Week 2: R Language Essentials

MS 282

September 04, 2017

Overview

- Functions & data
- `R` data structures
- `R` operators
- `R` functions
- The `R` workspace
- Built in data

Quick Review

Assignment

- Values in `R` are assigned to **variables**
  - "<-" is known as the *assignment operator*
  - The *assignment operator* assigns values to variables

```
x <- 3
```

Today we will learn about different types of these variables

Functions & Data

It's all about functions and data

2 sorts of things (**objects**): **data** and **functions**

- **Data**: things like 7, "seven", 7.000, the matrix $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$
- **Functions**: things like $\log$, $+$ (two arguments), $<$ (two), $\bmod$ (two), `mean` (one)

The Applied Statistics Pattern

Applied statisticians write `functions()` that take `data` and transform, manipulate, or analyze it in some way to produce output or results. A majority of the time is spent wrangling data.

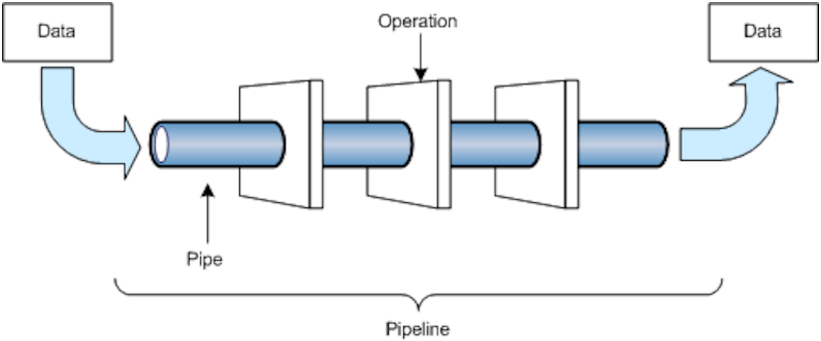


Figure 1: pipeline

First, some data

R Data Types

Objects in *R* can be a number of different types. These are types you are most likely to encounter:

1. **Character**: character objects are letters, words, or strings.
   - Ex: `'j'`, `'hello'`, `'treatment A'`
2. **Numeric**: numeric objects are integers or real numbers
   - Ex: `1`, `550`, `3.14`
3. **Logical**: logical objects take the value of `TRUE` and `FALSE` and are often used to control programming flow
4. **Missing or ill-defined values**: `NA`, `NaN`, etc.

Arithmetic Operators

```r
7 + 5
```

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

```r
7 - 5
```

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

```r
7 * 5
```

```
## [1] 35
```

```r
7 ^ 5
```

```
## [1] 16807
```

Arithmetic Operators (cont.)

```
7/5
```

```
## [1] 1.4
```

```
7 %% 5 # modulo division
```

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

```
7 %/% 5 # integer division
```

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

Comparison Operators

```r
7 > 5
```

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

```r
7 < 5
```

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

```r
7 >= 7
```

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

```r
7 <= 5
```

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

Comparison Operators (cont.)

```r
'a' < 'b'
```

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

```r
'a' > 'b'
```

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

```r
'A' > 'a'
```

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

```r
'A' < 'a'
```

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

Equality Operators

```r
7 == 7
```

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

```r
7 == 5
```

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

```r
'one' != 1
```

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

```r
'one' == 'one'
```

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

Boolean operators

Basically "and" and "or":

```r
(5 > 7) & (6*7 == 42)
```

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

```r
(5 > 7) | (6*7 == 42)
```

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

Remember De Morgan's Laws from discrete math.

Checking Types

- `typeof()` function returns the type
- more on the topic here and here
- `is.`*foo*`()` functions return Booleans for whether the argument is of type *foo*

```r
typeof(7)
```

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

```r
is.numeric(7)
```

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

```r
is.na(7)
```

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

```r
is.na(NA)
```

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

```r
is.character(7)
```

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

```r
is.character("7")
```

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

```r
is.character("seven")
```

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

```r
is.na("seven")
```

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

Data Structures

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

(Atomic) Vectors

Vectors

Vectors are *R*'s most basic data structure. When we created the variable `x` in the review section, we had actually created a *vector* of length 1. The elements contained in a vector must be of the same *type* (see prev. section). Vectors including more than one element are frequently constructed using the `c()` (concatenate) function:

```
a <- c(1, 2, 5.3, 6, -2, 4) # numeric vector
b <- c("one", "two", "three") # character vector
c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE) # logical vec
is(a)
```

```
## [1] "numeric" "vector"
```

```
is.vector(a)
```

