Lecture 2: Vectors, Lists and Data Frames (base R)

Harris Coding Camp – Accelerated Track

Summer 2022

Today's agenda:

- Vectors
- Data types
- `data.frames` and `tibbles`

Vectors: the foundational data structure in R

Vectors store an arbitrary[1] number of *same* type items

Use c() to build vectors

```
# numeric vector of length 6
my_numbers <- c(1, 2, 3, 4, 5, 6)
my_numbers
```

```
## [1] 1 2 3 4 5 6
```

```
# character vector of length 3
my_characters <- c("public", "policy", "101")
my_characters
```

```
## [1] "public" "policy" "101"
```

[1] Within limits determined by hardware

Vectors are the smallest unit of data in R

```r
# vectors of length 1
i_am_a_vector <- 12.0
as_am_i <- TRUE

is.vector(i_am_a_vector)
```

```
## [1] TRUE
```

```r
is.vector(as_am_i)
```

```
## [1] TRUE
```

The c() function combines vectors

```r
x <- c(c(1, 2, 3), c(4, 5, 6))
x
```

```
## [1] 1 2 3 4 5 6
```

```r
y <- c(x, 2022)
y
```

```
## [1]    1    2    3    4    5    6 2022
```

i.e. c() "adds" elements to a vector

```
z <- c("Bo", "Cynthia", "David")
z
```

```
## [1] "Bo"      "Cynthia" "David"
```

```
z <- c(z, "Ernesto")
z
```

```
## [1] "Bo"      "Cynthia" "David"   "Ernesto"
```

```
z <- c("Amelia", z)
z
```

```
## [1] "Amelia"  "Bo"      "Cynthia" "David"   "Ernesto"
```

Technical detail: creates new objects in memory!

Create vectors of *seq()uential* numbers with `:` and `seq()`

```r
too_much_typing <- c(2, 3, 4, 5)
2:5
```

```
## [1] 2 3 4 5
```

```r
seq(2, 5)
```

```
## [1] 2 3 4 5
```

```r
seq(from = 2, to = 5, by = 1)
```

```
## [1] 2 3 4 5
```

Create rep()eated vectors

```r
too_much_typing <- c("a", "a", "a", "a")
rep("a", 4)
```

```
## [1] "a" "a" "a" "a"
```

Can you explain what is going on?

```
rep(c("a", 5), 4)
```

```
## [1] "a" "5" "a" "5" "a" "5" "a" "5"
```

```
rep(c("a", 5), each = 4)
```

```
## [1] "a" "a" "a" "a" "5" "5" "5" "5"
```

Creating placeholder vectors of a given type

```r
too_much_typing <- c("", "", "", "", "", "")

vector("character", length = 5)
```

```
## [1] "" "" "" "" ""
```

```r
# or this shorter short-cut.
character(5)
```

```
## [1] "" "" "" "" ""
```

Creating placeholder vectors of a given type

```r
# 1 million 0s
my_integers <- integer(1000000)
head(my_integers)
```

```
## [1] 0 0 0 0 0 0
```

```r
# 1 million FALSEs
my_lgl <- logical(1e6)
head(my_lgl, 3)
```

```
## [1] FALSE FALSE FALSE
```

Create *random* data following a distribution

```r
# Randomly choose 3 numbers from a Normal distribution
(my_random_normals <- rnorm(3))
```

```
## [1] 1.0799690 0.4404190 0.2834663
```

```r
# Randomly choose 4 numbers from a Uniform distribution
(my_random_uniforms <- runif(4))
```

```
## [1] 0.2766011 0.4250474 0.7822993 0.2178838
```

The pattern is **r**`distribution` (**r**`unif`, **r**`norm`, **r**`f`, **r**`chisq`)

Vectorization

R is a vectorized language

"Vectorized" means do something to a vector **element by element**

- In non-vectorized languages (e.g. Python, Java), you use a loop.
- Other vectorized languages include: Julia, MATLAB, numpy library in Python

Math is vectorized

Do the operation **element by element**

```
my_numbers <- 1:6
# 1 + 1,
# 2 + 2,
# 3 + 3,
# 4 + 4,
# 5 + 5,
# 6 + 6
my_numbers + my_numbers
```

```
## [1]  2  4  6  8 10 12
```

Add a single value to a vector

Do the operation **element by element**

- if one vector is “short”, recycle it’s elements.

```
# 1 + 6,
# 2 + 6,
# 3 + 6,
# 4 + 6,
# 5 + 6,
# 6 + 6
my_numbers + 6
```

```
## [1]  7  8  9 10 11 12
```

Many built-in functions are vectorized.

