Investigating objects and data patterns using base R

Managing and Manipulating Data Using R

Lecture outline

1. Investigate objects, base R
   - 1.1 Functions to describe objects
   - 1.2 Variables names
   - 1.3 View and print data
   - 1.4 Missing values
2. Subsetting using subset operators
   - 2.1 Subset atomic vectors using []
   - 2.2 Subsetting lists/data frames using []
   - 2.3 Subsetting lists/data frames using [[]] and $
   - 2.4 Subsetting data frames with [] combined with $
3. Subsetting using subset() function
4. Sorting data

Libraries we will use today

“Load” the package we will use today (output omitted)

```
library(tidyverse)
```

If package not yet installed, then must install before you load. Install in “console” rather than .Rmd file

- Generic syntax: `install.packages("package_name")`
- Install “tidyverse”: `install.packages("tidyverse")`

Note: when we load package, name of package is not in quotes; but when we install package, name of package is in quotes:

- `install.packages("tidyverse")`
- `library(tidyverse)`

Investigate objects, base R

Load .Rdata data frames we will use today

Data on off-campus recruiting events by public universities

- Data frame object `df_event`
  - One observation per university, recruiting event
- Data frame object `df_school`
  - One observation per high school (visited and non-visited)

```
rm(list = ls()) # remove all objects in current environment

getwd()
#> [1] "C:/Users/ozanj/Documents/rclass1/lectures/patterns_base_r"
#load dataset with one obs per recruiting event
load(url("https://github.com/ozanj/rclass/raw/master/data/recruiting/recruit_ev
#load("../../data/recruiting/recruit_event_somevars.Rdata")

#load dataset with one obs per high school
load(url("https://github.com/ozanj/rclass/raw/master/data/recruiting/recruit_sc
#load("../../data/recruiting/recruit_school_somevars.Rdata")
```

Functions to describe objects

Simple base R functions to describe objects

This section introduces some base R functions to describe objects (some of these you have seen before)

- list objects, `list.files()` and `ls()`
- remove objects, `rm()`
- object type, `typeof()`
- object length (number of elements), `length()`
- object structure, `str()`
- number of rows and columns, `ncol()` and `nrow`

I use the functions `typeof()`, `length()`, `str()` anytime I encounter a new object

- Helps me understand the object before I start working with it

Listing objects

Files in your working directory

`list.files()` function lists files in your current working directory

- if you run this code from .Rmd file, working directory is location .Rmd file is stored

```
getwd() # what is your current working directory
#> [1] "C:/Users/ozanj/Documents/rclass1/lectures/patterns_base_r"
list.files()
#> [1] "fp1.JPG"               "fp2.JPG"               "patterns_base_r.log"
#> [4] "patterns_base_r.pdf"   "patterns_base_r.Rmd"   "patterns_base_r.tex"
#> [7] "test.txt"              "transform-logical.png"
```

Objects currently open in your R session

Listing objects currently open in your R session

`ls()` function lists objects currently open in R

```
x <- "hello!"
ls() # Objects open in R
#> [1] "df_event"  "df_school" "x"
```

Removing objects currently open in your R session

`rm()` function removes specified objects open in R

```
rm(x)
ls()
#> [1] "df_event"  "df_school"
```

Command to remove all objects open in R (I don't run it)

```
#rm(list = ls())
```

Base R functions to describe objects, `typeof()`

`typeof()` function determines the the internal storage type of an object (e.g., logical vector, integer vector, list)

- syntax
  - `tyepof(x)`
- arguments
  - `x` : any R object
- help:

```
?typeof
```

Examples

- Recall that a data frame is an object where **type** is a list

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

Base R functions to describe objects, `length()`

`length()` function determines the length of an R object

- for atomic vectors and lists, `length()` is the number of elements in the object
- syntax
  - `length(x)`
- arguments
  - `x` : any R object
- help:

```
?length
```

Example, length of an atomic vector is

```
length(c(TRUE,TRUE,FALSE,NA))
#> [1] 4
```

Example, length of a list or data frame

- length of a list is the number of elements
- data frame is a list
- length of a data frame = number of elements = number of variables

```
length(df_event) # = num elements = num columns
#> [1] 33
```

Base R functions to describe objects, `str()`

`str()` function compactly displays the structure of an R object

- "structure" includes type, length, and attribute of object and also nested objects
- syntax: `str(object)`
- arguments (partial)
  - `object` : any R object
  - `max.level` : max level of nesting to display nested structures; default `NA` = all levels
- help: `?str`

Example, atomic vectors

```
str(c(TRUE,TRUE,FALSE,NA))
#>  logi [1:4] TRUE TRUE FALSE NA
str(object = c(TRUE,TRUE,FALSE,NA))
#>  logi [1:4] TRUE TRUE FALSE NA
```

Example, lists/data frames (output omitted)

```
x <- list(c(1,2), list("apple", "orange"), list(2, 3)) # list
str(x)

str(df_event) # data frame
```

Base R functions to describe objects, `ncol()` and `nrow()`

`ncol()` `nrow()` and `dim()` functions

- Description
  - `ncol()` = number of columns; `nrow()` = number of rows
- syntax: `ncol(x)` `nrow(x)` `dim(x)`
- arguments
  - `x` : a vector, array, data frame, or NULL
- value/return:
  - if object `x` is an atomic vector: `ncol()` and `nrow()` returns `NULL`
  - if object `x` is a list but not a data frame: `ncol()` and `nrow()` returns `NULL`
  - if object `x` is a data frame: `ncol()` and `nrow()` returns integer of length 1

Example, object is a data frame

```
ncol(df_event) # num columns = num elements = num variables
#> [1] 33
nrow(df_event) # num rows = num observations
#> [1] 18680
# can wrap ncol() or nrow() within str() to see what functions return
#str(ncol(df_event))
```

Example, object is atomic vector or list that is not a data frame (output omitted)

```
ncol(c(TRUE,TRUE,FALSE,NA)) # atomic vector
x <- list(c(1,2), list("apple", "orange"), list(2, 3)) # list
nrow(x)
```

Base R functions to describe objects, `dim()`

`dim()` function returns the dimensions of an object (e.g., number of rows and columns)

- syntax: `dim(x)`
- arguments
  - `x`: a vector, array, data frame, or NULL
- value/return:
  - if object `x` is a data frame: `dim()` returns integer of length 2
    - first element = number of rows; second element = number of columns
  - if object `x` is an atomic vector: `dim()` returns `NULL`
  - if object `x` is a list but not a data frame: `dim()` returns `NULL`

Example, object is a data frame

```
dim(df_event) # shows number rows by columns
#> [1] 18680    33

str(dim(df_event)) # can wrap dim() within str() to see what functions return
#>  int [1:2] 18680 33
```