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

Accessing values in vectors

- `[` is used to select elements of a vector
- `x[1]` is the first element, `x[4]` is the 4th element
  - note that `R` is one-based unlike many languages
- `x[-4]` is a vector containing all but the fourth element

```
x <- c(7, 8, 10, 45)
x[1]
```

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

```
x[-4]
```

```
## [1]  7  8 10
```

Accessing values in vectors

You can also supply a vector of indices to index with:

```r
x[c(2,4)]
```

```
## [1]  8 45
```

This works with negative values too:

```r
x[c(-1,-3)]
```

```
## [1]  8 45
```

Accessing values in vectors

Boolean vectors can be used for selection as well:

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

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

```
x > 9
```

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

```
x[x > 9]
```

```
## [1] 10 45
```

Accessing values in vectors

Often chaining together complex boolean rules is useful

```r
x[(x > 7) & (x < 45)]
```

```
## [1]  8 10
```

Vector arithmetic

Operators apply to vectors “pairwise” or “elementwise”:

```
y <- c(-7, -8, -10, -45)
x + y
```

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

```
x * y
```

```
## [1]   -49   -64  -100 -2025
```

Recycling

Vector operations in `R` *recycle* values.

Recycling repeats elements in shorter vector when combined with longer

```
x + c(-7,-8)
```

```
## [1]  0  0  3 37
```

```
x^c(1,0,-1,0.5)
```

```
## [1] 7.000000 1.000000 0.100000 6.708204
```

Recycling

Single numbers are vectors of length 1 for purposes of recycling (scalar multiplication):

```
2*x
```

```
## [1] 14 16 20 90
```

Checking Equality

To compare whole vectors, best to use `identical()` or `all.equal()`:

```
x == -y
```

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

```
identical(x,-y)
```

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

```
all.equal(x,-y)
```

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

Named components

You can give names to elements or components of vectors; you can subset by these

```r
names(x) <- c("v1", "v2", "v3", "fred")
x
```

```
##   v1   v2   v3 fred
##    7    8   10   45
```

```r
names(x)
```

```
## [1] "v1"   "v2"   "v3"   "fred"
```

```r
x[c("fred","v1")]
```

```
## fred   v1
##   45    7
```

Coercion

- All vector elements must be same type
- Attempting to combine different types will *coerce* all values to the most flexible type
- Flexibility order from (least to most): logical, integer, double, and character.
- For example, combining a character and an integer yields a character:

```
c("a", 1)
```

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

```
c(TRUE, 0)
```

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

Side Note: Factors

What is a Factor

- `R`'s way of representing categorical/nominal variables
- Useful if you want to restrict values of a vector (i.e. sex, study group)
- We will avoid them for the first part of the course
- But will illustrate their usefulness later

What is a Factor

Say you have a study with three groups:

```
group <- c(1, 1, 1, 2, 2, 2, 3, 3, 3)
```

This probably doesn't make sense:

```
group + 2
```

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

What is a Factor

Solution: `factor()`

```r
group <- as.factor(group)
group
```

```
## [1] 1 1 1 2 2 2 3 3 3
## Levels: 1 2 3
```

```r
is(group)
```

```
## [1] "factor"                 "integer"                "oldClas
## [4] "numeric"                "vector"                 "data.fr
```

What is a Factor

Now values in vector can only be 1, 2, or 3 (and addition doesn't work)

```
group[1] <- 4
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = 4): invalid
## generated
```

```
group
```

```
## [1] <NA> 1    1    2    2    2    3    3    3
## Levels: 1 2 3
```

The allowed values are known as the `levels`

What is a Factor

Even though `group` `is()` numeric, addition is no longer allowed

```
group + 2
```

```
## Warning in Ops.factor(group, 2): '+' not meaningful for
```

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

Lists

Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using list() instead of c():

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
```

```
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.3 5.9
```

Lists

```r
nested_lists <- list(list('hello', 'goodbye'), "a", list('t
str(nested_lists)
```

```
## List of 3
##  $ :List of 2
##   ..$ : chr "hello"
##   ..$ : chr "goodbye"
##  $ : chr "a"
##  $ :List of 2
##   ..$ : chr "test"
##   ..$ : chr "test2"
```

Lists Can Have Names Too

