Lecture 10: Accessing object elements and looping

EDUC 263: Managing and Manipulating Data Using R

Ozan Jaquette

1 Introduction

Logistics

Course evaluations

Please take a few minutes to complete instructor and T.A. evaluations

- Complete online evaluation for Ozan
  - Should have received an email or log on to your MyUCLA account
- Complete online evaluation for Patricia (TA)
  - Link here

Problem set 10

- Due next Friday at noon; focuses on accessing vector elements and basics of looping
- We give big hints; less emphasis on "figure it out on your own"

Reading:

- Grolemund and Wickham 20.4 - 20.5 (Chapter 20 is on "Vectors")
- Grolemund and Wickham 21.1 - 21.3 (Chapter 21 is on "Iteration")
- [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

What we will do today

Libraries and data

```
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()
```

Data frame

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

2 Accessing elements of vectors and lists

2.1 Review: Types of vectors

Review: Types of vectors

Recall that there are two broad types of vectors, **atomic vectors** and **lists**

1. **Atomic vectors**. There are six types:
   - logical, integer, double, character, complex, and raw
2. **lists**. "sometimes called recursive vectors lists can contain other lists"

Main difference between atomic vectors and lists:

- atomic vectors are "homogenous," meaning each element in vector must have same type (e.g., integer, logical, character)
- lists are "heterogeneous," meaning that data type can differ across elements within a list

Link to figure of data structures overview HERE

Review: Types of atomic vectors

1. logical. each element can be three potential values: `TRUE`, `FALSE`, `NA`

```
typeof(c(TRUE,FALSE,NA))
#> [1] "logical"
typeof(c(1==1,1==2))
#> [1] "logical"
```

2. Numeric (integer or double)

```
typeof(c(1.5,2,1))
#> [1] "double"
typeof(c(1,2,1))
#> [1] "double"
```

   - Numbers are doubles by default. To make integer, place `L` after number:

```
typeof(c(1L,2L,1L))
#> [1] "integer"
```

3. character

```
typeof(c("element of character vector","another element"))
#> [1] "character"
length(c("element of character vector","another element"))
#> [1] 2
```

`TRUE/FALSE` functions that identify **type** of vector

Identifying vector **type**, Grolemund and Wickham:

- "Sometimes you want to do different things based on the type of vector. One option is to use `typeof()`. Another is to use a test function which returns a `TRUE` or `FALSE`"

`is_*()` functions are provided by `purrr` package within `Tidyverse`:

Function	logical	int	dbl	chr	list
`is_logical()`	X				
`is_integer()`		X			
`is_double()`			X		
`is_numeric()`		X	X		
`is_character()`				X	
`is_atomic()`	X	X	X	X	
`is_list()`					X
`is_vector()`	X	X	X	X	X

```
is_numeric(c(5,6,7))
#> Warning: Deprecated
#> [1] TRUE
```

TRUE/FALSE functions that identify **type** of vector

Recall that elements of a vector must have the same type

- If vector contains elements of different type, type will be most "complex"
- From simplest to most complex: logical, integer, double, character

```
is_logical(c(TRUE,TRUE,NA))
is_logical(c(TRUE,TRUE,NA,1))

typeof(c(TRUE,1L))
is_integer(c(TRUE,1L))

typeof(c(TRUE,1L,1.5,"b"))
is_character(c(TRUE,1L,1.5,"b"))
```

`TRUE/FALSE` functions that identify **type** vs. **class** of vector

Comparing `is_*()` vs. `is.*()` functions

- `is_*()` functions (e.g., `is_numeric()`) identifies vector **type**
  - They are the `TRUE/FALSE` versions of `typeof()` function
- `is.*()` functions (e.g., `is.numeric()`) refer to both **type** and **class**
  - Review: **class** is an object **attribute** that defines how object can be treated by object oriented programming language (e.g., which functions you can apply)
  - Recall that R functions care about **class**, not **type**

```
df_event %>% select(instnm,univ_id,event_date,med_inc,titlei_status_pub) %>% str
```

Variable = `univ_id`

```
typeof(df_event$univ_id)
class(df_event$univ_id)
is_numeric(df_event$univ_id)
is.numeric(df_event$univ_id)
```

Variable = `event_date`

```
typeof(df_event$event_date)
class(df_event$event_date)
is_numeric(df_event$event_date)
is.numeric(df_event$event_date)
```

`TRUE/FALSE` functions that identify **type** vs. **class** of vector

Comparing `is_*()` vs. `is.*()` functions

- `is_*()` functions (e.g., `is_numeric()`) identifies vector **type**
- `is.*()` functions (e.g., `is.numeric()`) refer to both **type** and **class**

Variable = `med_inc`

```
typeof(df_event$med_inc)
class(df_event$med_inc)
is_numeric(df_event$med_inc)
is.numeric(df_event$med_inc)
```

Variable = `titlei_status_pub`

```
typeof(df_event$titlei_status_pub)
class(df_event$titlei_status_pub)
is_numeric(df_event$titlei_status_pub)
is.numeric(df_event$titlei_status_pub)
```

2.2 Accessing elements of (atomic) vectors

Subsetting elements of vector

"Subsetting" a vector, refers to isolating particular elements of a vector

- I sometimes refer to this as "accessing elements of a vector"
- subsestting elements of a vector is similar to "filtering" rows of a data-frame

`[]` is the subsetting function for vectors

- contents inside `[]` can refer to element number (also called "position").
  - e.g., `[3]` refers to contents of 3rd element (or position 3)
- contents inside `[]` can also refer to name of element
  - e.g., `["a"]` refers to contents inside an element named "a"

Subsetting elements of vector, based on position number

- `[]` is the subsetting function for vectors
  - contents inside `[]` can refer to element number (also called "position").
    - e.g., `[3]` refers to contents of 3rd element (or position 3)

Examples of referring to elements based on element position number

```
x <- c("a","b","c","d","e")
x # all elements
#> [1] "a" "b" "c" "d" "e"

x[1] # 1st element
#> [1] "a"
x[5] # 5th element
#> [1] "e"

c(x[1],x[2],x[2]) # 1st, 2nd, and 2nd element
#> [1] "a" "b" "b"

x[c(1,2,2)] # 1st, 2nd, and 2nd element
#> [1] "a" "b" "b"
```

Subsetting elements of vector, based on position number

Examples of referring to elements based on position number, continued

```
y <- c(4,5,10,29,15,12)
length(y)
#> [1] 6

y[c(1,3,6)]
#> [1]  4 10 12
y[c(3,6,1)]
#> [1] 10 12  4
```

While subsetting with positive numbers keeps elements in those positions, subsetting with negative numbers drops elements at those positions

```
y
#> [1]  4  5 10 29 15 12

y[c(-3,-4,-5,-6)] # show elements except 3rd, 4th, 5th and 6th elements
#> [1] 4 5
```

Subsetting elements of named vector, by element name

Review: naming vectors

- All vectors can be “named” (i.e., you name individual elements within the vector)

