM Amherst   166629 public hs   71714.   -0.608
#>  2 UM Amherst   166629 public hs   89122.    0.920
#>  3 UM Amherst   166629 public hs   70136.   -0.747
#>  4 UM Amherst   166629 public hs   70136.   -0.747
#>  5 USCC         218663 private hs  71024.   -0.669
#>  6 UM Amherst   166629 private hs  71024.   -0.669
#>  7 Stony Brook  196097 public hs   71024.   -0.669
#>  8 UM Amherst   166629 private hs  97225     1.63
#>  9 UM Amherst   166629 public hs   97225     1.63
#> 10 UM Amherst   166629 public hs   77800.   -0.0740
```

Practice: the `count_events` function

`count_events` function

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

The dataset `df_event` has one obs for each university-recruiting_event

- Variable `event_type` identifies location type of recruiting event (e.g., public high school, private high school)

Task: ceate the following 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

Before writing function, we perform task outside a function. But first, we should figure out how we can identify each university

```
names(df_event)
#looks like "univ_id" is id var associated with each public university
df_event %>% count(univ_id)
#"instnm" is the name of each public university
df_event %>% count(instnm)
#need to identify univ_id value assoicated with each university name
df_event %>% select(instnm,univ_id) %>% group_by(univ_id) %>%
  filter(row_number()==1) %>% arrange(univ_id)
```

`count_events` function

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

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

Create "by event-type" table outside function

```
df_event %>% count(event_type)

#For all public universities
df_event %>% group_by(event_type) %>% summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))

#For a particular public university; e/g/. U. Arkansas is univ_id==106397
df_event %>% filter(univ_id==106397) %>% group_by(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
2. event-type and whether event is in-state/out-of-state

Create "by event-type and in/out of state" table outside function

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

#For all public universities
df_event %>% group_by(event_inst,event_type) %>% summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))

#For a particular public university; e/g/. U. Arkansas is 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
2. event-type and whether event is in-state/out-of-state

Create function

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

`count_events` function

Success! But do we like this function?

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

I don't like that there is a lot of repeated code within function.

- Function should be able to create the `group_by(event_type)` table and/or the `group_by(event_inst,event_type)` table without repeating code

What improvements should we make?

1. specify `group_by()` variables in function call
2. specify which variable(s) to calculate mean of in function call
3. specify which statistic(s) we want to calculate (e.g., mean, median, max)
4. A title with university name and other info to make table readable

Turns out that several of these improvements require additional programming concepts that we have not introduced yet

Student tasks

Student task: `fix_missing` function

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

A common task when working with survey data is to replace negative values with `NA`

Let's create a sample dataset with some negative values

```
#> # 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 task: `fix_missing` function

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

A common task when working with survey data is to replace negative values with `NA`

Task:

- count the number of observations with negative values for a specific variable

Hints:

Student task: `num_missing` function [SOLUTION]

Task:

- count the number of observations with negative values for a specific variable

Perform task outside of function

```
names(df)
#> [1] "id"     "age"    "sibage" "parage"

df$age
#>   [1]  17  15 -97  13 -97  12 -99 -97  16  16 -98  20 -99  20  11  20  12
#>  [18]  17  19  17 -97 -99  12  13  11  15  20  14 -99  11  20 -98  11 -98
#>  [35]  12  16  12  18  12  19  12 -97  20  17  11  19  19  12 -98  11  15
#>  [52]  18  15 -98  15  19 -97  13 -98  16  13  12  16  19 -99  19 -98  13
#>  [69] -97  20  15  19  15  12  18 -99  18 -98 -98 -98 -97  12  14  19 -97
#>  [86]  11  20  18  14 -99  15  20 -97  14  14  19  18  17  20  15

sum(df$age<0)
#> [1] 27
```

Write function

```
num_missing <- function(x){

  sum(x<0)
}

num_missing(df$age)
```

Student task: `num_missing` function

Task:

- modify function, so that function includes a second argument where you specify values associated with missing

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)
}

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
```

Student task: `prop_missing` function

Task:

- modify function, so that function includes a second argument where you specify values associated with missing and calculates the proportion missing

Write function

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

  mean(x %in% miss_vals)
}

prop_missing(df$age,c(-97,-98,-99))
#> [1] 0.27
prop_missing(df$sibage,c(-97,-98,-99))
#> [1] 0.22
prop_missing(df$parage,c(-4,-7,-8))
#> [1] 0.17
```

Student task: `fix_missing` function [PUT BEN'S TASK LATER?]

Perform task outside of function