```r
named_list <- list(element_one = 1:3, two = c(TRUE, FALSE,
                   bike = c('wheel', 'brakes'))
str(named_list)
```

```
## List of 3
##  $ element_one: int [1:3] 1 2 3
##  $ two        : logi [1:3] TRUE FALSE TRUE
##  $ bike       : chr [1:2] "wheel" "brakes"
```

```r
names(named_list)
```

```
## [1] "element_one" "two"         "bike"
```

Accessing Values in Lists

Can access by index using [[

```
named_list[[2]]
```

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

```
named_list[[3]]
```

```
## [1] "wheel"  "brakes"
```

Accessing Values in Lists

or by name

```
named_list[['two']]
```

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

```
named_list[['bike']]
```

```
## [1] "wheel"  "brakes"
```

Combining Lists

- use the `c()` function to combine multiple lists
- `c(list, list)` returns another list

```
combined_list <- c(x, named_list)
str(combined_list)
```

```
## List of 7
##  $            : int [1:3] 1 2 3
##  $            : chr "a"
##  $            : logi [1:3] TRUE FALSE TRUE
##  $            : num [1:2] 2.3 5.9
##  $ element_one: int [1:3] 1 2 3
##  $ two        : logi [1:3] TRUE FALSE TRUE
##  $ bike       : chr [1:2] "wheel" "brakes"
```

The $ Shortcut

The $ symbol can be used as an extraction operator (like `[[`) on lists. The basic usage is as follows

```
named_list[['bike']]
```

```
## [1] "wheel"  "brakes"
```

```
named_list$bike
```

```
## [1] "wheel"  "brakes"
```

```
identical(named_list[['bike']], named_list$bike)
```

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

The $ Shortcut

I suggest **not** to using the $ shortcut as it can introduce bugs in your code if list names have spaces in them.

```
test <- list(1, c(3, 4))
names(test) <- c('element_one', 'element two')
test[['element two']]
```

```
## [1] 3 4
```

```
test$element two
```

```
Error: unexpected symbol in "test$element two"
```

Adding Elements to a List

This can be done using the concatenate (`c()`) operator:

```
named_list <- c(named_list,'dog')
str(named_list)
```

```
## List of 4
##  $ element_one: int [1:3] 1 2 3
##  $ two        : logi [1:3] TRUE FALSE TRUE
##  $ bike       : chr [1:2] "wheel" "brakes"
##  $            : chr "dog"
```

Adding Elements to a List

Or by indexing (by any method) a slot (possibly new) and writing data to it

```
named_list$new_slot <- c(1,2,9)
str(named_list)
```

```
## List of 5
##  $ element_one: int [1:3] 1 2 3
##  $ two        : logi [1:3] TRUE FALSE TRUE
##  $ bike       : chr [1:2] "wheel" "brakes"
##  $            : chr "dog"
##  $ new_slot   : num [1:3] 1 2 9
```

Removing Elements from a List

To remove an element, index the element you would like to remove and set it to NULL

```
named_list[[4]] <- NULL
str(named_list)
```

```
## List of 4
##  $ element_one: int [1:3] 1 2 3
##  $ two        : logi [1:3] TRUE FALSE TRUE
##  $ bike       : chr [1:2] "wheel" "brakes"
##  $ new_slot   : num [1:3] 1 2 9
```

Matrices & Arrays

Matrices & Arrays

- Matrices & arrays are just vectors that have **dimensions**
- Since they are vectors, they can only 1 one data type
  - same coercion rules apply
- Dimension is controlled by the `dim()` attribute of a vector
- A matrix is a special case of an array with exactly two dimensions
- Matrices are used commonly in statistics; you will see them often
- Arrays are much rarer, but worth being aware of

Creating Matrices & Arrays

Matrices are created with `matrix()`

```r
a <- matrix(1:6, ncol = 3, nrow = 2)
a
```

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

```r
is(a)
```

```
## [1] "matrix"    "array"     "structure" "vector"
```

Creating Matrices & Arrays

Arrays are created with `array()`; vector

```
b <- array(1:12, c(2, 3, 2))
b
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Creating Matrices & Arrays

They can also be created using the assignment form of dim():