Example, object is atomic vector or list that is not a data frame (output omitted)

```
dim(c(TRUE,TRUE,FALSE,NA)) # atomic vector
x <- list(c(1,2), list("apple", "orange"), list(2, 3)) # list
dim(x)
```

Variables names

`names()` function

`names()` function gets or sets the names of elements of an object

- syntax:
  - get the names of an object: `names(x)`
  - set the names of an object: `names(x) <- value`
- arguments (partial)
  - `x` : an R object
  - `value` : a character vector with same length as object `x` or `NULL`
- value/return
  - `names(x)` returns a character vector of legnth = `length(x)` in which each element is the name of the element of `x`

Example, get names (of atomic vector)

```
a <- c(v1=1,v2=2,3,v4="hi!") # named atomic vector
a
#>    v1    v2          v4
#>   "1"   "2"   "3" "hi!"
length(a)
#> [1] 4
names(a)
#> [1] "v1" "v2" ""   "v4"
length(names(a)) # investigate length of object names(a)
#> [1] 4
str(names(a)) # investigate structure of object names(a)
#>  chr [1:4] "v1" "v2" "" "v4"
```

`names()` function

`names()` function gets or sets the names of elements of an object

- syntax:
  - get the names of an object: `names(x)`
  - set the names of an object: `names(x) <- value`
- arguments (partial)
  - `x` : an R object
  - `value` : a character vector with same length as object `x` or `NULL`
- value/return
  - `names(x)` returns a character vector of legnth = `length(x)` in which each element is the name of the element of `x`

Example, set names (of atomic vector)

```
names(a) <- NULL # set names of vector a to NULL
a
#> [1] "1"   "2"   "3"   "hi!"
names(a)
#> NULL

names(a) <- c("var1","var2","var3","var4") # set names of vector a
a
#>  var1  var2  var3  var4
#>   "1"   "2"   "3" "hi!"
names(a)
#> [1] "var1" "var2" "var3" "var4"
```

Applying `names()` function to a data frame

Recall that a data frame is an object where **type** is a **list** and each **element** is **named**

- each element is a variable
- each element name is a variable name

Example (output omitted)

```
names(df_event)
```

Investigate the object `names(df_event)`

```
typeof(names(df_event)) # type = character vector
#> [1] "character"
length(names(df_event)) # length = number of variables in data frame
#> [1] 33
str(names(df_event)) # structure of names(df_event)
#>  chr [1:33] "instnm" "univ_id" "instst" "pid" "event_date" "event_type" ...
```

We can even assign a new object based on `names(df_event)`

```
names_event <- names(df_event)
typeof(names_event) # type = character vector
#> [1] "character"
length(names_event) # length = number of variables in data frame
#> [1] 33
str(names_event) # structure of names(df_event)
#>  chr [1:33] "instnm" "univ_id" "instst" "pid" "event_date" "event_type" ...
```

Variable names

Refer to specific named elements of an object using this syntax:

- `object_name$element_name`

When object is data frame, refer to specific variables using this syntax:

- `data_frame_name$varname`
- **This approach to isolating variables very useful for investigating data**

```r
#df_event$instnm
typeof(df_event$instnm)
#> [1] "character"
typeof(df_event$med_inc)
#> [1] "double"
```

Variable names

Data frames are lists with following criteria:

- each element of list is (usually) a vector; each element of list is a variable
- length of data frame = number of variables

```
length(df_event)
#> [1] 33
nrow(df_event)
#> [1] 18680
#str(df_event)
```

- each element of the list (i.e., variable) has the same length
  - Length of each variable is equal to number of observations in data frame

```
typeof(df_event$event_state)
#> [1] "character"
length(df_event$event_state)
#> [1] 18680
str(df_event$event_state)
#>  chr [1:18680] "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" ..

typeof(df_event$med_inc)
#> [1] "double"
length(df_event$med_inc)
#> [1] 18680
str(df_event$med_inc)
#>  num [1:18680] 71714 89122 70137 70137 71024 ...
```

Variable names

The object `df_school` has one obs per high school

- variable `visits_by_100751` shows number of visits by University of Alabama to each high school
- like all variables in a data frame, the var `visits_by_100751` is just a vector

```
typeof(df_school$visits_by_100751)
#> [1] "integer"
length(df_school$visits_by_100751) # num elements in vector = num obs
#> [1] 21301
str(df_school$visits_by_100751)
#>  int [1:21301] 0 0 0 0 0 0 0 0 0 0 ...
sum(df_school$visits_by_100751) # sum of values of var across all obs
#> [1] 3338
```

We perform calculations on a variable like we would on any vector of same type

```
v <- c(2,4,6)
typeof(v)
#> [1] "double"
length(v)
#> [1] 3
sum(v)
#> [1] 12
```

View and print data

Viewing and printing, data frames

Many ways to view/print a data frame object. Here are three ways:

1. Simply type the object name (output omitted)
    - number of observations and rows printed depend on YAML header settings and on object "attributes" (attributes discussed in future week)

```
df_event
```

2. Use the `View()` function to view data in a browser

```
View(df_event)
```

3. `head()` to show the first *n* rows

```
#?head
head(df_event, n=5)
```

Viewing and printing, data frames

`obj_name[<rows>,<cols>]` to print specific rows and columns of data frame

- particularly powerful when combined with sequences (e.g., `1:10`)

Examples (output omitted):

- Print first five rows, all vars

```
df_event[1:5, ]
```

- Print first five rows and first three columns

```
df_event[1:5, 1:3]
```

- Print first three columns of the 100th observation

```
df_event[100, 1:3]
```

- Print the 50th observation, all variables

```
df_event[50,]
```

Viewing and printing, variables within data frames

Recall that:

- `obj_name$var_name` print specifics elements (i.e., variables) of a data frame

```
df_event$zip
```

- each element (i.e., variable) of data frame is an **atomic vector** with **length** = number of observations

```
typeof(df_event$zip)
#> [1] "character"
length(df_event$zip)
#> [1] 18680
```

- each element of a variable is the value of the variable for one observation

Print specific elements (i.e., observations) of variable based on element position

- syntax: `obj_name$var_name[<element position>]`
- vectors don't have "rows" or "columns"; they just have elements
- syntax be combined with sequences (e.g., print first 10 observations)

```
df_event$event_state[1:10] # print obs 1-10 of variable "event_state"
#>  [1] "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA" "MA"
df_event$event_type[6:10] # print obs 6-10 of variable "event_type"
#> [1] "private hs" "private hs" "public hs"  "private hs" "public hs"
```