- This may remind Excel users of "dragging" a function

```
# sqrt(1)
# sqrt(2)
# sqrt(3)
# sqrt(4)
# sqrt(5)
# sqrt(6)
sqrt(my_numbers)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.4494
```

You try: Guess the output before running the code

```
my_numbers <- 1:6
my_numbers - my_numbers
my_numbers * my_numbers
my_numbers / my_numbers
```

```
a_vector <- rnorm(6)
sqrt(a_vector)
round(a_vector, 2)
```

What happens when the vectors aren't the same size?

```r
a <- 1:6 + 1:3
a
```

```
## [1] 2 4 6 5 7 9
```

Warning: Vector recycling

The shorter vector re-starts from it's beginning.

```r
# 1 + 1,
# 2 + 2,
# 3 + 3,
# 4 + 1, -- '1' is 'recycled'
# 5 + 2, -- '2' is 'recycled'
# 6 + 3  -- '3' is 'recycled'
c(1, 2, 3, 4, 5, 6) + c(1, 2, 3)
```

```
## [1] 2 4 6 5 7 9
```

Some times we get a warning . . .

```
x <- c(1, 2, 3, 4, 5, 6) + c(1, 2, 3, 4, 5)
```

```
## Warning in c(1, 2, 3, 4, 5, 6) + c(1, 2, 3, 4, 5): longe
## multiple of shorter object length
```

. . . but not if vector lengths are multiples

```
x <- c(1, 2, 3, 4, 5, 6) + c(1, 2, 3)
```

Conditional operators are vectorized

```
# 1 > 1,
# 2 > 1,
# 3 > 3,
# 4 > 3,
# 5 > pi,
# 6 > pi

my_numbers > c(1, 1, 3, 3, pi, pi)
```

```
## [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE
```

Conditional operators are vectorized

```r
x <- c(1, 3, 6, 10)
# 4 is "recycled" think c(4, 4, 4)
x > 4
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

```r
x == 3
```

```
## [1] FALSE  TRUE FALSE FALSE
```

Warning: Vector recycling still an issue

You want to see if a value is equal to 1 or 6

```r
# FAIL
x == c(1, 6)
```

```
## [1]  TRUE FALSE FALSE FALSE
```

You want to see if a value is equal to 1 or 6

```r
# FAIL
x == c(1, 6)
```

```
## [1]  TRUE FALSE FALSE FALSE
```

```r
# Success!
x == 1 | x == 6
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

Boolean operators are vectorized

```r
# TRUE & TRUE
# TRUE & FALSE
# FALSE & TRUE
# FALSE & FALSE
rep(c(TRUE, FALSE), each = 2) & rep(c(TRUE, FALSE), 2)
```

```
## [1]  TRUE FALSE FALSE FALSE
```

Boolean operators are vectorized

```r
# TRUE | TRUE
# TRUE | FALSE
# FALSE | TRUE
# FALSE | FALSE
rep(c(TRUE, FALSE), each = 2) | rep(c(TRUE, FALSE), 2)
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

xor() exists in baseR as a function!

See section 13.3 of R4DS

```r
# TRUE | TRUE.
# TRUE | FALSE
# FALSE | TRUE
# FALSE | FALSE
xor(rep(c(TRUE, FALSE), each = 2), rep(c(TRUE, FALSE), 2))
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

Working with characters

- `paste0()` is a function that combines character vectors
- `str_c()` is a tidyverse cousin

```
paste0(c("a", "b", "c"), c("x", "y", "z"))
```

```
## [1] "ax" "by" "cz"
```

```
paste0("a", "w", "e", "s", "o", "m", "e")
```

```
## [1] "awesome"
```

