08 - R Basics I

ST 597 | Spring 2017
University of Alabama

08-rbasics.pdf

Assigning Variable

Data Types

Vectors

Creating Vectors

Vector Indexing

Required Packages

```r
library(tidyverse)
library(nycflights13)
library(stringr)        # stringr is part of tidyverse,
                        # but not auto loaded
```

R Manuals

If you want to know (almost) everything about the R language, check out the manuals `http://cran.r-project.org/manuals.html`.

Especially these two: - An Introduction to R

- The R language definition

Another good reference is Hadley Wickham's R Vocabulary

Assigning Variable

Setting Variables

Use the assignment operator (<- or =) to save an object. Here we will create the data frame object named `x`

```
x = tibble(a=1:5, b=a+5)     # remember tibble() requires dplyr
```

Notice that the object named `x` shows up in the environment.

If we want to see it, we can `View(x)` or just type `x` in console.

```
x
#> # A tibble: 5 × 2
#>         a     b
#>     <int> <dbl>
#> 1       1     6
#> 2       2     7
#> 3       3     8
#> 4       4     9
#> 5       5    10
```

Setting variables

And now we can work with `x`

```
mutate(x, c=2*b)
#> # A tibble: 5 × 3
#>       a     b     c
#>   <int> <dbl> <dbl>
#> 1     1     6    12
#> 2     2     7    14
#> 3     3     8    16
#> 4     4     9    18
#> 5     5    10    20
```

- Notice that we didn’t create a new variable, we just printed results to screen

- Can we now use `filter()` to select all rows with `c > 15`?

Variable naming

Variable names can contain any alphanumeric characters along with periods (.) and underscores (_), but they cannot start with a number or underscore.

Some examples:

```
var_1 <- 10+1
var.2 <- 22
Var.2 <- 222
my_var.2_3 = 23
.x = 5
```

Be careful, R is **case sensitive**:

```
Var_1
#> Error in eval(expr, envir, enclos): object 'Var_1' not found
```

Oops, did I mean lowercase `var_1`?

```
var_1
#> [1] 11
```

Variable naming

You cannot name objects with **reserved** words like:

- `TRUE`, `FALSE`, `NA`, `if`, `next`, `function`

See the help:

```
?reserved
```

for a full list of reserved words.

Variable re-assignment

If you don't like a variable, you can write over it

```
x = tibble(a=1:5, b=a+5)    # make initial x
x = mutate(x, c=2*b)            # create new modified x
x = 0                           # now set x = 0
```

If you really want to get rid of it, use `rm()`

```
rm(x)                          # Auf Wiedersehen!
```

Data Types

Data Types

There are five types of data that we will encounter today:

1. numeric (or double)
2. integer
3. character (string)
4. logical (TRUE/FALSE)
5. Date

The type of data contained in a variable can be obtained with the function `class()`

```
x = 0
class(x)            # numeric
class(0L)           # integer
class("zero")       # character
class(0 == "zero") # logical
```

Numeric Data

Numeric Data is represented as either

- numeric (double or floating point)
- integer

When you type a number into R, it will assign it as a double. If you really need an integer (which is not often), you can append the number with the capital letter `L`.

```r
x = 5
class(x)
#> [1] "numeric"
y = 5L
class(y)
#> [1] "integer"
```

Numeric Data Operations

- When you do math operations, R will automatically convert integers to numeric when needed

```
x = 2L       # integer 2
x + 1.25     # integer + double = double
#> [1] 3.25
```

- And numeric variables will be converted to integers for indexing

```
x = c(1.1, 2.2, 3.3)      # this is a vector of doubles
x[3]                      # return 3rd element. as x[3L]
#> [1] 3.3
```

Internally, R converts the 3.000 to an integer `x[3L]`.

What do you think `x[2.9]` will return?

Character Data

Instead of using numbers, we can use character data (or strings).

To create a character variable, enclose it in quotes (single or double):

```
x = "French Toast"
y = 'Bacon and Eggs'
```

Some useful functions for character manipulation are found in the `stringr` package

```
library(stringr)            # install.packages("tidyverse")
```

While the `stringr` package is installed with `tidyverse`, it needs to be loaded explicitly with `library(stringr)`

Your Turn #1 : Character Data Manipulation

Ensure you have the following objects in your environment

```
x = "French Toast"
y = 'Bacon and Eggs'
```

1. Load the `stringr` R package. `?stringr`
2. Use the function `str_length()` to find how many characters are in `x` and `y`. (Remember to type `?str_length` for help.)
3. Use `str_to_lower()` to convert everything to lowercase.
4. Remove the “and” from `y` using `str_replace()`

Logicals

Logical data are either `TRUE` or `FALSE`.