Viewing and printing, variables within data frames

Print specific elements (i.e., observations) of variable based on element position

- syntax: `obj_name$var_name[<element position>]`

Example, print individual elements

```
df_event$zip[1:5] # print obs 1-5 of variable for event zip code
#> [1] "01002" "01007" "01020" "01020" "01027"
df_event$zip[1] # print obs 1 of variable for event zip code
#> [1] "01002"
df_event$zip[5] # print obs 5 of variable for event zip code
#> [1] "01027"
df_event$zip[c(1,3,5)] # print obs 5 of variable for event zip code
#> [1] "01002" "01020" "01027"
```

Print specific elements of multiple variables using combine function `c()`

- syntax: `c(obj_name$var1_name[<element position>], obj_name$var2_name[<element posit`
- Example: print first five observations of variables `"event_state"` and `"event_type"`

```
c(df_event$event_state[1:5],df_event$event_type[1:5])
#>  [1] "MA"        "MA"        "MA"        "MA"        "MA"        "public hs"
#>  [7] "public hs" "public hs" "public hs" "public hs"
```

Exercise

Create a printing exercise using the df_school data frame

1. Use `obj_name[<rows>,<cols>]` to print the first 5 rows and 3 columns of data frame
2. Use head() to print first 4 observations
3. Use `obj_name$var_name[1:10]` to print the first 10 observations of a variable
4. Use combine() to print the first 3 observations of variables “school_type” & “name”

Solution

1. Use `obj_name[<rows>,<cols>]` to print the first 5 rows and 3 columns of data frame

```
df_school[1:5,1:3]
#> # A tibble: 5 x 3
#>   state_code school_type ncessch
#>   <chr>      <chr>       <chr>
#> 1 AK         public      020000100208
#> 2 AK         public      020000100211
#> 3 AK         public      020000100212
#> 4 AK         public      020000100213
#> 5 AK         public      020000300216
```

Solution

2. Use head() to print first 4 observations

```
head(df_school, n=4)
#> # A tibble: 4 x 26
#>   state_code school_type ncessch name  address city  zip_code pct_white
#>   <chr>      <chr>       <chr>   <chr> <chr>   <chr> <chr>        <dbl>
#> 1 AK         public      020000~ Beth~ 1006 R~ Beth~ 99559         11.8
#> 2 AK         public      020000~ Ayag~ 106 Vi~ Kong~ 99559          0  
#> 3 AK         public      020000~ Kwig~ 108 Vi~ Kwig~ 99622          0  
#> 4 AK         public      020000~ Nels~ 118 Vi~ Toks~ 99637          0  
#> # ... with 18 more variables: pct_black <dbl>, pct_hispanic <dbl>,
#> #   pct_asian <dbl>, pct_amerindian <dbl>, pct_other <dbl>, num_fr_lunch <dbl
#> #   total_students <dbl>, num_took_math <dbl>, num_prof_math <dbl>,
#> #   num_took_rla <dbl>, num_prof_rla <dbl>, avgmedian_inc_2564 <dbl>,
#> #   visits_by_110635 <int>, visits_by_126614 <int>, visits_by_100751 <int>,
#> #   inst_110635 <chr>, inst_126614 <chr>, inst_100751 <chr>
```

Solution

3. Use `obj_name$var_name[1:10]` to print the first 10 observations of a variable

```
df_school$name[1:10]
#>  [1] "Bethel Regional High School" "Ayagina'ar Elitnaurvik"
#>  [3] "Kwigillingok School"         "Nelson Island Area School"
#>  [5] "Alakanuk School"             "Emmonak School"
#>  [7] "Hooper Bay School"           "Ignatius Beans School"
#>  [9] "Pilot Station School"        "Kotlik School"
```

Solution

4. Use combine() to print the first 3 observations of variables “school_type” & “name”

```
c(df_school$school_type[1:3],df_school$name[1:3])
#> [1] "public"                       "public"
#> [3] "public"                       "Bethel Regional High School"
#> [5] "Ayagina'ar Elitnaurvik"       "Kwigillingok School"
```

Missing values

Missing values

Missing values have the value `NA`

- `NA` is a special keyword, not the same as the character string `"NA"`

use `is.na()` function to determine if a value is missing

- `is.na()` returns a logical vector

```
is.na(5)
#> [1] FALSE
is.na(NA)
#> [1] TRUE
is.na("NA")
#> [1] FALSE
typeof(is.na("NA")) # example of a logical vector
#> [1] "logical"

nvector <- c(10,5,NA)
is.na(nvector)
#> [1] FALSE FALSE  TRUE
typeof(is.na(nvector)) # example of a logical vector
#> [1] "logical"

svector <- c("e","f",NA,"NA")
is.na(svector)
#> [1] FALSE FALSE  TRUE FALSE
```

Missing values are “contagious”

What does “contagious” mean?

- operations involving a missing value will yield a missing value

```
7>5
#> [1] TRUE
7>NA
#> [1] NA
sum(1,2,NA)
#> [1] NA
0==NA
#> [1] NA
2*c(0,1,2,NA)
#> [1]  0  2  4 NA
NA*c(0,1,2,NA)
#> [1] NA NA NA NA
```

Functions and missing values example, `table()`

`table()` function is useful for investigating categorical variables

```
str(df_event$event_type)
#>  chr [1:18680] "public hs" "public hs" "public hs" "public hs" "public hs" ..
table(df_event$event_type)
#>
#> 2yr college 4yr college       other  private hs   public hs
#>         951         531        2001        3774       11423
```

Functions and missing values example, `table()`

By default `table()` ignores `NA` values

```
#?table
str(df_event$school_type_pri)
#>  int [1:18680] NA NA NA NA NA 1 1 NA 1 NA ...
table(df_event$school_type_pri)
#> 
#>    1    2    5 
#> 3765    8    1
```

`useNA` argument controls if table includes counts of `NA`s. Allowed values:

- never (“no”) [DEFAULT VALUE]
- only if count is positive (“ifany”);
- even for zero counts (“always”)”

```
nrow(df_event)
#> [1] 18680
table(df_event$school_type_pri, useNA="always")
#> 
#>     1     2     5  <NA> 
#>  3765     8     1 14906
```

Broader point: Most functions that create descriptive statistics have options about how to treat missing values‘

- When investigating data, good practice to *always* show missing values

Subsetting using subset operators

Subsetting to Extract Elements

“Subsetting” refers to isolating particular elements of an object

Subsetting operators can be used to select/exclude elements (e.g., variables, observations)

- there are three subsetting operators: `[]` , `$` , `[[]]`
- these operators function differently based on vector types (e.g, atomic vectors, lists, data frames)