Subseting Vectors

Accessing Elements by Index with Brackets [

```
z
```

```
## [1] "Amelia"   "Bo"        "Cynthia" "David"    "Ernesto"
```

```
z[3]
```

```
## [1] "Cynthia"
```

```
z[c(1, 2, 3)]
```

```
## [1] "Amelia"   "Bo"        "Cynthia"
```

Reassign Elements by Index

```r
z[1] <- "Arthur"
z[1:3]
```

```
## [1] "Arthur"  "Bo"       "Cynthia"
```

Excluding Elements by Index

Using a negative sign, returns everything *except* the selected one(s):

```
my_letters <- c("a", "b", "c", "d", "e")
# get all numbers besides the 1st
my_letters[-1]
```

```
## [1] "b" "c" "d" "e"
```

```
# get all numbers besides the 4th and 5th
my_letters[-c(4, 5)]
```

```
## [1] "a" "b" "c"
```

Try it out:

```r
a <- 0:9
```

- subset the number 7
- subset all the numbers not equal to 8 or 9
- subset all the even numbers

Try it out:

```
a[8]
```

```
## [1] 7
```

```
a[-c(9, 10)]
```

```
## [1] 0 1 2 3 4 5 6 7
```

```
a[c(1, 3, 5, 7, 9)]
```

```
## [1] 0 2 4 6 8
```

Accessing Element by Logical Vector

```
logical_index <-
  # 1,    2,    3,     4,     5,    6,
  c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE)
```

```
# same as my_numbers[c(1, 2, 5)]
my_numbers[logical_index]
```

```
## [1] 1 2 5
```

Accessing Element by Logical Vector

```
# 1  subset TRUE
# 2  subset TRUE
# 3  subset FALSE
# 4  subset FALSE
# 5  subset TRUE
# 6  subset FALSE
my_numbers[logical_index]
```

```
## [1] 1 2 5
```

Accessing elements that meet a condition (i.e. Logical Vector)

```
x <- c(-3, -2, 15, 11, -12, 13)
x < 0
```

```
## [1]  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

```
x[x < 0]
```

```
## [1]  -3  -2 -12
```

Reassigning elements that meet a condition

```r
# Replace elements which meet the condition with 0
x[x < 0] <- 0
x
```

```
## [1]  0  0 15 11  0 13
```

Try it out – with conditional expressions / logical vectors:

```r
a <- 0:9
```

- subset the number 7
- subset all the numbers not equal to 8 AND are not equal to 9
- subset all the even numbers

Try it out:

```r
a[a == 7]
```

```
## [1] 7
```

```r
a[a != 8 & a != 9]
```

```
## [1] 0 1 2 3 4 5 6 7
```

```r
a[a %% 2 == 1]
```

```
## [1] 1 3 5 7 9
```

Functions that reduce vectors

`length()` tells you how many items in the vector

```
# letters is the English alphabet
head(letters)
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
length(letters)
```

```
## [1] 26
```

Summarize your data

```
a_vector <- c(1, 3, 5, 7, 15)
sum(a_vector)       # add all numbers
mean(a_vector)      # find the mean
median(a_vector)    # find the median
sd(a_vector)        # find the standard deviation
```

Generalize logical operators

Generalize | with `any()`

```
# 1 > 12 | 3 > 12 | 5 > 12 | 7 > 12 | 15 > 12
any(a_vector > 12)
```

```
## [1] TRUE
```

Generalize & with `all()`

```
# 1 == 5 & 3 == 5 & 5 == 5 & 7 == 5 & 15 == 5
all(a_vector == 5)
```

```
## [1] FALSE
```

Functions that reduce vectors

Useful functions to summarize data

- Center: mean(), median()
- Spread: sd(), IQR(), mad()
- Range: min(), max(), quantile()
- Position: first(), last(), nth(),
- Count: n(), n_distinct()
- Logical: any(), all()

Data Types

What is going on here?

