

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

Harris Coding Camp – Accelerated Track

Summer 2022

Today's class

Build foundational skills with

- ▶ Vectors
- ▶ Data types
- ▶ `data.frame` and `tibble`

Preamble: Data Analysis via Tidyverse and base R

Tidyverse has become the leading way people clean & manipulate data in R

- ▶ These packages make data analysis easier than core base R commands
- ▶ Tidyverse commands can be more efficient (less lines of code)

However, you will inevitably run into edge cases where tidyverse commands don't work the way you expect them to, or where you have to reuse/debug code written in base R ... and hence you'll have to be familiar with **base R**

It's good to have a basic foundation on both approaches and then decide which you prefer when conducting data analysis!

Vectors

Vectors

Vectors are the foundational data structure in R.

Here we will discuss how to:

- ▶ construct vectors, lists and data frames
- ▶ do vectorized math and computations
- ▶ deal with missing values (NA)
- ▶ work with vectors of different data types

Vectors

Vectors store an arbitrary¹ number of items of the *same* type.

`c()` is used to create a vector with explicitly given items.

```
# 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"
```

¹Within limits determined by hardware

Vectors

In R, nearly every data object you will work with is a vector

```
# vectors of length 1  
i_am_a_vector <- 1919  
as_am_i <- TRUE  
  
is.vector(i_am_a_vector)
```

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

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

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

```
# Some objects are not vectors e.g. functions  
is.vector(mean)
```

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

Vectors

Thus, the `c()` function combines vectors

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

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

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

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


Vectors

There are also several ways to create vectors of *sequential* numbers:

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

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

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

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

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

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

Vectors

There are some nice shortcuts for creating vectors:

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

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

Try out the following:

Can you explain what is going on?

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

Vectors

Notice how each works.

```
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"
```

... also notice that 5 is in quotes

Creating placeholder vectors of a given type

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

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

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

```
# or this one.
```

```
character(5)
```

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

Creating placeholder vectors of a given type

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

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

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

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

Creating random vectors

Create *random* data following a distribution (we will be using this many times in Stats 1)

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

```
## [1] 1.274732 2.635654 1.061860
```

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

```
## [1] 0.1019676 0.2414333 0.5942105 0.3893389
```

The pattern is `rdistribution` (`runif`, `rnorm`, `rf`, `rchisq`)

Adding elements to an existing 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"
```


Accessing Elements by Index

```
z[3]
```

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

```
z[2:4]
```

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

Accessing Elements by Index

We can reassign accessed values too.

```
z[1] <- "Arthur"  
z[c(1,3)]
```

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

Accessing Elements by Index

Using a negative sign, allows subsetting 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:

```
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
```

Examining Vectors

```
# how many elements does a vector have?  
length(a)
```

```
## [1] 10
```

```
# what are the summary stats?  
summary(a)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   
##      0.00    2.25    4.50    4.50    6.75    9.00
```

Vectorized code

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

- ▶ In non-vectorized languages, you use a loop.

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
```


Doing math with vectors

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
```

Doing math with vectors

Many built-in functions naturally extend to vectors.

- ▶ 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 2.449490
```

Try it yourself

Guess the output before running the codes, and verify the results.

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

```
a_vector <- rnorm(200)  
sqrt(a_vector)  
round(a_vector, 2)
```

Warning: Vector recycling

Be careful when operating with vectors. What's happening here?

```
a <- 1:6 + 1:5
```

```
a
```

Warning: Vector recycling

If vectors are different lengths, the shorter vector re-starts from it's beginning.

```
# 1 + 1,  
# 2 + 2,  
# 3 + 3,  
# 4 + 4,  
# 5 + 5,  
# 6 + 1 -- '1' is 'recycled'  
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): longer object len  
## multiple of shorter object length
```

```
## [1]  2  4  6  8 10  7
```

Warning: Vector recycling

```
# some times we get a warning ...
```

```
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): longer object len
```

```
## multiple of shorter object length
```

```
## [1]  2  4  6  8 10  7
```

```
# ... but not if vector lengths are multiples
```

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

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

Binary operators are vectorized

We can do conditional expressions **element by element!**

```
# 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
```

Binary operators are vectorized

We can do boolean logic with vectors **element by element!**

```
my_numbers <- 1:6  
# Compare my_numbers with c(4, 4, 4, 4, 4, 4)  
my_numbers > 4
```

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

```
my_numbers == 3
```

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


Accessing Element by Logical Vector

```
logical_index <- 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]
```