Wichham refers to number of “dimensions” in R objects

An atomic vector is a 1-dimensional object that contains n elements

```
x <- c(1.1, 2.2, 3.3, 4.4, 5.5)
str(x)
#>  num [1:5] 1.1 2.2 3.3 4.4 5.5
```

Lists are multi-dimensional objects

- Contains n elements; each element may contain a 1-dimensional atomic vector or a multi-dimensional list. Below list contains 3 dimensions

```
list <- list(c(1,2), list("apple", "orange"))
str(list)
#> List of 2
#>  $ : num [1:2] 1 2
#>  $ :List of 2
#>   ..$ : chr "apple"
#>   ..$ : chr "orange"
```

Data frames are 2-dimensional lists

- each element is a variable (dimension=columns)
- within each variable, each element is an observation (dimension=rows)

```
ncol(df_school)
#> [1] 26
nrow(df_school)
#> [1] 21301
```

Subset atomic vectors using []

Subsetting elements of atomic vectors

“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

Six ways to subset an atomic vector using `[]`

1. Using positive integers to return elements at specified positions
2. Using negative integers to exclude elements at specified positions
3. Using logicals to return elements where corresponding logical is `TRUE`
4. Empty `[]` returns original vector (useful for dataframes)
5. Zero vector [0], useful for testing data
6. If vector is “named,” use character vectors to return elements with matching names

1. Using positive integers to return elements at specified positions (subset atomic vectors using [])

Create atomic vector `x`

```
(x <- c(1.1, 2.2, 3.3, 4.4, 5.5))
#> [1] 1.1 2.2 3.3 4.4 5.5
str(x)
#>  num [1:5] 1.1 2.2 3.3 4.4 5.5
```

`[]` 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)

```
x[5] #return 5th element
#> [1] 5.5

x[c(3, 1)] #return 3rd and 1st element
#> [1] 3.3 1.1

x[c(4,4,4)] #return 4th element, 4th element, and 4th element
#> [1] 4.4 4.4 4.4

#Return 3rd through 5th element
x[3:5]
#> [1] 3.3 4.4 5.5
```

2. Using negative integers to exclude elements at specified positions (subset atomic vectors using [])

Before excluding elements based on position, investigate object

```
x
#> [1] 1.1 2.2 3.3 4.4 5.5

length(x)
#> [1] 5
str(x)
#>  num [1:5] 1.1 2.2 3.3 4.4 5.5
```

Use negative integers to exclude elements based on element position

```
x[-1] # exclude 1st element
#> [1] 2.2 3.3 4.4 5.5

x[c(3,1)] # 3rd and 1st element
#> [1] 3.3 1.1
x[-c(3,1)] # exclude 3rd and 1st element
#> [1] 2.2 4.4 5.5
```

3. Using logicals to return elements where corresponding logical is `TRUE` (subset atomic vectors using [])

```
x
#> [1] 1.1 2.2 3.3 4.4 5.5
```

When using `x[y]` to subset `x`, good practice to have `length(x)==length(y)`

```
length(x) # length of vector x
#> [1] 5
length(c(TRUE,FALSE,TRUE,FALSE,TRUE)) # length of y
#> [1] 5
length(x) == length(c(TRUE,FALSE,TRUE,FALSE,TRUE)) # condition true
#> [1] TRUE
x[c(TRUE,TRUE,FALSE,FALSE,TRUE)]
#> [1] 1.1 2.2 5.5
```

Recycling rules:

- in `x[y]`, if `x` is different length than `y`, R “recycles” length of shorter to match length of longer

```
length(c(TRUE,FALSE))
#> [1] 2
x
#> [1] 1.1 2.2 3.3 4.4 5.5
x[c(TRUE,FALSE)]
#> [1] 1.1 3.3 5.5
```

3. Using logicals to return elements where corresponding logical is `TRUE` (subset atomic vectors using [])

```
x
#> [1] 1.1 2.2 3.3 4.4 5.5
```

Note that a missing value (`NA`) in the index always yields a missing value in the output

```
x[c(TRUE, FALSE, NA, TRUE, NA)]
#> [1] 1.1  NA 4.4  NA
```

Return all elements of object `x` where element is greater than 3

```
x # pritn object X
#> [1] 1.1 2.2 3.3 4.4 5.5
x>3 # for each element of X, print T/F whether element value > 3
#> [1] FALSE FALSE  TRUE  TRUE  TRUE
x[x>3]
#> [1] 3.3 4.4 5.5
```

CRYSTAL/PATRICIA - ADD ONE OR TWO MORE PRACTICAL EXAMPLES?

- ?THIS APPROACH IS USED A LOT ON REAL DATA WORK; WANT STUDENTS TO BE ABLE TO SEE HOW IT IS USED. SOMETHING LIKE:?

```
names(df_school)
#>  [1] "state_code"         "school_type"        "ncessch"
#>  [4] "name"               "address"            "city"
```

4. Empty `[]` returns original vector (subset atomic vectors using [])

```r
x
#> [1] 1.1 2.2 3.3 4.4 5.5

x[]
#> [1] 1.1 2.2 3.3 4.4 5.5
```

This is useful for sub-setting data frames, as we will show below

5. Zero vector [0] (subset atomic vectors using [])

Zero vector, `x[0]`

- R interprets this as returning element 0

```
x[0]
#> numeric(0)
```

Wickham states:

- "This is not something you usually do on purpose, but it can be helpful for generating test data."

6. If vector is named, character vectors to return elements with matching names (subset atomic vectors using [])

Create vector `y` that has values of vector `x` but each element is named

```
x
#> [1] 1.1 2.2 3.3 4.4 5.5

(y <- c(a=1.1, b=2.2, c=3.3, d=4.4, e=5.5))
#>   a   b   c   d   e
#> 1.1 2.2 3.3 4.4 5.5
```

Return elements of vector based on name of element

- enclose element names in single `''` or double `""` quotes

```
#show element named "a"
y["a"]
#>   a
#> 1.1

#show elements "a", "b", and "d"
y[c("a", "b", "d" )]
#>   a   b   d
#> 1.1 2.2 4.4
```

Subsetting lists/data frames using []

Subsetting lists using []

Using `[]` operator to subset lists works the same as subsetting atomic vector

- Using `[]` with a list always returns a list

```r
list_a <- list(list(1,2),3,"apple")
str(list_a)
#> List of 3
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num 3
#>  $ : chr "apple"

#create new list that consists of elements 3 and 1 of list_a
list_b <- list_a[c(3, 1)]
str(list_b)
#> List of 2
#>  $ : chr "apple"
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2

#show elements 3 and 1 of object list_a
#str(list_a[c(3, 1)])
```

Subsetting data frames using []