```r
a <- "4"
b <- 5
a * b
```

Error in a * b : non-numeric argument to binary operator

What is going on here?

The error we got when we tried `a * b` was because `a` is a character:

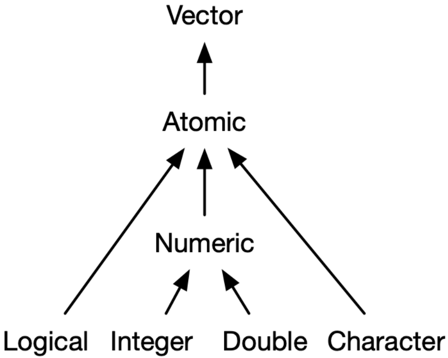
```
a <- "4"
b <- 5
a * b # invalid calculation
```

R does not multiply *non-numeric* vectors!

Data types

R has four primary types of atomic vectors

- these determine how R stores the data in memory



[2]

[2] Image from https://adv-r.hadley.nz/vectors-chap.html

Data types

logicals, also known as booleans

```
type_logical <- FALSE
type_logical <- TRUE
```

integer and double, together are called: numeric

```
type_integer <- 1000
type_double <- 1.0
```

character, need to use " " to include the text

```
type_character <- "abbreviated as chr"
type_character <- "also known as a string"
```

Testing types with is.<type>()

```
x <- "1"
typeof(x) # for atomic vectors, same as class()
```

```
## [1] "character"
```

```
is.integer(x)
```

```
## [1] FALSE
```

```
is.character(x)
```

```
## [1] TRUE
```

technical: `typeof()` returns types built-in to R. When we develop new structures, we can assign our own `class()`; class allows for more nuanced results (more later). You might also see people use `mode()`, which is nearly a synonym of `typeof()`.

Reassign types on the fly as.<type>()

The error we got when we tried `a * b` was because `a` is a character:

```
a <- "4"
b <- 5
as.numeric(a) * b
```

```
## [1] 20
```

What Happens When You Mix Types Inside a Vector?

```r
c(4, "harris")
c(TRUE, "harris")
c(TRUE, 5)
c(FALSE, 100)
```

Character > Numeric > Logical

```
# Numbers can be coerced into Characters.
c(4, "harris")
```

```
## [1] "4"      "harris"
```

```
# Logicals are coercible to numeric or character.
c(TRUE, "harris")
```

```
## [1] "TRUE"   "harris"
```

```
c(TRUE, 5)
```

```
## [1] 1 5
```

```
c(FALSE, 100)
```

```
## [1]   0 100
```

Automatic coercion

We make use of logical coercion a lot.

What do you think the following code will return?

```
TRUE + 10
sum(c(TRUE, TRUE, FALSE, FALSE, TRUE))
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

Automatic coercion

```r
TRUE + 10
```

```
## [1] 11
```

```r
sum(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

```
## [1] 3
```

```r
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

```
## [1] 0.6
```

Automatic coercion only works from simple to complex

```r
# Since Character > Numeric R won't turn "4" into 4
"4" + 4
```

Error in “4” + 4 : non-numeric argument to binary operator

```r
# But, it will turn 4 into "4"
paste0("4", 4)
```

```
## [1] "44"
```

NAs introduced by coercion

R does not know how to turn the string “unknown” into an integer. So, it uses `NA` which is how R represents *missing* or *unknown* values.

```
as.integer("Unknown")
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

NAs are contagious

`NA` could be **anything** so the output is also *unknown*

```
NA + 4
```

```
## [1] NA
```

```
max(c(NA, 4, 1000))
```

```
## [1] NA
```

```
mean(c(NA, 3, 4, 5))
```

```
## [1] NA
```

```
4 == NA
```

```
## [1] NA
```

Some functions let you ignore the missing values

```
b <- c(NA, 3, 4, 5)
sum(b)
```

```
## [1] NA
```

```
sum(b, na.rm = TRUE)
```

```
## [1] 12
```

```
mean(b, na.rm = TRUE)
```

```
## [1] 4
```

Do you remember how to test for missing values?

```r
x <- c(4, NA)
x == NA
```

```
## [1] NA NA
```

Test for NAs with is.na()

```
x <- c(4, NA)
is.na(x)
```

```
## [1] FALSE  TRUE
```

```
!is.na(x)
```

```
## [1]  TRUE FALSE
```

Lists

What do we do we when we want to store different types?