```
df_temp <- df
names(df_temp)
#> [1] "id"      "age"     "sibage" "parage"

df_temp %>% mutate(
  age_na = ifelse(age %in% c(-97,-98,-99),NA,age)
) %>% select(age,age_na)
#> # A tibble: 100 x 2
#>      age age_na
#>    <dbl>  <dbl>
#>  1    17     17
#>  2    15     15
#>  3   -97     NA
#>  4    13     13
#>  5   -97     NA
#>  6    12     12
#>  7   -99     NA
#>  8   -97     NA
#>  9    16     16
#> 10    16     16
#> # ... with 90 more rows
```

Write function

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

Functions are for humans and computers

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 careful 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 of 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

Coding style

PUT THIS LATER?

Conditional execution

Conditional execution

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

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

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

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

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

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

Conditional execution

Example

- Imagine some administrative software program that sends students an email about their cumulative GPA and whether they are on academic probation

```
email_gpa <- function(gpa) {

  if (gpa<2) {

    cat("All students with a GPA below 2.0 are on academic probation. Your GPA i

  } else {
    cat("Your GPA is",gpa,"and you are not on academic probation.")
  }

}
email_gpa(1.9)
#> All students with a GPA below 2.0 are on academic probation. Your GPA is 1.9
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`

```
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 false"
  }
}

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

The `condition` can have multiple logical expressions as long as it evaluates to either `TRUE` or `FALSE`

Use `||` (or) and `&&` (and) to combine multiple logical expressions

```
go_to_daycare <- function(weekday,temp) {

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

Make sure to use `||` (or) `&&` (and) to combine conditions; don't use `|` (or), `&` (and)

> *"You should never use | or & in an if statement: these are **vectorised** operations that apply to multiple values (that's why you use them in filter())"*

Multiple conditions

Can chain multiple `if` statements together you want to specify code for more than two conditions

```
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
}
```

Example: create text for email to students about their GPA

```
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?

IS THERE TIME TO GO OVER THIS? OR JUST POINT STUDENTS TO THIS

Conditional execution: practical example

IS THERE TIME FOR THIS? IF SO, ASK PATRICIA TO HELP THINK OF AN EXAMPLE

Function arguments

Types of arguments

Grolemund and Wickham broadly distinguish between two types of function arguments:

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

Grolemund and Wickham recommendations about order of arguments:

- 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

Example: the 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 a function, specify default values for an argument the same way you would specify values for that argument when calling the function

```
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

CUT THIS OR PUT IN AN EASIER EXAMPLE

```
letters[1:10]
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
commas <- function(...) {
  stringr::str_c(..., collapse = ", ")
}
commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"
letters[1:10]
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

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

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

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

Want to revise our function so to specify `group_by()` variables at program call

- Problem: 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

Solution

```
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")
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
   - Program body `group_by_(...)` is executed as `group_by_(event_inst,event_type)`

Note: `group_by_()` differs from `group_by()`

- Explanation HERE

Return values

Return values

The **return value** of a function is the object created ("returned") after the function runs

- this could be a vector, a list, a data frame, etc

In the help-file for any function, the section **Value** describes return value for that function

- e.g., the `sum()` function

```
?sum
```

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

```
w=c(NA,seq(1:5),NA)
w
#> [1] NA  1  2  3  4  5 NA
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
str(z_score(w)) # describe the object returned
#>  num [1:7] NA -1.265 -0.632 0 0.632 ...
```

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 pipeable functions

See (http://r4ds.had.co.nz/functions.html#writing-pipeable-functions)

Two types of pipeable functions, transformations and side-effects

- **transformations** "an object is passed to the function's first argument and a modified object is returned"
- **side effects** "the passed object is not transformed. Instead, the function performs an action on the object, like drawing a plot or saving a file"

If you are writing a side-effect type function, you should "invisibly" return the first argument (i.e., input object) so that this object is not printed but it can still be used in a pipeline

- do this using the `invisible()` function

Example from Grolemund and Wickham

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}
```

We can still use this in a pipe

```
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
#> Missing values: 0
```

Practice: the `read_admit` function

`read_admit` function

In previous lecture on appending/stacking data, we read in several annual IPEDS datasets on admissions characteristics

```
#read in data
admit_16_17 <- read_dta(file="../../data/ipeds/ic/ic16_17_admit.dta") %>%
  select(unitid,endyear,sector,contains("admcon"),
         contains("numapply"),contains("numadmit"))
admit_15_16 <- read_dta(file="../../data/ipeds/ic/ic15_16_admit.dta") %>%
  select(unitid,endyear,sector,contains("admcon"),
         contains("numapply"),contains("numadmit"))
admit_14_15 <- read_dta(file="../../data/ipeds/ic/ic14_15_admit.dta") %>%
  select(unitid,endyear,sector,contains("admcon"),
         contains("numapply"),contains("numadmit"))