You can get a logical by using the reserved words:

```
a = TRUE
b = FALSE
```

Or by comparing two things (<, <=, >, >=, ==, ! =):

```
2 == 3    # Does 2 equal 3?
2 != 3    # Does 2 not equal 3?
2 < 3     # Is 2 less than 3?
2 <= 3    # Is 2 less than or equal to 3
"case" == "Case"  # Is R case insensitive?
3+2 > 4   # logical operators have lower precedence than arithmetic
```

Logical Math

If you use math with a logical, it will convert it to a numeric (TRUE=1, FALSE=0)

```
win = .5 > runif(1)   # Did you win?
prize = win * 3       # If you won, your prize is $3, else its $0.
prize
#> [1] 3
```

Dates

R recognizes calendar dates. Dates are created with the `as.Date()` function. The default format is ISO 8601 standard of: *year-month-day*

```
date1 = as.Date("2017-02-20")
date1
#> [1] "2017-02-20"
class(1)
#> [1] "numeric"
```

But we can accept dates in other formats using the `format=` argument

```
date2 = as.Date("2/20/17", format="%m/%d/%y")
date1 == date2
#> [1] TRUE
```

Date Format

There is also a `format()` function that will extract elements of the date.

```r
format(date1, "%d:%m:%Y")
#> [1] "20:02:2017"
```

The codes `%d`, `%m`, and `%y` extracts the day, month, and year.

The full list of date (and time) codes found in `?strptime`,

Date Format: Codes

code	component	value
`%y`	year	year without century (00-99, e.g., 17)
`%Y`	year	year with century (e.g., 2017)
`%b`	month	month name (abbreviated)
`%B`	month	month name (full)
`%m`	month	month number (01-12)
`%d`	day	day of month number (01-31) double digits
`%e`	day	day of month number (1-31) single digits
`%j`	day	day of year (001-336)
`%a`	day	day of week name (abbreviated)
`%A`	day	day of week name (full)
`%u`	day	day of week number (1-7, Monday is 1)
`%w`	day	day of week number (0-6, Sunday is 0)
`%U`	week	week of year (00-53) using Sunday as first day 1 of the
`%V`	week	week of year (01-53) using ISO 8601 standard
`%W`	week	week of year (00-53) using Monday as first day of weel

Date Format: Example

```r
(today = as.Date("Feb 20, 17", format="%b %d, %y"))
#> [1] "2017-02-20"

format(today, "%m/%d/%Y")       # usualy US date convention
#> [1] "02/20/2017"
format(today, "%a=%u, %m=%B")   # extracting day and month
#> [1] "Mon=1, 02=February"
```

Date Math

Behind the scenes, R treats dates as *the number of days since Jan 1, 1970*, but prints dates out as characters.

```r
date1 = as.Date("2017-02-20")
date2 = as.Date("1970-01-01")

date1 + 1
#> [1] "2017-02-21"
date1 + 365
#> [1] "2018-02-20"
date1 - date2
#> Time difference of 17217 days
as.numeric(date1)
#> [1] 17217
as.numeric(date2)
#> [1] 0
```

Conversion between data types

We can convert between data types with `as.<format>`.

```
(x = tibble(a=1:3, b=c('001', '02', '3'), c=c(1.2, 2.6, 3.99999)))
mutate(x, a=as.character(a), b=as.integer(b), c=as.integer(c))
```

Vectors

Vectors

A vector is a collection of elements (or contiguous cells of data).

In R, vectors must **all** be of the same type (e.g., all numeric, integers, logicals).

We have actually been using vectors that consists of a single element, e.g.

```
x <- 1L          # remember 1L is the integer 1
y <- 1/3
z <- "Pancakes"  # I must be thinking about breakfast
```

Recall that each column of a data frame/tibble is a vector.

Column Vectors from data frames

A column vector can be extracted from a data frame with `$` or `[[ ]]`

```r
data(flights)
dd = flights$dep_delay          # extract column with $
ad = flights[["arr_delay"]]     # extract column with [[ ]]
```

Vector Creation with c()

Vectors can be created with function `c()` for combine:

```
v1 = c(1,2,3)
v1
#> [1] 1 2 3
v2 = c("Porter", "ST597", "Statistics")
v2
#> [1] "Porter"     "ST597"      "Statistics"
```

The function `length()` gives the number of elements in the vector

```
length(v2)        # the number of elements that are in the vector
#> [1] 3
```

Vector Coersion

Warning: R will automatically (and without warning) change the class of a vector so all objects are of the same type.