```
c(TRUE, c(2,2,2), "Last")
```

```
## [1] "TRUE" "2"    "2"    "2"    "Last"
```

What do we do we when we want to store different types?

Use lists!

Lists are useful building blocks for:

- data frames / tibbles
- output from statistical models

```
# vector coercion
typeof(c(1, "a", TRUE))
```

```
## [1] "character"
```

```
# no-coercion
typeof(list(1, "a", TRUE))
```

```
## [1] "list"
```

List

We can name the objects in a list for easy reference.

```
my_list <- list(can = c(TRUE, TRUE),
                hold = c(2, 2),
                anything = c("last", "last"))
str(my_list)
```

```
## List of 3
##  $ can     : logi [1:2] TRUE TRUE
##  $ hold    : num [1:2] 2 2
##  $ anything: chr [1:2] "last" "last"
```

... Hey that looks like a data frame!

Lists

[[and $ pull out a single object from a list by name or location.

```
my_list[[2]]
```

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

```
typeof(my_list[[2]])
```

```
## [1] "double"
```

```
my_list$anything
```

```
## [1] "last" "last"
```

```
typeof(my_list$anything)
```

```
## [1] "character"
```

Lists

We can also subset a [list and retain a list

```
my_list[c(1,3)]
```

```
## $can
## [1] TRUE TRUE
##
## $anything
## [1] "last" "last"
```

```
# this is still a list
typeof(my_list[c(1,3)])
```

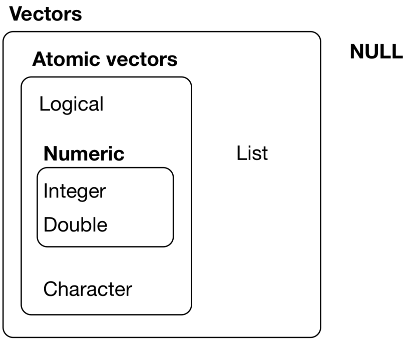
```
## [1] "list"
```

Lists are vectors?

- lists are still vectors, just not **atomic**

```
is.vector(my_list)
```

```
## [1] TRUE
```



3

[3] image from https://r4ds.had.co.nz/vectors.html

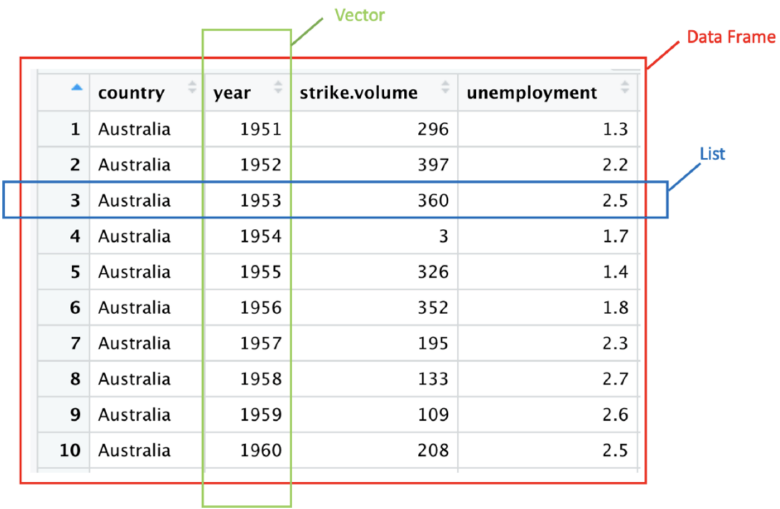
Empty list creation

To create an empty list of a given size use `vector()`