Adding 2 dimensions gives you a matrix:

```r
# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
```

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

Creating Matrices & Arrays

Adding 3 or more dimensions gives you an array:

```
d <- 1:12
dim(d) <- c(2, 3, 2)
d
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Accessing Values in Matrices & Arrays

- Use `[` again to access values
- Need to provide enough indices for each dimension
- For matrices, subset as `a[row, column]`

```
a
```

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

```
a[1, 3]
```

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

Accessing Values in Matrices & Arrays

If you omit an index, all values from that dimension are selected

```
a[1,]
```

```
## [1] 1 3 5
```

```
a[,3]
```

```
## [1] 5 6
```

Data Frames

Data Frames

- The data frame is the most common way to store data in `R`
- Data frames look like matrices; but columns can have different types
- A data frame is actually a list of equal-length vectors
- Can be created manually with `data.frame`
- Created by default by data import functions

Creating Data Frames

```r
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
df
```

```
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

```r
is(df)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

Creating Data Frames

R will throw an error if the vectors have unequal length

```r
df <- data.frame(x = 1:4, y = c("a", "b", "c"))
```

```
Error in data.frame(x = 1:4, y = c("a", "b", "c")) :
  arguments imply differing number of rows: 4, 3
```

Creating Data frames

You can also create data frames from vectors already in your environment

```
a <- 1:3
b <- letters[1:3]
my_df <- data.frame(a, b)
my_df
```

```
##   a b
## 1 1 a
## 2 2 b
## 3 3 c
```

Creating Data frames

Beware `data.frame()`'s default behavior which turns strings into factors. Use stringAsFactors = FALSE to suppress this behavior:

```r
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

```r
df <- data.frame(x = 1:3, y = c("a", "b", "c"),
                 stringsAsFactors = FALSE)
str(df)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: chr  "a" "b" "c"
```

Subsetting a Data Frame

Data frames can be subset using numeric indices's just like matrices

```
df[1,2]
```

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

```
df[3,]
```

```
##   x y
## 3 3 c
```

```
is(df[3,])
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

Subsetting a Data Frame

You can also use names and logical indexes:

```
df[ ,'x']
```

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

```
is(df[ ,'x'])
```

```
## [1] "integer"                "numeric"                "vector'
## [4] "data.frameRowLabels"
```

```
df[df[,'x'] < 3, ]
```

```
##   x y
## 1 1 a
## 2 2 b
```

The R Workspace

Defining & Viewing the Workspace

The *R* workspace can be thought of as a container holding all of the objects you've created during your *R* session. You can print a list of all of the objects in your current workspace using the `ls()` function. If we start a new *R* session, our workspace will be empty:

```
ls()
```

```
## character(0)
```

Viewing Your Workspace in R

And we'll be able to see some objects if we add them:

```
x <- 20
y <- 30
z <- x + y
df <- data.frame(nums = 1:10, b = letters[1:10])
ls()
```