Recall that a data frame is just a particular kind of list

- each element = a column = a variable

Using `[]` with a list always returns a list

- Using `[]` with a data frame always returns a data frame

Two ways to use `[]` to extract elements of a data frame

1. use "single index" `df_name[<columns>]` to extract columns (variables) based on element position number (i.e., column number)
2. use "double index" `df_name[<rows>, <columns>]` to extact particular rows and columns of a data frame

Subsetting data frames using [] to extract columns (variables) based on element position

Use "single index" `df_name[<columns>]` to extract columns (variables) based on element number (i.e., column number)

Examples [output omitted]

```
names(df_event)

#extract elements 1 through 4 (elements=columns=variables)
df_event[1:4]
df_event[c(1,2,3,4)]

str(df_event[1:4])
#extract columns 13 and 7
df_event[c(13,7)]
```

Subsetting Data Frames to extract columns (variables) and rows (observations) based on positionality

use “double index” syntax `df_name[<rows>, <columns>]` to extact particular rows and columns of a data frame

- often combined with sequences (e.g., `1:10`)

```
#Return rows 1-3 and columns 1-4
df_event[1:3, 1:4]
#> # A tibble: 3 x 4
#>   instnm     univ_id instst   pid
#>   <chr>        <int> <chr>  <int>
#> 1 UM Amherst  166629 MA     57570
#> 2 UM Amherst  166629 MA     56984
#> 3 UM Amherst  166629 MA     57105

#Return rows 50-52 and columns 10 and 20
df_event[50:52, c(10,20)]
#> # A tibble: 3 x 2
#>   event_state pct_tworaces_zip
#>   <chr>                  <dbl>
#> 1 MA                      1.98
#> 2 MA                      1.98
#> 3 MA                      1.98
```

Subsetting Data Frames to extract columns (variables) and rows (observations) based on positionality

use “double index” syntax `df_name[<rows>, <columns>]` to extact particular rows and columns of a data frame

recall that empty `[]` returns original object (output omitted)

```
#return original data frame
df_event[]

#return specific rows and all columns (variables)
df_event[1:5, ]

#return all rows and specific columns (variables)
df_event[, c(1,2,3)]
```

Use [] to extract data frame columns based on variable names

Selecting columns from a data frame by subsetting with `[]` and list of element names (i.e., variable names) enclose in quotes

"single index" approach extracts specific variables, all rows (output omittted)

```
df_event[c("instnm", "univ_id", "event_state")]
select(df_event,instnm,univ_id,event_state) # same same
```

"Double index" approach extracts specific variables and specific rows

- syntax `df_name[<rows>, <columns>]`

```
df_event[1:5, c("instnm", "event_state", "event_type")]
#> # A tibble: 5 x 3
#>   instnm      event_state event_type
#>   <chr>       <chr>       <chr>
#> 1 UM Amherst  MA          public hs
#> 2 UM Amherst  MA          public hs
#> 3 UM Amherst  MA          public hs
#> 4 UM Amherst  MA          public hs
#> 5 Stony Brook MA          public hs
```

Student exercises

Use subsetting operators from base R in extracting columns (variables), observations:

1. Use both “single index” and “double index” in subsetting to create a new dataframe by extracting the columns `instnm`, `event_date`, `event_type` from df_event. And show what columns (variables) are in the newly created dataframe.
2. Use subsetting to return rows 1-5 of columns `state_code`, `name`, `address` from df_school.

Solution to Student Exercises

Solution to 1

base R using subsetting operators

```
# single index
df_event_br <- df_event[c("instnm", "event_date", "event_type")]
#double index
df_event_br <- df_event[, c("instnm", "event_date", "event_type")]
names(df_event_br)
#> [1] "instnm"     "event_date" "event_type"
```

Solution to 2

base R using subsetting operators

```
df_school[1:5, c("state_code", "name", "address")]
#> # A tibble: 5 x 3
#>   state_code name                        address
#>   <chr>      <chr>                       <chr>
#> 1 AK         Bethel Regional High School 1006 Ron Edwards Memorial Dr
#> 2 AK         Ayagina'ar Elitnaurvik      106 Village Road
#> 3 AK         Kwigillingok School         108 Village Road
#> 4 AK         Nelson Island Area School   118 Village Road
#> 5 AK         Alakanuk School             9 School Road
```

Subsetting lists/data frames using [[]] and $

Subset single element from object using [[]] operator, atomic vectors

So far we have used `[]` to excract elements from an object

- Apply `[]` to atomic vector: returns atomic vector with elements you requested
- Apply `[]` to list: returns list with elements you requested

`[[]]` also extract elements from an object

- Applying `[[]]` to atomic vector gives same result as `[]`; that is, an atomic vector with element you request

```
(x <- c(1.1, 2.2, 3.3, 4.4, 5.5))
#> [1] 1.1 2.2 3.3 4.4 5.5

str(x[3])
#>  num 3.3

str(x[[3]])
#>  num 3.3
```

Subset single element from object using [[]] operator, lists

Create list

```r
list_a <- list(1:3, "a", 4:6)
str(list_a) # print list
#> List of 3
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : int [1:3] 4 5 6
```

- Apply `[]` to list: returns list with elements you requested

```r
str(list_a[1])
#> List of 1
#>  $ : int [1:3] 1 2 3
```

- Applying `[[]]` to list gives the “contents” of the list, rather than list itself

```r
str(list_a[[1]])
#>  int [1:3] 1 2 3
```

The “object” `[]` versus the “contents of the object” `[[]]`

START EDITING HERE!

Wickham “Advanced R” chapter 4.3 uses the “train metaphor” to differentiate the “object” from the “contents of the object” [LINK]

- We will use this train metaphor to understand the difference between a list (the object) and the contents of the list (contents of the object)

```
- Subsetting an object using `[[]]` returns the selected element itself, which
  - E.g., Using `[[]]` to select an element from a list that is a numeric vecto
    - Let `x` be a list with 3 elements (_Think of it as a train with 3 cars_)
    [![](https://d33wubrfki0l68.cloudfront.net/1f648d451974f0ed313347b78ba65389
    - `x[1]` will be a list containing the 1st element, which is a numeric vect
    - `x[[1]]` will be the numeric vector itself (i.e., _the objects within the
    [![](https://d33wubrfki0l68.cloudfront.net/aea9600956ff6fbbc29d8bd49124cca4
    - *Source: Subsetting from [R for Data Science](https://adv-r.hadley.nz/sub
- Object attributes are removed when using `[[]]`
  - E.g., Using `[[]]` on a named object returns just the selected element itse
```

Subset single element from object using [[]] operator

- We can only use `[[]]` to extract a single element rather than multiple elements