```
empty_list <- vector("list", 10)
```

Data Frames are lists with specific properties

Data Frame, Vector, List



	country	year	strike.volume	unemployment
1	Australia	1951	296	1.3
2	Australia	1952	397	2.2
3	Australia	1953	360	2.5
4	Australia	1954	3	1.7
5	Australia	1955	326	1.4
6	Australia	1956	352	1.8
7	Australia	1957	195	2.3
8	Australia	1958	133	2.7
9	Australia	1959	109	2.6
10	Australia	1960	208	2.5

- Row: holds elements of different types (e.g. numeric, character, logical)
- Column: store elements of the same type

We like `tidy` data

- Row: A distinct observation
- Column: A feature or characteristic of that observation

Vector
Data Frame
List

	country	year	strike.volume	unemployment
1	Australia	1951	296	1.3
2	Australia	1952	397	2.2
3	Australia	1953	360	2.5
4	Australia	1954	3	1.7
5	Australia	1955	326	1.4
6	Australia	1956	352	1.8
7	Australia	1957	195	2.3
8	Australia	1958	133	2.7
9	Australia	1959	109	2.6
10	Australia	1960	208	2.5

What columns define a distinct observation?

Columns are vectors

We can create a tibble or data.frame manually

- To test out code on a simpler tibble
- To organize data from a simulation

```
care_data <- tibble(
  id = 1:5,
  n_kids = c(2, 4, 1, 1, NA),
  child_care_costs = c(1000, 3000, 300, 300, 500),
  random_noise = rnorm(5, sd = 5)*30
)
```

Could create the same code with `data.frame()`

Ta-da

Take a look at our data set care_data:

```
care_data
```

```
## # A tibble: 5 x 4
##      id n_kids child_care_costs random_noise
##   <int>  <dbl>            <dbl>        <dbl>
## 1     1      2             1000         36.1
## 2     2      4             3000        -24.6
## 3     3      1              300       -267. 
## 4     4      1              300        385. 
## 5     5     NA              500       -105. 
```

Rows are lists

(we make use of this idea less often.)

```
bind_rows(
  list(id = 1, n_kids = 2, child_care_costs = 1000),
  list(id = 2, n_kids = 4, child_care_costs = 3000),
  list(id = 5, n_kids = NA, child_care_costs = 500)
)
```

```
## # A tibble: 3 x 3
##      id n_kids child_care_costs
##   <dbl>  <dbl>            <dbl>
## 1     1      2             1000
## 2     2      4             3000
## 3     5     NA              500
```

In fact, the whole data.frame/tibble is a list

```
typeof(storms)
```

```
## [1] "list"
```

- This says that the tibble use `list` like storage-mode

```
class(storms)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

- This says `storms` inherits the properties of "tbl_df" "tbl" and "data.frame"
- This effects how the data interacts with functions
  - e.g. compare `print(mtcars)` (a "data.frame") and `print(storms)`

Why do you keep saying `tibble`?

tibble is the tidyverse "subclass" of base R's `data.frame`

- tibbles print nicer
- tibbles avoid niche issues/gotchas that you get with `data.frame`
  - see `?tbl_df` for the tidyverse-people's detailed spiel

Base R ways to pull out a column as a vector:

```r
# base R way
care_data$n_kids
```

```
## [1]  2  4  1  1 NA
```

```r
# base R way (same result as above)
care_data[["n_kids"]]
```

```
## [1]  2  4  1  1 NA
```

Two base R ways to pull out a column as a tibble/data.frame:

```
care_data["n_kids"]
```

```
## # A tibble: 5 x 1
##   n_kids
##    <dbl>
## 1      2
## 2      4
## 3      1
## 4      1
## 5     NA
```

```
# recall n_kids is the second column!
care_data[2]
care_data[c(FALSE, TRUE, FALSE, FALSE)]
```

Subsetting and extracting

Notice similarity with lists

- [[and $ for extracting (or pulling)

vs.

- [for subsetting / selecting.

Idea: a data frame is a named list with equal length vectors for each object (i.e. columns)

subsetting [] vs [,]

We saw that using `[]` pulls out columns. (“single index”)

Using `[ , ]` allows us to subset rows and columns. (“double index”)

`data`[filter rows , select columns]

Using [with two indices

data[filter rows ,]

```r
care_data[c(1, 3), ]
```

```
## # A tibble: 2 x 4
##      id n_kids child_care_costs random_noise
##   <int>  <dbl>            <dbl>        <dbl>
## 1     1      2             1000         36.1
## 2     3      1              300       -267.
```