The coercion rules place: `character` > `numeric` > `integers` > `logical`

```
c(1,2,"three")   # all become characters
#> [1] "1"     "2"     "three"
c(1,2,FALSE)     # the logical becomes numeric
#> [1] 1 2 0
c(1L,2L,3)       # integers go to doubles
#> [1] 1 2 3
```

Vector Operations

- vectors are at the heart of R (vectorized language)
- many R operators are applied to each element of vector automatically (without loops)

```
x = c(1,2,3,4,5)
x + 10
#> [1] 11 12 13 14 15
x^2
#> [1]  1  4  9 16 25
```

```
v2 = c("Mike", "ST597", "Statistics     ")
str_length(v2)     # number of characters in each element (watch white
#> [1]  4  5 15
```

Multiple Vector Operations

Consider two vectors:

```
x = c(1,2,3,4,5)
y = c(5,4,3,2,1)
```

```
x + y
x - y
x^y
```

```
x >= y
#> [1] FALSE FALSE  TRUE  TRUE  TRUE
```

The `dplyr` functions `mutate()` performs vector operations. And the `filter()` function vector comparisons.

```
mutate(flights, gain = arr_delay - dep_delay)
filter(flights, arr_delay <= 0 )
```

Vector Recycling

R has several *interesting* features. One is **vector recycling.**

```
x = c(1,2,3,4,5,6)    # length(x) = 6
y = c(1,2)            # length(y) = 2
```

What do you think should happen if you add `x + y`?

```
z = x + y             # Really R, no warning message?
z
#> [1] 2 4 4 6 6 8
length(z)
#> [1] 6
```

Can you figure out what happened?

Vector Recycling Details

Shorter vectors (e.g., `y`) is recycled (expanded) so its length is same as `x`.

So `x + y` is actually

```
c(1,2,3,4,5,6) + c(1,2,1,2,1,2)
#> [1] 2 4 4 6 6 8
```

R assumes you know what you are doing if the length of the longer vector is a multiple of the length of the shorter one. If not, you will at least get a *warning message*:

```
a = c(1,2,3,4,5); y = c(1,2)  # Note: semi-colon acts like a new line
a/y
#> Warning in a/y: longer object length is not a multiple of shorter o
#> length
#> [1] 1 1 3 2 5
```

More Vector Recycling

Recycling is nice (and seems more appropriate) when the shorter vector is a single element:

```
x = c(1,2,3,4,5,6)
x < 3
#> [1]  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Notice how this returns a logical vector of the same length as `x`. This type of behavior is handy-dandy.

Remember our subsetting:

```
# Create a new data frame that contains only the flights
#   that were less than 1000 miles (distance)
library(nycflights13)
data(flights)
filter(flights, distance < 1000)
#> # A tibble: 189,671 × 19
#>      year month   day dep_time sched_dep_time dep_delay arr_time
#>     <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1   2013     1     1      554            600        -6      812
#> 2   2013     1     1      554            558        -4      740
#> 3   2013     1     1      557            600        -3      709
#> 4   2013     1     1      557            600        -3      838
```

Creating Vectors

Creating Vectors: `c()`

We have already introduced `c()`

```
x = c(1,2,3,4,5)
x
#> [1] 1 2 3 4 5
```

`c()` can combine multiple vectors

```
c(0,x,6)
#> [1] 0 1 2 3 4 5 6
```

Creating Vectors: colon

The colon (:) operator, `a:b` creates *integers* from `a` to `b`

```
1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
-5:5
#>  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
10:1      # blast-off; it recognizes direction!
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

Creating Vectors: colon

Colons take precedence over arithmetic:

```
2+1:5
#> [1] 3 4 5 6 7
(2+1):5
#> [1] 3 4 5
```

Creating Vectors: `seq()`

A more general version of `:` is the function `seq()`.

```
seq(10, 20, by=2)
#> [1] 10 12 14 16 18 20
seq(5.5, 10.2, length=10)
#>  [1]  5.500000  6.022222  6.544444  7.066667  7.588889  8.111111  8
#>  [8]  9.155556  9.677778 10.200000
seq(100, 0, by=-25)        # notice that 'by' accepts negatives
#> [1] 100  75  50  25   0
```

This also works with Dates:

```
today = as.Date('2017-2-20')
seq.Date(today, today+60, by="months")
#> [1] "2017-02-20" "2017-03-20" "2017-04-20"
seq.Date(today, today+60, by="30 days")
#> [1] "2017-02-20" "2017-03-22" "2017-04-21"
```

Creating Vectors: `rep()`

```
x = 1:5
rep(x, times=3)
#>  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
c(x,x,x)
#>  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
rep(x, each=3)
#>  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Creating Character Vectors: paste()

The paste() function combines vectors after converting to characters.