#append data
admit_append <- bind_rows(admit_16_17,admit_15_16,admit_14_15)
```

Lots of repeated code here. and often, I read in ten to twenty years of annual data. Can we write a function to read in annual data?

`read_admit` function

Task:

- write a function to read in annual IPEDS admissions data

We already performed the task outside of a function, so let's try writing a function

```
read_admit <- function(ayear) {
  admit_ayear <- read_dta(file="../../data/ipeds/ic/ayear_admit.dta") %>%
  select(unitid,endyear,sector,contains("admcon"),
         contains("numapply"),contains("numadmit"))
}
read_admit(ayear=16_17)
#rm(list = ls())
```

What went wrong?

1. in assignment statement `admit_ayear <-`, we wanted text "`16_17`" to be substituted for text "ayear"
2. in file-path, we wanted the text "`16_17`" to be substituted for text "ayear"

Before attempting to revise function, let's develop an approach for completing this task outside a function that is more conducive being placed in the body of a function

`read_admit` function

Task: write a function to read in annual IPEDS admissions data

Perform task outside of a function.

- First, create an object for desired name for data frame (admit_16_17)

```
x <- "16_17" #create an object for academic year
x
#> [1] "16_17"

#use cat() to create object w/ desired name for data frame
cat("admit_",x)  #we need to remove spaces
#> admit_ 16_17
#?cat
cat("admit_",x, sep="")
#> admit_16_17

dfname <- cat("admit_",x, sep="")
#> admit_16_17
dfname  #problem is that cat() function can't be used for variable assignment be
#> NULL

#use paste() to create object w/ desired name for data frame
paste("admit_",x, sep="")
#> [1] "admit_16_17"
dfname <- paste("admit_",x, sep="")
dfname
#> [1] "admit_16_17"
```

`read_admit` function

Task: write a function to read in annual IPEDS admissions data

Perform task outside of a function.

- Next, let's try to a different way to specify file-path and name of the dataset as objects

```
#name of the dataset we are reading, eg ic16_17_admit.dta
paste("ic",x,"_admit.dta",sep="")
#> [1] "ic16_17_admit.dta"
dtaname=paste("ic",x,"_admit.dta",sep="")
dtaname
#> [1] "ic16_17_admit.dta"

#Create object for filepath, eg. "../../data/ipeds/ic/ic15_16_admit.dta"
  #?file.path # this function creates a file path
dir <- file.path("..","..","data","ipeds","ic")
dir
#> [1] "../../data/ipeds/ic"
setwd(dir)
list.files()
#> [1] "ic14_15_admit.dta"       "ic15_16_admit.dta"
#> [3] "ic16_17_admit.dta"       "ic2017ay_small.csv"
#> [5] "ipeds_hd_2017_small.csv" "text"
```

`read_admit` function

Task: write a function to read in annual IPEDS admissions data

Perform task outside of a function.

- Now, we can put it all together

```
#rm(list = ls())
x <- "16_17" # academic year
dfname <- paste("admit_",x, sep="") # name we want to assign to data frame
dtaname=paste("ic",x,"_admit.dta",sep="") # name of Stata dataset we are reading
dir <- file.path("..","..","data","ipeds","ic") # name of filepath to Stata data

#read data
getwd()
#> [1] "C:/Users/ozanj/Documents/rclass/lectures/lecture9"
setwd(dir)
df <- read_dta(file=dtaname) %>% # read in data and assign default name df
  select(unitid,endyear,sector,contains("admcon"),
         contains("numapply"),contains("numadmit"))

assign(dfname,df) # create new object that has values of df; object name is valu
rm(df)
```

read_admit function

Task: write a function to read in annual IPEDS admissions data

Now we can create function for task

```
#rm(list = ls())
read_admit <- function(ayear) {

  dfname <- paste("admit_",ayear, sep="") # name we want to assign to data frame
  dtaname=paste("ic",ayear,"_admit.dta",sep="") # name of Stata dataset we are r
  dir <- file.path("..","..","data","ipeds","ic") # name of filepath to Stata da

  #read data
  getwd()
  setwd(dir)
  df <- read_dta(file=dtaname) %>% # read in data and assign default name df
    select(unitid,endyear,sector,contains("admcon"),
           contains("numapply"),contains("numadmit"))

  #default is that the return value is the last statement executed so we don't n
  return(df)
}

admit_16_17 <- read_admit(ayear="16_17")
admit_15_16 <- read_admit(ayear="15_16")
admit_14_15 <- read_admit(ayear="14_15")

#append data
```