Using [with two indices

`data[ , select columns ]` is equivalent to single index `data[get columns]`

```
# gets same results as single-bracket care_data[c("id", "n_
care_data[, c("id", "n_kids")]
```

```
## # A tibble: 5 x 2
##      id n_kids
##   <int>  <dbl>
## 1     1      2
## 2     2      4
## 3     3      1
## 4     4      1
## 5     5     NA
```

Using [with two indices

data[filter rows , select columns]

```
care_data[!is.na(care_data$n_kids), c("id", "n_kids")]
```

```
## # A tibble: 4 x 2
##      id n_kids
##   <int>  <dbl>
## 1     1      2
## 2     2      4
## 3     3      1
## 4     4      1
```

Recap

We discussed how to:

- Create vectors, lists and data frames for various circumstances
- Do vectorized operations and math with vectors
- Subset vectors and lists
- Understand data types and use type coercion when necessary

Next steps

Lab:

- *Today:* Vectorized math
- *Tomorrow:* Using `[` for data analysis

Touchstone: I can subset and extract from data and vectors with `[`

Next lecture:

- Using *dplyr* for data exploration!

(If you want to get ahead go through section 5.1-5.5 of `r4ds.had.co.nz`)

Appendix A: Subsetting with `[,]`

Using data[subset rows , *subset columns*]

We can refer to columns by name or index location.

```
care_data[1:2, c("n_kids", "child_care_costs")]
```

```
## # A tibble: 2 x 2
##   n_kids child_care_costs
##    <dbl>            <dbl>
## 1      2             1000
## 2      4             3000
```

Using data[subset rows , *subset columns*]

Or even a logical vector. (… this should remind you of vector subsetting!!)

```
care_data[1:2, c(TRUE, FALSE, TRUE, FALSE)]
```

```
## # A tibble: 2 x 2
##      id child_care_costs
##   <int>            <dbl>
## 1     1             1000
## 2     2             3000
```

Using data[*subset rows* , subset columns]

Similarly for rows!

```
care_data[c(1,3), c("id","n_kids")]
```

```
## # A tibble: 2 x 2
##        id n_kids
##     <int>  <dbl>
## 1       1      2
## 2       3      1
```

```
logical_indexing <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
care_data[logical_indexing , c("id","n_kids")]
```

```
## # A tibble: 2 x 2
##        id n_kids
##     <int>  <dbl>
## 1       1      2
## 2       3      1
```

Using data[*subset rows* , subset columns]

More usual usage for logical indexing

```r
logical_index <- care_data$id < 3
care_data[logical_index, c("id","n_kids")]
```

```
## # A tibble: 2 x 2
##      id n_kids
##   <int>  <dbl>
## 1     1      2
## 2     2      4
```

```r
# put the conditional right into the brackets.
care_data[care_data$id < 3 , "id"]
```

Let's get you to try.

First, we need data.

`us_rent_income` is a practice data set that comes tidyverse.

```
library(tidyverse)
head(us_rent_income)
```

```
## # A tibble: 6 x 5
##   GEOID NAME    variable estimate   moe
##   <chr> <chr>   <chr>       <dbl> <dbl>
## 1 01    Alabama income      24476   136
## 2 01    Alabama rent          747     3
## 3 02    Alaska  income      32940   508
## 4 02    Alaska  rent         1200    13
## 5 04    Arizona income      27517   148
## 6 04    Arizona rent          972     4
```

Testing out `data[` *subset rows* , subset columns]

Explore `us_rent_income` quickly with `glimpse()` and `head()`

How would you use a single bracket `[` ...

1. to select the state names and variable columns?
2. to get the rows 1, 3, 5, 7 ?
3. to get all the rows about "income".[4]
4. to get the `variable` and `estimate` columns for rows about Illinois?

[4] hint: test if *something* `== "income"`?

More examples [

```
# 1
us_rent_income[c("NAME", "variable")] # with single indexi
```

```
## # A tibble: 104 x 2
##    NAME       variable
##    <chr>      <chr>
##  1 Alabama    income
##  2 Alabama    rent
##  3 Alaska     income
##  4 Alaska     rent
##  5 Arizona    income
##  6 Arizona    rent
##  7 Arkansas   income
##  8 Arkansas   rent
##  9 California income
## 10 California rent
## # ... with 94 more rows
```