Unnamed vector

```
x <- c(1,2,3,"hi!")
x
#> [1] "1"   "2"   "3"   "hi!"
str(x)
#>  chr [1:4] "1" "2" "3" "hi!"
```

Named vector

```
y <- c(a=1,b=2,3,c="hi!")
y
#>     a     b           c
#>   "1"   "2"   "3" "hi!"
str(y)
#>  Named chr [1:4] "1" "2" "3" "hi!"
#>  - attr(*, "names")= chr [1:4] "a" "b" "" "c"
```

Subsetting elements of named vector, by element name

If you have a **named vector**, you can subset it with a character vector:

- i.e., access **element values** based on **element names**

```
x <- c(abc = 1, def = 2, xyz = 5)
x
#> abc def xyz
#>   1   2   5

x["xyz"] # show value of element named "xyz"
#> xyz
#>   5

x[c("xyz", "def")] # show value of element named "xyz" and element named "def"
#> xyz def
#>   5   2
```

Subsetting elements of vector, with a logical vector

Subsetting elements with a logical vector

- i.e., access elements that satisfy some `TRUE/FALSE` condition

```
(x <- c(10, 3, NA, 5, 8, 1, NA,"Hi!"))
#> [1] "10"  "3"   NA    "5"   "8"   "1"   NA    "Hi!"

typeof(x)
#> [1] "character"

x[is_character(x)] # since vector type is character, all elements are character
#> [1] "10"  "3"   NA    "5"   "8"   "1"   NA    "Hi!"
x[is_numeric(x)] # since vector type is character, no elements are numeric
#> character(0)

(y <- c(10, 3, NA, 5, 8, 1, NA))
#> [1] 10  3 NA  5  8  1 NA

typeof(y)
#> [1] "double"
y[is_numeric(y)]
#> [1] 10  3 NA  5  8  1 NA
```

2.3 Accessing elements of lists

Review: Lists

Like atomic vectors, lists are objects that contain elements. However:

- “type” of elements can vary within a list
- elements of a list can contain another list

Examples:

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

`str()` function is helpful for understanding structure and contents of a list

```
str(x1)
#> List of 2
#>  $ : num [1:2] 1 2
#>  $ : num [1:2] 3 4
str(x2)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ :List of 2
#>   ..$ : num 3
#>   ..$ : num 4
```

Review: Data frames are lists

Recall the relationship between "lists" and "data frames"

- data frames have "type==list"
- data frames are lists with these additional structure requirements
  - each element of data frame must be a vector (not a list)
  - each element (i.e., vector) in data frame must have the same length
- data frames have additional attributes (e.g., each vector is named)

```
(df <- tibble(x = 1:3, y = 3:1))
#> # A tibble: 3 x 2
#>       x     y
#>   <int> <int>
#> 1     1     3
#> 2     2     2
#> 3     3     1
typeof(df)
#> [1] "list"
str(df)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: int  3 2 1

typeof(df_event)
#> [1] "list"
```

Subsetting/accessing elements of a list

Accessing elements of a list is important for writing loops, writing functions, and many other applications in R

I will demonstrate accessing elements of a list using two lists:

1. A list that has more complicated structure than a data frame (from Grolemund and Wickham example)

```
list_a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
typeof(list_a)
str(list_a)
```

2. List that is 7 variables and first 5 obs of `df_event`, corresponding to University of Alabama

```
df_bama <- df_event %>% arrange(univ_id,event_date) %>%
  select(instnm,univ_id,event_date,event_type,event_state,zip,med_inc) %>%
  filter(row_number()<6)

typeof(df_bama)
str(df_bama)
```

Subsetting/accessing elements of a list

Accessing elements of a list is important for writing loops, writing functions, and many other applications in R

GW 20.5.2 ("Subsetting"): 3 ways to "subset" (i.e., access elements of) list

1. `[]` "extracts a sub-list. The result will always be a list"
   - like subsetting vectors, you can subset with a logical, integer, or character vector
2. `[[]]` "extracts a single component from a list. It removes a level of hierarchy from the list"
3. `$` "shorthand for extracting named elements of a list. It works similarly to `[[]]` except that you don't need to use quotes."

Subset a list using `[]`

`[]` "extracts a sub-list"

- contents of `[]` can be position number, name of element in list, logical vector, etc.

```
str(list_a)
length(list_a)

list_a[1] # extract first element of list
str(list_a[1]) # extract first element of list
str(list_a["a"]) # extract element named "a"

str(list_a[1:2]) # extract first two elements of list
str(list_a[c(1,2)]) # extract first two elements of list

str(list_a[c("a","c")]) # extract element named "a" and element named "c"
```

Key takeaway about subsetting a list using `[]`: **The result will always be a list**

- that is, `[]` does not remove a level of hierarchy
- structure and attributes of object you isolate using `[]` will be the same as its structure and attributes in the list it is taken from

Subset a list using `[]`: Student task

Applying `[]` to the object `df_bama`:

- Isolate the 1st element of `df_bama`
- Isolate the 3rd through 5th element of `df_bama`
- Isolate the 3rd, 7th, and 1st element of `df_bama`
- Isolate the element named `"event_type"`
- Isolate the elements named `"event_type"` and `"med_inc`

Subset a list using `[]` : Student task [SOLUTIONS]

Applying `[]` to the object `df_bama` :

```
#- Isolate the 1st element of `df_bama`
df_bama[1]
str(df_bama[1])
#- Isolate the 3rd through 5th element of `df_bama`
df_bama[3:5]
str(df_bama[3:5])
#- Isolate the 3rd, 7th, and 1st element of `df_bama`
df_bama[c(3,7,1)]
#- Isolate the element named `"event_type"`
df_bama["event_type"]
str(df_bama["event_type"])
#- Isolate the elements named `"event_type"` and `"med_inc`
df_bama[c("event_type","med_inc")]
```

Subset a list using `[[]]`

GW: " `[[]]` extracts a single component from a list." More specifically, `[[]]` :

1. Extracts a **single** element of the list **AND**
2. Removes a "level of hierarchy"" from the list

```
str(list_a)

str(list_a[1]) # []; result is a one-element list [length=1]; this list contains
str(list_a[[1]]) # [[]]; result is a numeric vector with 3 elements

str(list_a["a"]) # []; result is a one-element list [length=1]; this list contai
str(list_a[["a"]]) # [[]]; result is a numeric vector with 3 elements

str(list_a[4]) # []; result is a one-element list, which contains a two-element
str(list_a[[4]]) # [[]]; result is a two-element list
```

Subset a list using `[[]]`, data frames

Comparing `[]` to `[[]]` when working with lists that are data frames

- Data frame object always has type=list and each element is a vector
- If you subset using `[]` the result will always have type==list
- If you subset using `[[]]` the result will always have type==vector

`[]` vs. `[[]]`: Subsetting data frame using **element position number**

```
df_bama[1]
df_bama[[1]]

str(df_bama[1])
str(df_bama[[1]])

typeof(df_bama[1])
typeof(df_bama[[1]])

class(df_bama[1])
class(df_bama[[1]])

attributes(df_bama[3])
attributes(df_bama[[3]])
```