```r
paste("Stats", "is", "fun")        # length of 1
#> [1] "Stats is fun"
c("Stats", "is", "fun")            # length of 3
#> [1] "Stats" "is"    "fun"
```

Create vectors of values X1, X2, . . . , X5:

```r
paste("X", 1:5, sep="")
#> [1] "X1" "X2" "X3" "X4" "X5"
```

Your Turn #2 : Creating Vectors

1. Find a way to create the vector with elements `1,0,3,3,3,7,6,5,4,3,2,1,2.7`
2. Without running it, can you determine what `y` is:

```r
x = 1:4
y = c(x, seq(10,4,by=-2), rep(x,each=2), TRUE, FALSE)
```

3.Without running it, can you determine what `z` is:

```r
n = 5
z = 1:n-1
```

Vector Indexing

Vector Indexing

Extract or change value of certain elements of a vector.

http://cran.r-project.org/doc/manuals/r-release/R-intro.html#Index-vectors

Vector Indexing: by position

Elements of a vector can be extracted with square brackets, `[]`.

```
x = 10:1
x[1]
#> [1] 10
x[1:3]
#> [1] 10  9  8
x[c(1,3,5,7,9)]
#> [1] 10  8  6  4  2
x[seq(1,10,by=2)]
#> [1] 10  8  6  4  2
x[c(1,1,1,2,2,2)]  # this asks for the same value multiple times
#> [1] 10 10 10  9  9  9
```

Vector Indexing: by exclusion

If you index with with negative numbers, it returns everything except those indices.

```
x = 1:10
x[-1]
#> [1]  2  3  4  5  6  7  8  9 10
x[-(1:5)]
#> [1]  6  7  8  9 10
```

Use `head(x,3)` and `tail(x,2)` to get the first 3 and last 2 elements of `x`

Vector Indexing: by logical vector

You can index with a logical vector. It will return every index that is `TRUE`

```
x = 1:5
ind = c(TRUE,TRUE,FALSE,FALSE,TRUE)
x[ind]
#> [1] 1 2 5
```

What will be returned?

```
ind2 = c(TRUE,FALSE)
x[ind2]
```

Vector Indexing: Assignment

Elements can be assigned different values with indexing:

```
x = 1:5
x[3] <- 99  # or x[3] = 99
x
#> [1]  1  2 99  4  5
```

Here's something you probably don't expect:

```
x[10]  <- 10
x
#>  [1]  1  2 99  4  5 NA NA NA NA 10
```

R will (without warning) extend the vector to the appropriate length, filling in with `NA`.

Your Turn #3 : Vector Indexing

```r
set.seed(01312016)    # set random seed
x = runif(100)        # 100 uniform random numbers [0,1]
```

1. Find the index for the elements of `x` greater than the median
2. Extract all elements of `x` greater than the median
3. Extract all elements of `x` that are in the lower and upper 5%. Hint: `quantile()`.

The dplyr `filter()` only works on data frames. Here, I am looking for the solutions without using data frames. But you could always create a data frame: `df = tibble(x)` to use `dplyr` functions.

Change vector elements by condition

The `ifelse()` function is very handy. It takes a logical vector `test`, and for each element *i* returns `yes[i]` if `test[i]=TRUE` and `no[i]` if `test[i]=FALSE`

```
if(test, yes, no)
```

Tax Rates. Suppose a simple graduated tax rate of 20% on income less than $100K/year and 30% for the portion of incomes that exceeds $100K

```
income = c(20, 210, 99, 387, 101)  # income in thousands
tax =  ifelse(income < 100, income*0.20, 100*0.20 + (income-100)*0.30)
tax/income                          # overall tax rate
#> [1] 0.2000000 0.2523810 0.2000000 0.2741602 0.2009901
```

Notice how *vector recycling* is used to make “rich” and “poor” into vectors

```
class =  ifelse(income < 100, "poor", "rich")
class
#> [1] "poor" "rich" "poor" "rich" "rich"
```

Modify columns in a data frame

The dplyr `mutuate()` function can be used to modify columns from a data frame.

Suppose we were just told that our `flights` data had an error. All flights that arrived more than 1 hour early have an `air_time` value that is 20 minutes too large.

```r
corrected = mutate(flights,
                   air_time = ifelse(arr_delay < -60,
                                     air_time-20,      # if TRUE
                                     air_time)   )     # if FALSE

# note: watch spacing;  arr_delay<-60 vs. arr_delay < - 60
```

Double check we did it correctly:

```r
filter(flights, arr_delay < -60)$air_time
filter(corrected, arr_delay < -60)$air_time
```