Wickham "Advanced R" chapter 4.3 [LINK HERE] uses "Train Metaphor" to differentiate list vs. contents of list

The list is the entire train. Create a list with three elements (three "carriages")

```
list_a <- list(1:3, "a", 4:6)
str(list_a)
#> List of 3
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : int [1:3] 4 5 6
```

When extracting element(s) of a list you have two options:

1. Extracting elements using `[]` always returns a smaller list (smaller train)

```
str(list_a[1]) # returns a list
#> List of 1
#>  $ : int [1:3] 1 2 3
```

2. Extracting element using `[[]]` returns contents of particular carriage
    - I say applying `[[]]` to a list or data frame returns a simpler object that moves up one level of hierarchy

```
str(list_a[[1]]) # returns an atomic vector
#>  int [1:3] 1 2 3
```

Subset single element from object using [[]] operator

In contrast to `[]`, we use `[[]]` to extract individual elements rather than multiple elements

- we could write `x[4]` or `x[4:6]`
- we could write `x[[4]]` but not `x[[4:6]]`

Subset single element from object using [[]] operator

Just like `[]` can use `[[]]` to return contents of **named** elements, specified using quotes

- syntax: `obj_name[["element_name"]]`

```
list_b <- list(var1=1:3, var2="a", var3=4:6)
str(list_b)
#> List of 3
#>  $ var1: int [1:3] 1 2 3
#>  $ var2: chr "a"
#>  $ var3: int [1:3] 4 5 6

str(list_b["var1"])
#> List of 1
#>  $ var1: int [1:3] 1 2 3

str(list_b[["var1"]])
#>  int [1:3] 1 2 3
```

Works the same with data frames

```
str(df_event["zip"])
#> tibble [18,680 x 1] (S3: tbl_df/tbl/data.frame)
#>  $ zip: chr [1:18680] "01002" "01007" "01020" "01020" ...

str(df_event[["zip"]])
#>  chr [1:18680] "01002" "01007" "01020" "01020" "01027" "01027" "01027"
```

Subset lists/data frames using $

`obj_name$element_name` shorthand operator for `obj_name[["element_name"]]`

```
str(list_b)
#> List of 3
#>  $ var1: int [1:3] 1 2 3
#>  $ var2: chr "a"
#>  $ var3: int [1:3] 4 5 6

list_b[["var1"]]
#> [1] 1 2 3
list_b$var1
#> [1] 1 2 3

str(list_b[["var1"]])
#>  int [1:3] 1 2 3
str(list_b$var1)
#>  int [1:3] 1 2 3
```

`df_name$var_name` : easiest way in base R to refer to variable in a data frame

```
str(df_event[["zip"]])
#>  chr [1:18680] "01002" "01007" "01020" "01020" "01027" "01027" "01027" ...
str(df_event$zip)
#>  chr [1:18680] "01002" "01007" "01020" "01020" "01027" "01027" "01027" ...
```

Subsetting data frames with [] combined with $

Subsetting Data Frames with [] combined with $

Combine `[]` with `$` to subset data frame same as `filter()` or `subset()`

Syntax: `df_name[df_name$var_name <condition>, ]`

- Note: Uses “double index” `df_name[<rows>, <columns>]` syntax
- **Cannot** use “single index” `df_name[<columns>]`

Examples (output omitted)

- All observations where the hich school received at least 1 visit from UC Berkeley (var= `visits_by_110635`) and all columns

```
df_school[df_school$visits_by_110635 >= 1, ]
```

- All obs where the high school received at least 1 visit from UC Berkeley and the first three columns

```
df_school[df_school$visits_by_110635 == 1, 1:3]
```

- All obs where the high school received at least 1 visit from UC Berkeley and variables “state_code” “school_type” “name”

```
df_school[df_school$visits_by_110635 == 1, c("state_code","school_type","name")
```

Subsetting Data Frames with [] combined with $

Combine `[]` with `$` to subset data frame same as `filter()` or `subset()`

- Syntax: `df_name[df_name$var_name <condition>, ]`
- Can be combined with `count()` or `nrow()` to avoid printing many rows

Count obs where high schools received at least 1 visit by Bama (100751) and at least one visit by Berkeley (110635)

- compare with `filter()` and `subset()` approaches

```
#[] combined with $ approach
count(df_school[df_school$visits_by_110635 >= 1
  & df_school$visits_by_100751 >= 1, ])
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   247

count(df_school[df_school[["visits_by_110635"]] >= 1
  & df_school[["visits_by_100751"]] >= 1, ])
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   247

df_school[]
#> # A tibble: 21,301 x 26
```

Subsetting Data Frames with [] and $, NA Observations

When sub-setting via `[]` combined with `$`, result will include:

- rows where condition is `TRUE`
- **as well as** rows with `NA` (missing) values for condition.

Task: How many events at public high schools with at least $50k median household income

- extracting observations via `[]` combined with `$`

```r
#num obs event_type=="public hs" and med_inc is missing
nrow(df_event[df_event$event_type == "public hs"
  & is.na(df_event$med_inc)==1 , ])
#> [1] 75

#num obs event_type=="public hs" & med_inc is not NA & med_inc >= $50,000
nrow(df_event[df_event$event_type == "public hs"
  & is.na(df_event$med_inc)==0 & df_event$med_inc>=50000 , ])
#> [1] 9941

#num obs event_type=="public hs" and med_inc >= $50,000
nrow(df_event[df_event$event_type == "public hs"
  & df_event$med_inc>=50000 , ])
#> [1] 10016
```

Subsetting Data Frames with [] and $, NA Observations

subset using `[]` combined with `$`, result includes:

- rows where condition `TRUE`; **AND** rows with `NA` for condition

Base R filter using `subset()` excludes rows with `NA` for condition

```
#num obs event_type=="public hs" and med_inc is missing
nrow(subset(df_event, event_type == "public hs" & is.na(med_inc)==1))
#> [1] 75
#num obs event_type=="public hs" & med_inc is not NA & med_inc >= $50,000
nrow(subset(df_event, event_type == "public hs" & is.na(med_inc)==0
  & med_inc>=50000))
#> [1] 9941
#num obs event_type=="public hs" & med_inc >= $50,000
nrow(subset(df_event, event_type == "public hs"
  & med_inc>=50000))
#> [1] 9941
```

Tidyverse `filter()` excludes rows with `NA` for condition.

```
#num obs event_type=="public hs" and med_inc is missing
nrow(filter(df_event, event_type == "public hs", is.na(med_inc)==1))
#> [1] 75
#num obs event_type=="public hs" & med_inc is not NA & med_inc >= $50,000
nrow(filter(df_event, event_type == "public hs", is.na(med_inc)==0, med_inc>=500
#> [1] 9941
#num obs event_type=="public hs" & med_inc >= $50,000
nrow(filter(df_event, event_type == "public hs", med_inc>=50000))
```