Subset a list using `[[]]`, data frames

Comparing `[]` to `[[]]` when working with lists that are data frames

- If you subset using `[]` the result will always have type==list
- If you subset using `[[]]` the result will always have type==vector

`[]` vs. `[[]]`: Subsetting data frame using **element name** (i.e., variable name)

- note: whether using `[]` or `[[]]`, element name must be in quotes

```
str(df_bama["event_type"]) # a "tibble" data frame with one variable
str(df_bama[["event_type"]]) # a character vector with 5 elements

attributes(df_bama["event_type"]) # contains attributes (e.g., variiable name)
attributes(df_bama[["event_type"]]) # no attributes; just the data

str(df_bama["event_date"]) # a "tibble" data frame with one variable
str(df_bama[["event_date"]]) # a numeric "date" vector with 5 elements

attributes(df_bama["event_date"]) # attributes of the data frame
attributes(df_bama[["event_date"]]) # class=date
```

Subset a list using `$`

`$` is a shorthand for extracting **named** elements of a list.

- works similarly to `[[]]` except that you don't need to use quotes.
- Like `[[]]`, subsetting using `$` removes a level of hierarchy

Note: we have been using this method of subsetting variables in a data frame all quarter!

```
str(list_a)

list_a["a"] # list of one element, which contains integer vector of 3 elements
list_a[["a"]] # integer vector of 3 elements
list_a$a # integer vector of 3 elements
```

These two approaches yield the same result:

```
df_bama[["med_inc"]]
#> [1]  77380  39134  38272  89203 127972
df_bama$med_inc
#> [1]  77380  39134  38272  89203 127972
```

Extracting multiple elements of a list using `[[]]` and `$`

Each instance of `[[]]` or `$` can only extract a **single** element from the list

Using `[[]]` to extract multiple elements of list

```
c(df_bama[["med_inc"]],df_bama[["event_type"]])
#>  [1] "77380"        "39134"        "38272"        "89203"        "127972"
#>  [6] "private hs"  "2yr college" "other"        "private hs"  "public hs"
```

Using `$` to extract multiple elements of list

```
c(df_bama$med_inc,df_bama$event_type)
#>  [1] "77380"        "39134"        "38272"        "89203"        "127972"
#>  [6] "private hs"  "2yr college" "other"        "private hs"  "public hs"
```

By contrast, `[]` can extract multiple elements within each instance of `[]`

```
str(df_bama[c("instnm","med_inc")]) # "tibble" data frame with two variables

#df_bama[[c("instnm","med_inc")]] # this code will yield an error
```

Subset a vector using `[[]]`: Student task

Applying `[[]]` to the object `df_bama`:

- Isolate the 5th element of `df_bama`
- Isolate the element `"event_type"` of `df_bama`
- Isolate the element `"zip"` of `df_bama` using `$`
- Isolate the elements named `"event_date"` and `"event_type"`

Subset a vector using `[]` : Student task [SOLUTIONS]

Applying `[[]]` to the object `df_bama` :

```
#- Isolate the 5th element of `df_bama`
df_bama[[5]]
str(df_bama[[5]])
#- Isolate the element `"event_type"` of `df_bama`
df_bama[["event_type"]]
str(df_bama[["event_type"]])
#- Isolate the element `"zip"` of `df_bama` using `$`
df_bama$zip
#- Isolate the elements named `"event_date"` and `"event_type"`
c(df_bama[["event_date"]],df_bama[["event_type"]])
#> Warning in as.POSIXlt.Date(x): NAs introduced by coercion
str(c(df_bama[["event_date"]],df_bama[["event_type"]]))
#> Warning in as.POSIXlt.Date(x): NAs introduced by coercion
```

2.4 Review key concepts for loops

Sequences

(Loose) definition

- a sequence is a list of numbers in ascending or descending order

Creating sequences using colon operator

```
-5:5
#>  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
5:-5
#>  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Creating sequences using `seq()` function

- basic syntax:

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

- examples:

```
seq(10,15)
#> [1] 10 11 12 13 14 15
seq(from=10,to=15,by=1)
#> [1] 10 11 12 13 14 15
seq(from=100,to=150,by=10)
#> [1] 100 110 120 130 140 150
```

Length of atomic vectors

Definition: **length** of an object is its number of elements

Length of vectors, using `length()` function

```
x <- c(1,2,3,4,"ha ha"); length(x)
#> [1] 5
y <- seq(1,10); length(y)
#> [1] 10
z <- c(seq(1,10),"ho ho"); length(z)
#> [1] 11
```

Once vector length known, isolate element contents based on position number using `[]`

```
x[5]
#> [1] "ha ha"
z[1]
#> [1] "1"
```

For atomic vectors, applying `[[]]` to vector gives same result as `[]`

```
x[[5]]
#> [1] "ha ha"
z[[1]]
#> [1] "1"
```

Length of lists

Definition: **length** of an object is its number of elements

```
typeof(df_bama); length(df_bama)
#> [1] "list"
#> [1] 7
```

Once list length known, isolate element contents based on position number using `[]` or `[[]]`

- subset one element of list with `[]` yields list w/ length==1

```
typeof(df_bama[7]); length(df_bama[7])
#> [1] "list"
#> [1] 1
```

- subset one element of list with `[[]]` yields vector w length== # rows

```
df_bama[[7]]; typeof(df_bama[[7]]); length(df_bama[[7]])
#> [1]  77380  39134  38272  89203 127972
#> [1] "double"
#> [1] 5
```

subset one element of list with `$` is same as `[[]]`

```
df_bama$med_inc; typeof(df_bama$med_inc); length(df_bama$med_inc)
#> [1]  77380  39134  38272  89203 127972
#> [1] "double"
#> [1] 5
```

Combine sequences and length

When writing loops, very common to create a sequence from `1` to the length (i.e., number of elements) of an object

Here, we do this with a vector object

```
(x <- c("a","b","c","d","e"))
#> [1] "a" "b" "c" "d" "e"
length(x)
#> [1] 5

1:length(x)
#> [1] 1 2 3 4 5
seq(from=1,to=length(x),by=1)
#> [1] 1 2 3 4 5
```

Can do same thing with list object

```
length(df_bama)
#> [1] 7

1:length(df_bama)
#> [1] 1 2 3 4 5 6 7
seq(2,length(df_bama))
#> [1] 2 3 4 5 6 7
```

3 Loop basics

Simple loop example

What are loops?: **Loops** execute some set of commands multiple times

- We build loops using the `for()` function
- Each time the loop executes the set of commands is an **iteration**
- The below loop iterates 4 times

Example

- Create loop that prints each value of vector `c(1,2,3,4)`, one at a time

```
c(1,2,3,4)
#> [1] 1 2 3 4

for(i in c(1,2,3,4)) { # Loop sequence
  print(i) # Loop body
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
```

I use loops to perform practical tasks more efficienlty (e.g., read in data)

- But we'll introduce loop concepts by doing things that aren't very useful

Components of a loop