```
## [1] "df" "x"  "y"  "z"
```

Viewing Your Workspace in *RStudio*

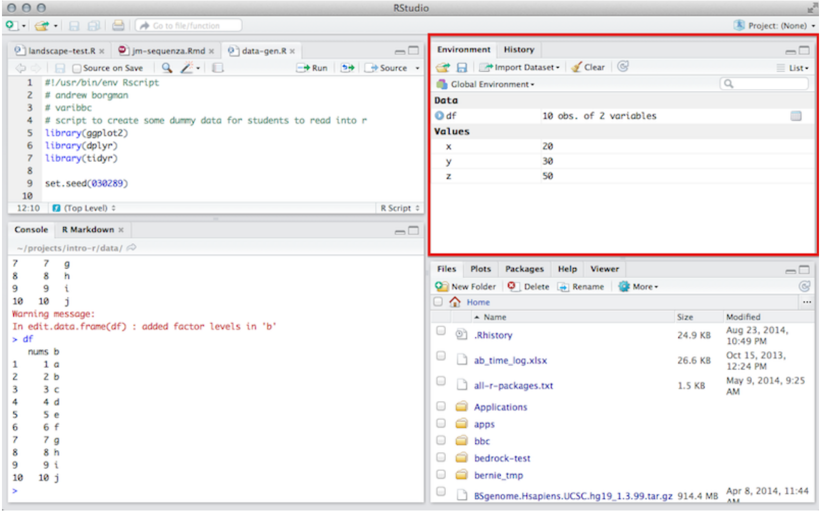


Figure 2: env-panel

Viewing Your Data in *RStudio*

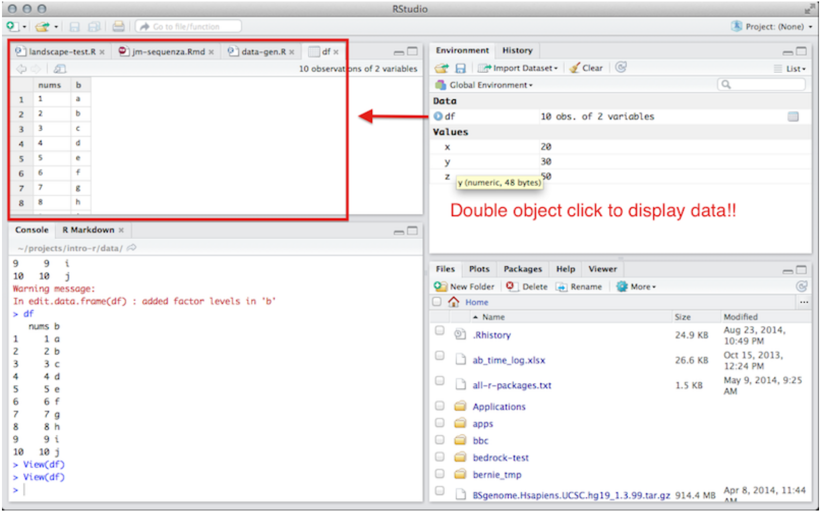


Figure 3: env-click

Saving Your Workspace for Later

- `save.image`: saves a snapshot of entire workspace to a file
  - Typical usage: `save.image('my-work.RData')`
- `save`: saves a snapshot of a few specified objects to a file.
  - Useful if you only want to save one or 2 things (like a modified data set).
  - To save, `df`, the `data.frame` we created in the last section:
  - `save(df, file = 'my-dataframe.RData')`
- `load`: loads your saved `*.RData` files back in to *R*
  - To load, `df`, the `data.frame` we just saved:
  - `load('my-dataframe.RData')`

Clearing Your Workspace

- There are two main ways to clear your workspace in *R*/*RStudio*
  1. using the `rm()` function to remove objects and
  2. using the *Clear* button in *RStudio*.
- `rm()` lets you remove objects from your *R* session when they aren't needed
- Example of the usage:

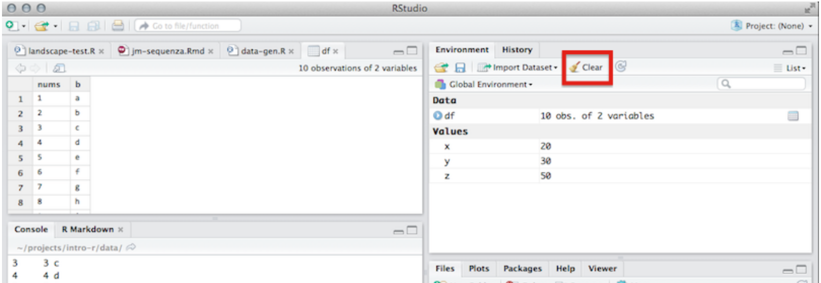
```
> ls() # start in an empty workspace
character(0)
> y <- 1
> z <- 1
> ls() # can see the two objects we created
[1] "y" "z"
> rm(y) # remove y
> ls()
[1] "z"
```

Clearing Your Workspace

You can remove *all* objects in your workspace by using `ls()` to generate a vector of all the objects that have been created, and passing that to the `rm()` function:

```
rm(list = ls(all = TRUE))
```

You can also use a button in *RStudio*'s *Environment* panel to remove all of the objects in your workspace. *RStudio* will prompt you asking if you are sure you want to go through with deleting all objects, choosing *Yes* will permanently delete all objects in the workspace.



Built In Data

Built In Data

R has a number of built in data sets one can access. Built in data is loaded using the `data()` function. Running `data()` with no arguments lists all available data sets. Running `data()` with a valid name of a data set loads that data into the workspace:

```
ls()
```

```
## character(0)
```

```
data(ChickWeight)
ls()
```

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

Finally, some functions. To the Lab!