Subsetting Data Frames with [] and $, NA Observations

To exclude rows where condition is `NA` if subset using `[]` combined w/ `$`

- use `which()` to ask only for values where condition evaluates to `TRUE`
- `which()` returns position numbers for elements where condition is `TRUE`

```
#?which
c(TRUE,FALSE,NA,TRUE)
#> [1]  TRUE FALSE    NA  TRUE
str(c(TRUE,FALSE,NA,TRUE))
#>  logi [1:4] TRUE FALSE NA TRUE
which(c(TRUE,FALSE,NA,TRUE))
#> [1] 1 4
```

Task: Count events at public HS with at least $50k median household income?

```
#Tidyverse, filter()
nrow(filter(df_event, event_type == "public hs" & med_inc>=50000))
#> [1] 9941

#Base R, `[]` combined with `$`; without which()
nrow(df_event[df_event$event_type == "public hs" & df_event$med_inc>=50000, ])
#> [1] 10016

#Base R, `[]` combined with `$`; with which()
nrow(df_event[which(df_event$event_type == "public hs"
  & df_event$med_inc>=50000), ])
#> [1] 9941
```

Student Exercises

Subsetting Data Frames with (1) [] and $; (2) subset() and filter():

1. Show how many public high schools in California with at least 50% Latinx (hispanic in data) student enrollment from df_school.
2. Show how many out-state events at public high schools with more than $30K median from df_event (do not forget to exclude missing values).

Solution to Student Exercises

Solution to 1

base R using [] and $

```
df_school_br1<- df_school[df_school$school_type == "public"
                  & df_school$pct_hispanic >= 50
                  & df_school$state_code == "CA", ]
nrow(df_school_br1)
#> [1] 713
```

base R using `subset()` function

```
df_school_br2 <- subset(df_school, school_type == "public"
                  & pct_hispanic >= 50
                  & state_code == "CA" )
nrow(df_school_br2)
#> [1] 713
```

tidyverse using `filter()` function

```
df_school_tv <- df_school %>% filter(school_type == "public"
                  & pct_hispanic >= 50
                  & state_code == "CA" )
nrow(df_school_tv)
#> [1] 713
```

Solution to Student Exercises

Solution to 2:

base R using [] and $ (NA included)

```
# use is.na to exclude NA
nrow(df_event[df_event$event_type == "public hs" & df_event$event_inst =="Out-S
               & df_event$med_inc > 30000 & is.na(df_event$med_inc) ==0, ])
#> [1] 7784

# use which to exclude NA
nrow(df_event[which(df_event$event_type == "public hs" & df_event$event_inst ==
               & df_event$med_inc > 30000 ), ])
#> [1] 7784
```

base R using `subset()` function (NA excluded)

```
nrow(subset(df_event, event_type == "public hs"
                          & event_inst =="Out-State"& df_event$med_inc > 30000 ))
#> [1] 7784
```

tidyverse using `filter()` function (NA excluded)

```
count(filter(df_event, event_type == "public hs"
                          & event_inst =="Out-State" & df_event$med_inc > 30000 )
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1  7784
```

Subsetting using subset() function

Subset function

The `subset()` is a base R function and easiest way to "filter" observations

- can also used `subset()` to select variables
- Like tidyverse `filter()`, `subset()` can be combined with:
  - with assignment (`<-`) to create new objects
  - with `count()` to count number of observations that satisfy criteria

```
?subset
```

Syntax [when object is data frame]: **subset(x, subset, select, drop = FALSE)**

- `x` is object to be subset
- `subset` is the logical expression(s) (evaluates to `TRUE/FALSE`) indicating elements (rows) to keep
- `select` indicates columns to select from data frame (if argument is not used default will keep all columns)
- `drop` to preserve original **dimensions** [SKIP]
  - cane take values `TRUE` or `FALSE`; default is `FALSE`
  - only need to worry about dataframes when subset output is single column

Subset function, examples

Using `df_school`, show all public high schools that are at least 50% Latinx (var=`pct_hispanic`) student enrollment in California

- Using tidyverse `filter()` [output omitted]

```
filter(df_school, school_type == "public", pct_hispanic >= 50,
   state_code == "CA")

filter(df_school, school_type == "public" & pct_hispanic >= 50
   & state_code == "CA") # same as above
```

- Using base R, `subset()` [output omitted]

```
#public high schools with at least 50% Latinx student enrollment
subset(df_school, school_type == "public" & pct_hispanic >= 50
     & state_code == "CA")
```

Subset function, examples

Count all CA public high schools that are at least 50% Latinx

- Can wrap `filter()` or `subset()` within `count()` to count number of observations that satisfy criteria

```
#filter()
count(filter(df_school, school_type == "public", pct_hispanic >= 50,
   state_code == "CA"))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   713
count(filter(df_school, school_type == "public" & pct_hispanic >= 50
   & state_code == "CA"))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   713

#subset()
count(subset(df_school, school_type == "public" & pct_hispanic >= 50
     & state_code == "CA"))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   713
```

Subset function, examples

Note that both `filter()` and `subset()` identify the number of observations for which the condition is `TRUE`

```
count(filter(df_school, TRUE))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1 21301
count(subset(df_school, TRUE))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1 21301

count(filter(df_school, FALSE))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1     0
count(subset(df_school, FALSE))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1     0
```

Subset function, examples

Count all CA public high schools that are at least 50% Latinx and received at least 1 visit from UC Berkeley (var= `visits_by_110635`)

```
#filter()
count(filter(df_school, school_type == "public", pct_hispanic >= 50,
   state_code == "CA", visits_by_110635 >= 1))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   100

#subset()
count(subset(df_school, school_type == "public" & pct_hispanic >= 50
  & state_code == "CA" & visits_by_110635 >= 1))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   100
```

Subset function, examples

`subset()` can also use %in% operator, which is more efficient version of **OR** operator `|`

- Count number of schools from MA, ME, or VT that received at least one visit from University of Alabama (var= `visits_by_100751`)

```
#filter()
count(filter(df_school, state_code %in% c("MA","ME","VT"),
  visits_by_100751 >= 1))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   108

#subset()
count(subset(df_school, state_code %in% c("MA","ME","VT")
  & visits_by_100751 >= 1))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   108
```