```
for(i in c(1,2,3,4)) { # Loop sequence
  print(i) # Loop body
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
```

Components of a loop

1. **Sequence**. Determines what to "loop over" (e.g., from 1 to 4 by 1)
   - sequence in above loop is `for(i in c(1,2,3,4))`
   - this creates a temporary/local object named `i`; could name it anything
     - `i` will no longer exist after the loop is finished running
   - each iteration of loop will assign a different value to `i`
   - c(1,2,3,4) is the set of values that will be assigned to `i`
     - in first iteration, value of `i` is `1`
     - in second iteration, value of `i` is `2`, etc.
2. **Body**. What commands to execute for each iteration through the loop
   - Body in above loop is `print(i)`
   - Each time (i.e., iteration) through the loop, body prints the value of object `i`

Using `cat()` to print value of sequence var for each iteration

When building a loop, I always include a line like `cat("z=",z, fill=TRUE)` **to help me understand what loop is doing**

Below two loops are essentially the same; I prefer second approach. Why?:

- Writing name of sequence var object (here `z`) and seeing value of sequence var object for each iteration helps me understand loop better

```
for(z in c(1,2,3)) { # Loop sequence
  print(z) # Loop body
}
#> [1] 1
#> [1] 2
#> [1] 3
for(z in c(1,2,3)) { # Loop sequence
  cat("object z=",z, fill=TRUE) # "fill=TRUE" forces line break after each itera
}
#> object z= 1
#> object z= 2
#> object z= 3
```

Without `fill=TRUE` [not recommended]

```
for(z in c(1,2,3)) { # Loop sequence
  cat("object z=",z) # "Loop body
}
#> object z= 1object z= 2object z= 3
```

Components of a loop

Note that these three loops all do the same thing

- **Loop body** is the same in each loop
- **Loop sequence** written slightly differently in each loop

```
for(z in c(1,2,3)) { # Loop sequence
  cat("object z=",z, fill=TRUE) # Loop body
}
#> object z= 1
#> object z= 2
#> object z= 3

for(z in 1:3) { # Loop sequence
  cat("object z=",z, fill=TRUE) # Loop body
}
#> object z= 1
#> object z= 2
#> object z= 3

num_sequence <- 1:3
for(z in num_sequence) { # Loop sequence
  cat("object z=",z, fill=TRUE) # Loop body
}
#> object z= 1
#> object z= 2
#> object z= 3
```

Student exercise

Try on your own or just follow along.

Task

1. Create a numeric vector that has year of birth of members of your family
   - you decide who to include
   - e.g., for my mom, dad, wife, son: `birth_years <- c(1944,1950,1981,2016)`
2. Write a loop that calculates current year minus birth year and prints this number for each member of your family
   - Within this loop, you will create a new variable that calculates current year minus birth year

Note: multiple correct ways to complete this task

Student exercise [SOLUTION]

1. Create a numeric vector that has year of birth of members of your family (you decide who to include)
2. Write a loop that calculates current year minus birth year and prints this number for each member of your family

```
birth_years <- c(1944,1950,1981,2016)
birth_years
#> [1] 1944 1950 1981 2016

for(y in birth_years) { # Loop sequence
  cat("object y=",y, fill=TRUE) # Loop body
  z <- 2018-y
  cat("value of",y,"minus",2018,"is",z, fill=TRUE)
}
#> object y= 1944
#> value of 1944 minus 2018 is 74
#> object y= 1950
#> value of 1950 minus 2018 is 68
#> object y= 1981
#> value of 1981 minus 2018 is 37
#> object y= 2016
#> value of 2016 minus 2018 is 2
```

4 Three ways to loop over a vector (atomic vector or a list)

Plan for learning more about loops

Rest of lecture on loops will proceed as follows:

1. Describe the three different ways to “loop over” a vector
2. Describe the two broad sorts of tasks to accomplish within body of a loop
   2.1 Modify an existing object (e.g., vector or list/data frame)
   2.2 Create a new object

Throughout, I’ll try to give you lots of examples and practice

Three ways to loop over an object

There are 3 ways to loop over elements of an object

1. **Loop over the elements** [approach we have used so far]
2. **Loop over names of the elements**
3. **Loop over numeric indices associated with element position** [approach recommended by Grolemnund and Wickham]

Will demonstrate 3 approaches on a named atomic vector and list/data frame

- Create named vector

```
vec=c("a"=5,"b"=-10,"c"=30)
vec
#>   a   b   c
#>   5 -10  30
```

- Create data frame with fictitious data, 3 columns (vars) and 4 rows (obs)

```
set.seed(12345) # so we all get the same variable values
df <- tibble(a = rnorm(4),b = rnorm(4),c = rnorm(4))
str(df)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    4 obs. of  3 variables:
#>  $ a: num  0.586 0.709 -0.109 -0.453
#>  $ b: num  0.606 -1.818 0.63 -0.276
#>  $ c: num  -0.284 -0.919 -0.116 1.817
```

4.1 Loop over elements

Approach 1: loop over elements of object [object=atomic vector]

- **sequence** syntax: `for (i in object_name)`
  - Sequence iterates through each element of the object
  - That is, **sequence iterates through *value* of each element, rather than *name* or *position* of element**
- in **body**.
  - value of `i` is equal to the contents of the `ith` element of the object

```
vec # print atomic vector object
#>   a   b   c
#>   5 -10  30

for (i in vec) {
  cat("value of object i=",i, fill=TRUE)
  cat("object i has: type=",typeof(i),"; length=",length(i),"; class=",class(i),
      "; attributes=",attributes(i),"\n",sep="",fill=TRUE) # "\n" adds line brea
}
#> value of object i= 5
#> object i has: type=double; length=1; class=numeric; attributes=
#>
#> value of object i= -10
#> object i has: type=double; length=1; class=numeric; attributes=
#>
#> value of object i= 30
#> object i has: type=double; length=1; class=numeric; attributes=
```

Approach 1: loop over elements of object [object=list]

- **sequence** syntax: `for (i in object_name)`
  - Sequence iterates through each element of the object
  - That is, **sequence iterates through *value* of each element**
- in **body**: value of `i` is equal to **contents** of `ith` element of object

```
df # print list/data frame object
#> # A tibble: 4 x 3
#>        a      b      c
#>    <dbl>  <dbl>  <dbl>
#> 1  0.586  0.606 -0.284
#> 2  0.709 -1.82  -0.919
#> 3 -0.109  0.630 -0.116
#> 4 -0.453 -0.276  1.82
#class(df)
#attributes(df)
for (i in df) {
  cat("value of object i=",i, fill=TRUE)
  cat("object type=",typeof(i),"; length=",length(i),"; class=",class(i),
      "; attributes=",attributes(i),"\n",sep="",fill=TRUE)
}
#> value of object i= 0.5855288 0.709466 -0.1093033 -0.4534972
#> object type=double; length=4; class=numeric; attributes=
#>
#> value of object i= 0.6058875 -1.817956 0.6300986 -0.2761841
#> object type=double; length=4; class=numeric; attributes=
#>
```

Approach 1: loop over elements of object

Example task:

- calculate mean value of each element of list object `df`