Accessing Element by Condition (i.e. Logical Vector)

```
x <- c(1, 2, 3, 11, 12, 13)
# Choose elements which meet the condition
# c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)
logical_index <- x < 10
x[x < 10]
```

```
## [1] 1 2 3
```

Accessing Element by Condition (i.e. Logical Vector)

```
# Replace elements which meet the condition with 0  
x[x < 10] <- 0  
x
```

```
## [1]  0  0  0 11 12 13
```

Functions that reduce vectors

Some functions take a vector and return a summary

- ▶ These are used with `summarize()`

```
a_vector <- c(1, 3, 5, 7, 9, 11, 13)
sum(a_vector)      # add all numbers
median(a_vector)   # find the median
length(a_vector)   # how long is the vector
any(a_vector > 1)  # TRUE if any number in a_vector > 1
```

- ▶ `paste0()` is a function that combines character vectors

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

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

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?

```
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
```

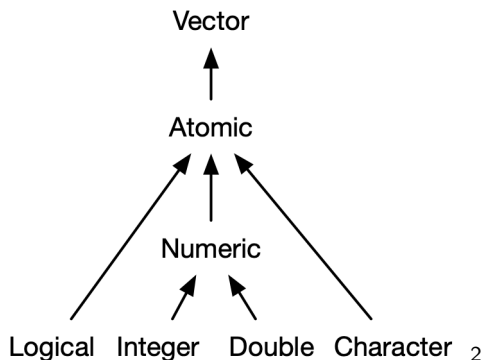
R does not have logic for multiplying character vectors!

Wait what's a character vector?

Data types

R has four primary types of atomic vectors

- ▶ these determine how R stores the data (technical)



²Image from <https://adv-r.hadley.nz/vectors-chap.html>

Data types

Focusing on the primary types, we have:

```
# 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

```
x <- "1"  
typeof(x) # similar result as mode(x) and class(x)
```

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

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

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

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

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

technical note: `typeof()` and `mode()` are basically synonyms returning types builtin to R. When programs develop new structures they can assign new `class()`, so `class` allows for more nuanced results.

Type coercion

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

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

We can reassign types on the fly:

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

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

What Happens When You Mix Types Inside a Vector?

```
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

```
TRUE + 10
```

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

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

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

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

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

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
```

NAs are contagious

We can tell R to ignore the missing values in some functions:

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

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

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

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

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

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

Testing for NA requires a special function

```
x <- 4  
x == NA
```

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

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

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

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

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

Vectors require a single type

```
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?

What do we do 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"
```

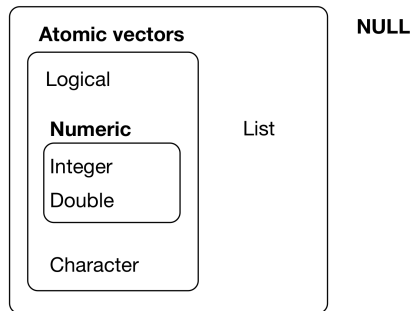
Lists are vectors?

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

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

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

Vectors



3

³image 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

Data Frame vs Vector vs List

The diagram shows a table with 10 rows and 5 columns. A red rectangle outlines the entire table, labeled 'Data Frame'. A green rectangle highlights the 'year' column, labeled 'Vector'. A blue rectangle highlights the third row, labeled '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

Data Frame vs Vector vs List

We like tidy data where:

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

The diagram illustrates the relationship between a Data Frame, a Vector, and a List using a table of Australian data. A red rectangle outlines the entire table, labeled "Data Frame". A green rectangle highlights the "year" column, labeled "Vector". A blue rectangle highlights the third row (Australia, 1953, 360, 2.5), labeled "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

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         187.
## 2     2     4     3000         102.
## 3     3     1      300        -92.7
## 4     4     1      300         191.
## 5     5    NA      500          9.91
```

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
```

Extracting

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

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

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

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

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

Subsetting

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
```

```
care_data[2] # recall n_kids is the second column!
```

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[get rows , get columns]`

Using [with two indices

```
data[ get rows , get columns ]
```

```
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           187.
## 2     3     1           300          -92.7
```

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

```
## # A tibble: 5 x 2
##       id child_care_costs
##   <int>          <dbl>
## 1     1          1000
## 2     2          3000
## 3     3           300
## 4     4           300
```

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

```
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
```

```
# 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?

⁴hint: test if *something* == "income"?

More examples [

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`)