Subset function, examples

Use the `select` argument within `subset()` to keep selected variables

- syntax: `select = c(var_name1,var_name2,...,var_name_n)`

Subset all CA public high schools that are at least 50% Latinx **AND** only keep variables `name` and `address`

```
subset(df_school, school_type == "public" & pct_hispanic >= 50
             & state_code == "CA", select = c(name, address))
#> # A tibble: 713 x 2
#>    name                        address
#>    <chr>                       <chr>
#>  1 Tustin High                 1171 El Camino Real
#>  2 Bell Gardens High           6119 Agra St.
#>  3 Santa Ana High              520 W. Walnut
#>  4 Warren High                 8141 De Palma St.
#>  5 Hollywood Senior High       1521 N. Highland Ave.
#>  6 Venice Senior High          13000 Venice Blvd.
#>  7 Sequoia High                1201 Brewster Ave.
#>  8 Santa Barbara Senior High 700 E. Anapamu St.
#>  9 Santa Paula High            404 N. Sixth St.
#> 10 Azusa High                  240 N. Cerritos Ave.
#> # ... with 703 more rows
```

Subset function, examples

Combine `subset()` with assignment (`<-`) to create a new data frame

Create a new date frame of all CA public high schools that are at least 50% Latinx **AND** only keep variables `name` and `address`

```
df_school_v2 <- subset(df_school, school_type == "public" & pct_hispanic >= 50
  & state_code == "CA", select = c(name, address))

head(df_school_v2, n=5)
#> # A tibble: 5 x 2
#>   name                     address
#>   <chr>                    <chr>
#> 1 Tustin High              1171 El Camino Real
#> 2 Bell Gardens High        6119 Agra St.
#> 3 Santa Ana High           520 W. Walnut
#> 4 Warren High              8141 De Palma St.
#> 5 Hollywood Senior High 1521 N. Highland Ave.

nrow(df_school_v2)
#> [1] 713
```

Student Exercises

Compare tidyverse to subset() from base R in extracting columns (variables), observations:

1. Use both base R and tidyverse to create a new dataframe by extracting the columns `instnm`, `event_date`, `event_type` from df_event. And show what columns (variables) are in the newly created dataframe.
2. Use both base R and tidyverse to create a new dataframe from df_school that includes out-of-state public high schools with 50%+ Latinx student enrollment that received at least one visit by the University of California Berkeley (var= visits_by_110635). And count the number of observations.
3. Use both base R and tidyverse to count the number of public schools from CA, FL or MA that received one or two visits from UC Berkeley from df_school.
4. Use base R to subset all public out-of-state high schools visited by University of California Berkeley that enroll at least 50% Black students, and only keep variables "state_code", "name" and "zip_code" .

Solution to Student Exercises

Solution to 1

base R using `subset()` function

```
df_event_br <- subset(df_event, select=c(instnm, event_date, event_type))
names(df_event_br)
#> [1] "instnm"     "event_date" "event_type"
```

tidyverse using `select()` function

```
df_event_tv <- select(df_event, instnm, event_date, event_type)
names(df_event_tv)
#> [1] "instnm"     "event_date" "event_type"
```

Solution to 2

base R using `subset()` function

```
df_school_br <- subset(df_school, state_code != "CA" & school_type == "public"
                        & pct_hispanic >= 50 & visits_by_110635 >=1 )
nrow(df_school_br)
#> [1] 10
```

tidyverse using `filter()` function

```
df_school_tv <- filter(df_school, state_code != "CA" & school_type == "public"
                       & pct_hispanic >= 50 & visits_by_110635 >=1 )
nrow(df_school_tv)
#> [1] 10
```

Solution to Student Exercises

Solution to 3

base R using `subset()` function

```
count(subset(df_school, state_code %in% c("CA", "FL", "MA")
             & school_type == "public" & visits_by_110635 %in% c(1,2) ))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   246
```

tidyverse using `filter()` function

```
count(filter(df_school, state_code %in% c("CA", "FL", "MA")
             & school_type == "public" & visits_by_110635 %in% c(1,2) ))
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   246
```

Solution to Student Exercises

Solution to 4

base R using `subset()` function

```
subset(df_school, school_type == "public" & state_code != "CA"
       & visits_by_100751 >= 1 & pct_hispanic >= 50,
       select = c(state_code, name, zip_code))
#> # A tibble: 73 x 3
#>    state_code name                                 zip_code
#>    <chr>      <chr>                                <chr>
#>  1 AZ         Agua Fria High School                85323
#>  2 AZ         Desert Edge High School              85338
#>  3 AZ         Tempe High School                    85281
#>  4 AZ         Westview High School                 85353
#>  5 AZ         Apollo High School                   85302
#>  6 AZ         South Mountain High School           85040
#>  7 AZ         Tolleson Union High School           85353
#>  8 CO         THORNTON HIGH SCHOOL                 80229
#>  9 CO         MARTIN LUTHER KING JR. EARLY COLLEGE 80249
#> 10 CO         BATTLE MOUNTAIN HIGH SCHOOL          81620
#> # ... with 63 more rows
```

Sorting data

Base R `sort()` for vectors

`sort()` is a base R function that sorts vectors

Syntax: `sort(x, decreasing=FALSE, ...)`

- where x is object being sorted
- By default it sorts in ascending order (low to high)
- Need to set decreasing argument to `TRUE` to sort from high to low

```
#?sort()
x<- c(31, 5, 8, 2, 25)
sort(x)
#> [1]  2  5  8 25 31
sort(x, decreasing = TRUE)
#> [1] 31 25  8  5  2
```

Base R `order()` for dataframes

`order()` is a base R function that sorts vectors

- Syntax: `order(..., na.last = TRUE, decreasing = FALSE)`
- where `...` are variable(s) to sort by
- By default it sorts in ascending order (low to high)
- Need to set decreasing argument to `TRUE` to sort from high to low

Descending argument only works when we want either one (and only) variable descending or all variables descending (when sorting by multiple vars)

- use `-` when you want to indicate which variables are descending while using the default ascending sorting

```r
df_event[order(df_event$event_date), ]
df_event[order(df_event$event_date, df_event$total_12), ]

#sort descending via argument
df_event[order(df_event$event_date, decreasing = TRUE), ]
df_event[order(df_event$event_date, df_event$total_12, decreasing = TRUE), ]

#sorting by both ascending and descending variables
df_event[order(df_event$event_date, -df_event$total_12), ]
```

Compare tidyverse to base r, sorting

-Create a new dataframe from df_events that sorts by ascending by `event_date`, ascending `event_state`, and descending `pop_total` .

tidyverse

```r
df_event_tv <- arrange(df_event, event_date, event_state, desc(pop_total))
```

base R using `order()` function

```r
df_event_br1 <- df_event[order(df_event$event_date, df_event$event_state,
                               -df_event$pop_total), ]
```