```
df # print list/data frame object
#> # A tibble: 4 x 3
#>        a      b      c
#>    <dbl>  <dbl>  <dbl>
#> 1  0.586  0.606 -0.284
#> 2  0.709 -1.82  -0.919
#> 3 -0.109  0.630 -0.116
#> 4 -0.453 -0.276  1.82

for (i in df) {
  # sequence
  cat("value of object i=",i, fill=TRUE)
  cat("mean value of object i=",mean(i, na.rm = TRUE), "\n", fill=TRUE)

}
#> value of object i= 0.5855288 0.709466 -0.1093033 -0.4534972
#> mean value of object i= 0.1830486
#>
#> value of object i= 0.6058875 -1.817956 0.6300986 -0.2761841
#> mean value of object i= -0.2145385
#>
#> value of object i= -0.2841597 -0.919322 -0.1162478 1.817312
#> mean value of object i= 0.1243956
```

4.2 Loop over element names

Approach 2: loop over names of object elements

To use this approach, elements in object must have name attributes

sequence syntax: `for (i in names(object_name))`

- Sequence iterates through the *name* of each element in object

in **body**, value of `i` is equal to *name* of `ith` element in object

- Access element contents using `object_name[i]`
  - same object type as `object_name`; retains attributes (e.g., *name*)
- Access element contents using `object_name[[i]]`
  - removes level of hierarchy, thereby removing attributes
  - Approach recommended by Wickham because isolates value of element

Example: Object= atomic vector

```
vec  # print atomic vector object
#>   a   b   c
#>   5 -10  30
names(vec)
#> [1] "a" "b" "c"
```

```
for (i in names(vec)) {
  cat("\n","value of object i=",i,"; type=",typeof(i),sep="",fill=TRUE)
  print(str(vec[i])) # "Access element contents using []"
  print(str(vec[[i]])) # "Access element contents using [[]]"
}
```

Approach 2: loop over names of object elements [object = list]

sequence syntax: `for (i in names(object_name))`

- Sequence iterates through the *name* of each element in object

in **body**, value of `i` is equal to *name* of `ith` element in object

- Access element contents using `object_name[i]`
  - Same object type as `object_name`; retains attributes (e.g., *name*)
- Access element contents using `object_name[[i]]`
  - Removes level of hierarchy, thereby removing attributes
  - Approach recommended by Wickham because isolates value of element

Example, object is a list

```
names(df)
#> [1] "a" "b" "c"
```

```
for (i in names(df)) {
  cat("\n","value of object i=",i,"; type=",typeof(i),sep="",fill=TRUE)
  print(str(df[i])) # "Access element contents using []"
  print(str(df[[i]])) # "Access element contents using [[]]"
}
```

Approach 2: loop over names of elements in object

Example task: calculate mean value of each element of list object `df`, using `[[]]` to access element contents

```
str(df)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    4 obs. of  3 variables:
#>  $ a: num  0.586 0.709 -0.109 -0.453
#>  $ b: num  0.606 -1.818 0.63 -0.276
#>  $ c: num  -0.284 -0.919 -0.116 1.817

for (i in names(df)) {
  cat("mean of element named",i,"is",mean(df[[i]], na.rm = TRUE), fill=TRUE)
}
#> mean of element named a is 0.1830486
#> mean of element named b is -0.2145385
#> mean of element named c is 0.1243956
```

What if we try to complete task using , `[]` to access element contents?

```
for (i in names(df)) {
  cat("mean of element named",i,"is",mean(df[i],na.rm = TRUE), fill=TRUE)
  #print(typeof(df[i]))
  #print(class(df[i]))
}
#?mean # mean function only works for particular *classes* of objects
```

4.3 Loop over element position number

Approach 3: Loop over numeric indices of element position

First explain sequence syntax, using atomic vector `vec` as object

- **sequence** syntax: `for (i in 1:length(object_name))`

```
vec # print named atomic vector vec
#>   a   b   c
#>   5 -10  30
length(vec)
#> [1] 3
1:length(vec)
#> [1] 1 2 3

for (i in 1:length(vec)) { # loop sequence
  cat("value of object i=",i,fill=TRUE) # loop body
}
#> value of object i= 1
#> value of object i= 2
#> value of object i= 3
```

Note: These two approaches yield same result as above

```
for (i in c(1,2,3)) {
  cat("value of object i=",i,fill=TRUE)
}
for (i in 1:3) {
  cat("value of object i=",i,fill=TRUE)
}
```

Approach 3: Loop over numeric indices of element position

Loop over element position number: Simple sequence syntax

```
for (i in 1:length(vec)) {
  cat("value of object i=",i,fill=TRUE)
}
#> value of object i= 1
#> value of object i= 2
#> value of object i= 3
```

Wickham's preferred sequence syntax: `for (i in seq_along(object_name))`

- `seq_along(x)` function returns a sequence from 1 value of `length(x)`

```
length(vec)
#> [1] 3
seq_along(vec)
#> [1] 1 2 3

for (i in seq_along(vec)) {
  cat("value of object i=",i,fill=TRUE)
}
#> value of object i= 1
#> value of object i= 2
#> value of object i= 3
```

Approach 3: Loop over numeric indices [SKIP]

Why Wickham prefers `seq_along(object_name)` over `1:length(object_name)`

- `seq_along` handles zero-length vectors correctly, and is therefore the "safe" version of `1:length(object_name)`

```
# create vector of length=0
y <- vector("double", 0)
length(y)
#> [1] 0

1:length(y)
#> [1] 1 0
for (i in 1:length(y)) {
  cat("value of object i=",i,fill=TRUE)
}
#> value of object i= 1
#> value of object i= 0

seq_along(y)
#> integer(0)
for (i in seq_along(y)) {
  cat("value of object i=",i,fill=TRUE)
}
```

Personally, I find `1:length(object_name)` much more intuitive

Approach 3: Loop over numeric indices of element position

sequence syntax: `for (i in 1:length(object_name))` **OR** `for (i in seq_along(object_name))`

- Sequence iterates through *position number* of each element in the object

In **body**, value of `i` equals the *position number* of `ith` element in object

- Access element contents using `object_name[i]`
  - Same object type as `object_name`; retains attributes (e.g., *name*)
- Access element contents using `object_name[[i]]` [RECOMMENDED]
  - Removes level of hierarchy, thereby removing attributes

Example, object is atomic vector

```
vec
#>   a   b   c
#>   5 -10  30
```

```
for (i in 1:length(vec)) {
  cat("\n","value of object i=",i,"; type=",typeof(i),sep="",fill=TRUE)
  print(str(vec[i])) # "Access element contents using []"
  print(str(vec[[i]])) # "Access element contents using [[]]"
}
```

Approach 3: Loop over numeric indices of element position

sequence: `for (i in 1:length(object_name))` **OR** `for (i in seq_along(object_name))`

- Sequence iterates through *position number* of each element in the object

In **body**, value of `i` equals the *position number* of `ith` element in object

- Access element contents using `object_name[i]`
  - Same object type as `object_name`; retains attributes (e.g., *name*)
- Access element contents using `object_name[[i]]` [RECOMMENDED]
  - Removes level of hierarchy, thereby removing attributes

Example, object is a list

```
df %>% head(n=3)
#> # A tibble: 3 x 3
#>        a      b      c
#>    <dbl>  <dbl>  <dbl>
#> 1  0.586  0.606 -0.284
#> 2  0.709 -1.82  -0.919
#> 3 -0.109  0.630 -0.116
```

```
for (i in 1:length(df)) {
  cat("\n","value of object i=",i,"; type=",typeof(i),sep="",fill=TRUE)
  print(str(df[i])) # "Access element contents using []"
  print(str(df[[i]])) # "Access element contents using [[]]"
}
```

Approach 3: Loop over numeric indices of element position

Example task:

- Calculate mean value of each element of list object `df`, using `for (i in seq_along(df))` to create sequence and using `[[]]` to access element contents

```
for (i in seq_along(df)) {
  cat("mean of element named",i,"is",mean(df[[i]], na.rm = TRUE), fill=TRUE)
}
#> mean of element named 1 is 0.1830486
#> mean of element named 2 is -0.2145385
#> mean of element named 3 is 0.1243956
```

What happens if we try to complete task using , using `[]` to access element contents?

```
for (i in seq_along(df)) {
  cat("mean of element named",i,"is",mean(df[i],na.rm = TRUE), fill=TRUE)
  #print(typeof(df[i]))
  #print(class(df[i]))
}
#?mean # mean(object) requires object to be numeric or logical
```

Approach 3: Loop over numeric indices of element position

When looping over numeric indices, you can extract element names based on element position

- First, let's experiment w/ `attributes()` and `names()` functions

`attributes()` function [output omitted]

```
attributes(df)
attributes(df[1]) # not null
attributes(df[[1]]) # null: removing level of hierarchy removes attributes
```

`names()` functions

```
names(df)
#> [1] "a" "b" "c"
names(df[1]) # not null
#> [1] "a"
names(df[[1]]) # null: object df[[1]] has no attributes; just values
#> NULL

names(df)[[1]] # not null: we extract names of df, then select first element
#> [1] "a"
```

Approach 3: Loop over numeric indices of element position

When looping over numeric indices, you can extract element names based on element position

- First, experiment w/ `names()` function

```
names(df)
#> [1] "a" "b" "c"
names(df)[[1]] # not null: we extract names of df, then select first element
#> [1] "a"
```

- Second, apply what we learned to loop

```
for (i in seq_along(df)) {
  #print(names(df)[[i]])
  cat("i=",i,"; names=",names(df)[[i]],sep="",fill=TRUE)
}
#> i=1; names=a
#> i=2; names=b
#> i=3; names=c
```

Summary: Three ways to loop over object

1. Loop over elements
2. Loop over element names
3. Loop over numeric indices of element position

Why Wickham prefers "loop over numeric indices of element" approach [3]:

- given element position number, can extract element name[2] and value[1]

```
for (i in seq_along(df)) {
  cat("i=",i,sep="",fill=TRUE)

  name <- names(df)[[i]] # value of object "name" is what we loop over in approa
  cat("name=",name,sep="",fill=TRUE)

  value <- df[[i]] # value of object "value" is what we loop over in approach 1
  cat("value=",value,"\n",sep="",fill=TRUE)
}
#> i=1
#> name=a
#> value=0.58552880.709466-0.1093033-0.4534972
#>
#> i=2
#> name=b
#> value=0.6058875-1.8179560.6300986-0.2761841
#>
#> i=3
#> name=c
#> value=-0.2841597-0.919322-0.11624781.817312
```

5 Modifying vs. Creating new object

Modify object or create new object

Grolemund and Wickham differentiate between two types of tasks loops accomplish: (1) modify existing object; and (2) create new object

1. **Modify an existing object**
   - example: looping through a set of variables in a data frame to:
     - Modifying these variables OR
     - Creating new variables (within the existing data frame object)
   - When writing loops in Stata/SAS/SPSS, we are usually modifying an existing object because these programs typically only have one object - a dataset - open at a time)
2. **Create a new object**
   - Example: Create an object that has summary statistics for each variable; this object will be the basis for a table or graph
   - Often the new object will be a vector of results based on looping through elements of a data frame
   - In R (as opposed to Stata/SAS/SPSS) creating a new object is very common because R can hold many objects at the same time

5.1 Loops that create new object

Creating a new object

So far our loops have two components:

1. sequence
2. body

When we create a new object to store the results of a loop, our loops have three components

1. sequence
2. body
3. output
   - this is the new object that will store results created from your loop

Grolemund and Wickham recommend creating this new object **prior** to writing the loop (rather than creating the new object within the loop)

> *"Before you start loop...allocate sufficient space for the output. This is very important for efficiency: if you grow the for loop at each iteration using c() (for example), your for loop will be very slow."*

Creating a new object

Create sample data frame named `df`

```
set.seed(54321)
df <- tibble(a = rnorm(10),b = rnorm(10),c = rnorm(10),d = rnorm(10))
```

Task:

- Using the data frame `df`, which contains data on four numeric variables, create a new object that contains the mean value of each variable

In a previous example, we calculated mean for each variable

```
for (i in seq_along(df)) {
  cat("mean of element named",i,"is",mean(df[[i]], na.rm = TRUE),fill=TRUE)
}
#> mean of element named 1 is -0.2646042
#> mean of element named 2 is 0.6025297
#> mean of element named 3 is 0.0349128
#> mean of element named 4 is -0.4557522
```

Now we just have to create an object to store these results

Creating a new object

Task: Create a new object that contains mean value of each variable in `df`

Wickham recommends creating new object **prior** to creating loop

- You must specify type and length of new object
- New object will contain mean for each variable; should be numeric vector with number of elements (length) equal to number of variables in `df`

Create object to hold output; we'll name this object `output`

```
output <- vector("double", ncol(df)) # create object
typeof(output)
#> [1] "double"
length(output)
#> [1] 4
length(df)
#> [1] 4
```

Create loop; use position number to assign variable means to elements of vector `output`

```
for (i in seq_along(df)) {
  #cat("i=",i,fill=TRUE)
  output[[i]] <- mean(df[[i]], na.rm = TRUE) # mean of df[[1]] assigned to outpu
}
output
#> [1] -0.2646042  0.6025297  0.0349128 -0.4557522
```

5.2 Loops that modify existing object

Example of modifying an object: z-score loop

Task (from Christenson lecture):

- Write a loop that calculates z-score for a set of variables in a data frame and then replaces the original variables with the z-score variables

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

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

We wrote a z-score function in the functions lecture; this can be basis of our z-score loop

```
z_score <- function(x) {
  (x - mean(x, na.rm=TRUE))/sd(x, na.rm=TRUE)
}
z_score(df$a)
#>  [1]  0.06413227 -0.49645520 -0.38869153 -1.03714747 -0.10735356
#>  [6] -0.62178518 -1.06830812  2.08077332  1.24215309  0.33268238
z_score(df[["a"]]) # same
#>  [1]  0.06413227 -0.49645520 -0.38869153 -1.03714747 -0.10735356
#>  [6] -0.62178518 -1.06830812  2.08077332  1.24215309  0.33268238
z_score(df[[1]]) # same
#>  [1]  0.06413227 -0.49645520 -0.38869153 -1.03714747 -0.10735356
#>  [6] -0.62178518 -1.06830812  2.08077332  1.24215309  0.33268238
```

Example of modifying an object: z-score loop

Task: write loop that replaces variables with z-scores of those variables

When modifying existing object, we only need to write **sequence** and **body**

- **sequence**.
  - data frame `df` has 4 variables and all are quantitative
  - so write a sequence that loops across each element of `df`
    - `for (i in seq_along(df))`
- **body**.
  - body of z-score function:
    - `(x - mean(x, na.rm=TRUE))/sd(x, na.rm=TRUE)`
  - Substitute `df[[i]]` for `x`:
    - `(df[[i]] - mean(df[[i]], na.rm=TRUE))/sd(df[[i]], na.rm=TRUE)`
  - Assign (replace) each observation the value of its z-score:
    - `df[[i]] <- (df[[i]] - mean(df[[i]], na.rm=TRUE))/sd(df[[i]], na.rm=TRUE)`

```
set.seed(54321)
(df <- tibble(a = rnorm(10),b = rnorm(10),c = rnorm(10),d = rnorm(10)))

for (i in seq_along(df)) {
  cat("i=",i,"; mean=",mean(df[[i]], na.rm=TRUE),"; sd=",sd(df[[i]], na.rm=TRUE)
  #print((df[[i]] - mean(df[[i]], na.rm=TRUE))/sd(df[[i]], na.rm=TRUE)) # show z
  df[[i]] <- (df[[i]] - mean(df[[i]], na.rm=TRUE))/sd(df[[i]], na.rm=TRUE) # mod
}
str(df)
```

Modify z-score loop to work with non-numeric variables

What happens if we apply our loop to the data frame `df_bama`, which has both string and numeric variables?

Create data frame `df_bama`

```
load(url("https://github.com/ozanj/rclass/raw/master/data/recruiting/recruit_eve
df_bama <- df_event %>% arrange(univ_id,event_date) %>%
  select(instnm,univ_id,event_date,event_type,event_state,zip,med_inc) %>%
  filter(row_number()<6)
str(df_bama)
```

Attempt to run loop; what went wrong?

```
for (i in seq_along(df_bama)) {
  cat("i=",i,"; mean=",mean(df_bama[[i]], na.rm=TRUE),"; sd=",sd(df_bama[[i]], n
  #print((df_bama[[i]] - mean(df_bama[[i]], na.rm=TRUE))/sd(df_bama[[i]], na.rm=
  df_bama[[i]] <- (df_bama[[i]] - mean(df_bama[[i]], na.rm=TRUE))/sd(df_bama[[i]
}
df_bama
```

Modify z-score loop to work with non-numeric variables

What happens if we apply our loop to the data frame `df_bama`, which has both string and numeric variables?

Let's modify our loop so that it only calculates z-score if for non-integer, numeric variables

```
str(df_bama)
for (i in seq_along(df_bama)) {
  cat("i=",i,"; var name=",names(df_bama)[[i]],"; type=",typeof(df_bama[[i]]),
      "; class=",class(df_bama[[i]]),sep="",fill=TRUE)

  if(is.numeric(df_bama[[i]]) & (!is_integer(df_bama[[i]]))) {
    df_bama[[i]] <- (df_bama[[i]] - mean(df_bama[[i]], na.rm=TRUE))/sd(df_bama[[
  } else {
    # do nothing
  }
}
str(df_bama)
```

Modify object: embed z-score loop in function [SKIP]

Recreate `df` and `df_bama` [ouput and code omitted]

Can we embed this loop in a function that takes the data frame as an argument so we don't have to modify loop for each data frame?

```
z_score <- function(x) {

  for (i in seq_along(x)) {
    cat("i=",i,"; var name=",names(x)[[i]],"; type=",typeof(x[[i]]),
        "; class=",class(x[[i]]),sep="",fill=TRUE)

    if(is.numeric(x[[i]]) & (!is_integer(x[[i]]))) {
      x[[i]] <- (x[[i]] - mean(x[[i]], na.rm=TRUE))/sd(x[[i]], na.rm=TRUE)
    } else {
       #do nothing
    }
  }
}
#apply to data frame df
df_z <- z_score(df)
df; df_z
#apply to data frame df_bama
df_bama_z <- z_score(df_bama)
df_bama; df_bama_z
```

6 When to write a loop (vs. a function); recipe for writing loops

When to write a loop

Broadly, rationale for writing loop same as rationale for writing function:

- Do not duplicate code
- Can make changes to code in one place rather than many

When to write a loop:

- Grolemund and Wickham say **don't copy and paste more than twice**
- If you find yourself doing this, consider writing a loop or function

Don't worry about knowing all the situations you should write a loop

- Rather, you'll be creating analysis dataset or analyzing data and you will notice there is some task that you are repeating over and over
- Then you'll think "oh, I should write a loop or function for this"

When to write a loop vs a functions

Usually obvious when you are duplicating code, but unclear whether you should write a loop or whether you should write a function.

- Often, a repeated task can be completed with a loop or a function

In my experience, loops are better for repeated tasks when the individual tasks are **very** similar to one another

- e.g., a loop that reads in data sets from individual years; each dataset you read in differs only by directory and name
- e.g., a loop that converts negative values to `NA` for a set of variables

Because functions can have many arguments, functions are better when the individual tasks differ substantially from one another

- Example: function that runs regression and creates formatted results table
  - function allows you to specify (as function arguments): dependent variable; independent variables; what model to run, etc.

Note

- Can embed loops within functions; can call functions within loops
- But for now, just try to understand basics of functions and loops

Recipe for how to write loop

The general recipe for how to write a loop is very similar to the recipe for writing a function:

1. Complete the task for one instance outside a loop (this is akin to writing the **body** of the loop)
2. Write the **sequence**
3. Which parts of the body need to change with each iteration
4. *if* you are creating a new object store output of the loop, create this outside of the loop
5. Construct the loop

7 Practice: How well do public universities cover in-state public high schools?

Load recruiting data

Load data frame with one observation per high school and variables for visits by each public research university in sample

- Note: this data frame has more vars than previous data frame we used

```
rm(list = ls()) # remove all objects
load(url("https://github.com/ozanj/rclass/raw/master/data/recruiting/recruit_sch
```

We are interested in creating measures of how good a job public universities are doing visiting in-state public high schools

- Create data frame with one observation for each public high school

```
#names(df_school_all)
df_school_all %>% str()
df_pubhs <- df_school_all %>% # Create data-frame that keeps public high schools
  filter(school_type=="public") %>% select(-school_type)
rm(df_school_all)
```

Create standalone objects (output and code omitted)

1. Character vector containing ID for each public university
2. A named list containing university name

How well do public universities cover in-state public high schools

Task: for each public research university, calculate the number and percent of public high schools in the university's home state that received a visit

First, let's accomplish task outside of a loop for one university [Tidyverse]

- let's choose `"U of South Carolina"`, `ID==218663`

```
#"state_code" is the 2-letter high school state code
df_pubhs %>% select(state_code) %>% str()

#variables starting with "inst_" identify state the university is located in
df_pubhs %>% select(inst_218663) %>% str()
df_pubhs %>% select(inst_218663) %>% count(inst_218663) # these vars don't vary

#variables starting with "visits_by_" indicate number of visits HS got in 2017
df_pubhs %>% select(visits_by_218663) %>% str()
df_pubhs %>% select(visits_by_218663) %>% count(visits_by_218663)

#filter only obs where HS state code equals home state of university
df_pubhs %>% filter(state_code==inst_218663) %>% count() # count pub HS in SC

#Create measures: number pub HS in SC; number w/ visit; pct w/ visit
df_pubhs %>% filter(state_code==inst_218663) %>% select(visits_by_218663) %>%
  mutate(got_visit=ifelse(visits_by_218663>0,1,0)) %>%
  summarise(n_hs=n(),n_visit=sum(got_visit),pct_visit=sum(got_visit)/n())
```

How well do public universities cover in-state public high schools

Task: for each public research university, calculate the number and percent of public high schools in the university's home state that received a visit

First, let's accomplish task outside of a loop for one university [Base R]

- let's choose `"U of South Carolina"`, `ID==218663`

```
#"state_code" is the 2-letter high school state code
str(df_pubhs$state_code)

#variables starting with "inst_" identify state the university is located in
str(df_pubhs$inst_218663)
table(df_pubhs$inst_218663, useNA='ifany') # these vars don't vary

#variables starting with "visits_by_" indicate number of visits HS got in 2017
str(df_pubhs$visits_by_218663)
table(df_pubhs$visits_by_218663, useNA='ifany')

#filter only obs where HS state code equals home state of university
tempdf <- subset(df_pubhs,df_pubhs[["state_code"]]==df_pubhs[["inst_218663"]])
  #tempdf <- subset(df_pubhs,df_pubhs$state_code==df_pubhs$inst_218663) # same a
  #tempdf <- subset(df_pubhs,state_code==inst_218663) # same as above

#Create 0/1 indicator of whether got visit
tempdf$got_visit <- ifelse(tempdf$visits_by_218663>0,1,0)

#frequency count of schools that got visits vs. not
table(tempdf$got_visit, useNA='ifany')
```

How well do public universities cover in-state public high schools

Task: for each public research university, calculate the number and percent of public high schools in the university's home state that received a visit

Build loop [Base R approach]

- first, loop through each value of list `instnm`

```
instnm
for (i in seq_along(instnm)) {
  cat("\n","i=",i,sep="",fill=TRUE)

  name <- names(instnm)[[i]] # name of element
  cat("name=",name,sep="",fill=TRUE)

  value <- instnm[[i]] # value of element
  cat("value=",value,sep="",fill=TRUE)
}
```

How well do public universities cover in-state public high schools

Task: for each public research university, calculate the number and percent of public high schools in the university's home state that received a visit

Build loop

- next, create "inst_..." and "visits_by_..." vars for each id
- keep obs in same state as university
- create 0/1 variable of whether high school got a visit

```
for (i in seq_along(instnm)) {
  cat("\n","i=",i,"; ",names(instnm)[[i]],sep="",fill=TRUE)

  #create object called inst_var; value is "inst_id" (e.g., "inst_166629")
  cat("inst_",instnm[[i]],sep="",fill=TRUE)
  inst_var <- paste("inst_",instnm[[i]],sep="")
  print(inst_var)

  #create object called visits_by_var; value is "visits_by_id" (e.g., "visits_by
  visits_by_var <- paste("visits_by_",instnm[[i]],sep="")
  print(visits_by_var)

  #create subset of data with high schools in same state as the university
  tempdf <- subset(df_pubhs,df_pubhs[["state_code"]]==df_pubhs[[inst_var]])
      # code df_pubhs[[inst_var]]) evaluates to df_pubhs[["inst_166629"]]) or wh
      # this is same as instnm[[i]] evaluating to instnm[[16]] or whatever curre
  #Create 0/1 indicator of whether got visit
  tempdf$got_visit <- ifelse(tempdf[[visits_by_var]]>0,1,0)
```

How well do public universities cover in-state public high schools

Task: for each public research university, calculate the number and percent of public high schools in the university's home state that received a visit

Build loop

- next, create count of number of visted and non-visited in-state schools

```
for (i in seq_along(instnm)) {
  cat("\n","i=",i,"; ",names(instnm)[[i]],sep="",fill=TRUE)

  inst_var <- paste("inst_",instnm[[i]],sep="")
  visits_by_var <- paste("visits_by_",instnm[[i]],sep="")

  tempdf <- subset(df_pubhs,df_pubhs[["state_code"]]==df_pubhs[[inst_var]]) # ke
  tempdf$got_visit <- ifelse(tempdf[[visits_by_var]]>0,1,0) # create 0/1 indicat

  #create frequency table of number of schools with and without visits
  print(table(tempdf$got_visit, useNA='ifany'))
  ct_table <- table(tempdf$got_visit, useNA='ifany') # named vector with 2 eleme

  #create proportion table
  print(prop.table(ct_table))
  pr_table <- prop.table(ct_table) # named vector with 2 elements str(pr_table)
}
```

How well do public universities cover in-state public high schools

Task: for each public research university, calculate the number and percent of public high schools in the university's home state that received a visit

Here is tidyverse approach to loop, which uses some programming concepts we haven't covered

```
for (i in seq_along(instnm)) {
  cat("\n","i=",i,"; ",names(instnm)[[i]],sep="",fill=TRUE)

  #create object called inst_var; value is "inst_id" (e.g., "inst_166629")
  inst_var <- paste("inst_",instnm[[i]],sep="")

  #create object called visits_by_var; value is "visits_by_id" (e.g., "visits_by
  visits_by_var <- paste("visits_by_",instnm[[i]],sep="")

  #Create measures: number pub HS in SC; number w/ visit; pct w/ visit
  df_pubhs %>% filter_(glue::glue("state_code=={inst_var}")) %>%
      select_(visits_by_var) %>%
      mutate_(got_visit=glue::glue("ifelse({visits_by_var}>0,1,0)")) %>%
      summarise(n_hs=n(),n_visit=sum(got_visit),pct_visit=sum(got_visit)/n())
}
```