R Objects & Programmatic Data Manipulation

Fundamental Techniques in Data Science

Kyle M. Lang

Department of Methodology & Statistics
Utrecht University

Outline

R Objects & Data Types

Vectors

Vectors are the simplest kind of R object.

- There is no concept of a “scalar” in R.

Vectors come in one of six “atomic modes”:

- numeric/double
- logical
- character
- integer
- complex
- raw

Vectors

```
(v1 <- vector("numeric", 3))

[1] 0 0 0

(v2 <- vector("logical", 3))

[1] FALSE FALSE FALSE

(v3 <- vector("character", 3))

[1] "" "" ""

(v4 <- vector("integer", 3))

[1] 0 0 0

(v5 <- vector("complex", 3))

[1] 0+0i 0+0i 0+0i

(v6 <- vector("raw", 3))

[1] 00 00 00
```

Generating Vectors

We have many ways of generating vectors.

```
(y1 <- c(1, 2, 3))

[1] 1 2 3

(y2 <- c(TRUE, FALSE, TRUE, TRUE))

[1]  TRUE FALSE  TRUE  TRUE

(y3 <- c("bob", "suzy", "danny"))

[1] "bob"   "suzy"  "danny"

1:5

[1] 1 2 3 4 5

1.2:5.3

[1] 1.2 2.2 3.2 4.2 5.2
```

Generating Vectors

```
rep(33, 4)

[1] 33 33 33 33

rep(1:3, 3)

[1] 1 2 3 1 2 3 1 2 3

rep(y3, each = 2)

[1] "bob"   "bob"   "suzy"  "suzy"  "danny" "danny"

seq(0, 1, 0.25)

[1] 0.00 0.25 0.50 0.75 1.00
```

The Three Most Useful Data Types

Numeric

```
(a <- 1:5)

[1] 1 2 3 4 5
```

Character

```
(b <- c("foo", "bar"))

[1] "foo" "bar"
```

Logical

```
(c <- c(TRUE, FALSE))

[1]  TRUE FALSE
```

Combining Data Types in Vectors

What happens if we try to concatenate different data types?

```
c(a, b)

[1] "1"   "2"   "3"   "4"   "5"   "foo" "bar"

c(b, c)

[1] "foo"   "bar"   "TRUE"  "FALSE"

c(a, c)

[1] 1 2 3 4 5 1 0
```

Matrices

Matrices generalize vectors by adding a dimension attribute.

```
(m1 <- matrix(a, nrow = 5, ncol = 2))

     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    4
[5,]    5    5

attributes(v1)

NULL

attributes(m1)

$dim
[1] 5 2
```

Matrices

Matrices are populated in column-major order, by default.

```
(m2 <- matrix(1:9, 3, 3))

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

The `byrow = TRUE` option allows us to fill by row-major order.

```
(m3 <- matrix(1:9, 3, 3, byrow = TRUE))

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Mixing Data Types in Matrices

Like vectors, matrices can only hold one type of data.

```
cbind(c, letters[1:5])

     c
[1,] "TRUE"  "a"
[2,] "FALSE" "b"
[3,] "TRUE"  "c"
[4,] "FALSE" "d"
[5,] "TRUE"  "e"

cbind(c, c(TRUE, TRUE, FALSE, FALSE, TRUE))

         c
[1,]  TRUE  TRUE
[2,] FALSE  TRUE
[3,]  TRUE FALSE
[4,] FALSE FALSE
[5,]  TRUE  TRUE
```

Lists

Lists are the workhorse of R data objects.

- An R list can hold an arbitrary set of other R objects.

We create lists using the `list()` function.

```
(l1 <- list(1, 2, 3))

[[1]]
[1] 1

[[2]]
[1] 2

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

Lists

```
(l2 <- list("bob", TRUE, 33, 42+3i))

[[1]]
[1] "bob"

[[2]]
[1] TRUE

[[3]]
[1] 33

[[4]]
[1] 42+3i
```

Lists

List elements have no defualt names, but we can define our own.

```
(l3 <- list(name = "bob",
            alive = TRUE,
            age = 33,
            relationshipStatus = 42+3i)
)

$name
[1] "bob"

$alive
[1] TRUE

$age
[1] 33

$relationshipStatus
[1] 42+3i
```

Lists

We can also assign post hoc names via the `names()` function.

```
names(l1) <- c("first", "second", "third")
l1

$first
[1] 1

$second
[1] 2

$third
[1] 3
```

Lists

We can append new elements onto an existing list.

```
(l4 <- list())

list()

l4$people <- c("Bob", "Alice", "Suzy")
l4$money <- 0
l4$logical <- FALSE
l4

$people
[1] "Bob"   "Alice" "Suzy"

$money
[1] 0

$logical
[1] FALSE
```

Lists

The elements inside a list don't really know that they live in a list; they'll pretty much behave as normal.

```
l4$money + 42

[1] 42

paste0("Hello, ", l4$people, "!\n") %>% cat()

Hello, Bob!
 Hello, Alice!
 Hello, Suzy!
```

Data Frames

Data frames are R's way of storing rectangular data sets.

- Each column of a data frame is a vector.
- Each of these vectors can have a different type.

We create data frames using the `data.frame()` function.

```
(d1 <- data.frame(1:6, c(-1, 1), seq(0.1, 0.6, 0.1)))

  X1.6 c..1..1. seq.0.1..0.6..0.1.
1    1       -1                0.1
2    2        1                0.2
3    3       -1                0.3
4    4        1                0.4
5    5       -1                0.5
6    6        1                0.6
```

Data Frames

```
(d2 <- data.frame(x = 1:6, y = c(-1, 1), z = seq(0.1, 0.6, 0.1)))

  x  y   z
1 1 -1 0.1
2 2  1 0.2
3 3 -1 0.3
4 4  1 0.4
5 5 -1 0.5
6 6  1 0.6
```

Data Frames

```
(d3 <- data.frame(a = sample(c(TRUE, FALSE), 10, replace = TRUE),
                  b = sample(c("foo", "bar"), 10, replace = TRUE),
                  c = runif(10)
                  )
)

       a   b         c
1  FALSE foo 0.4708232
2   TRUE foo 0.2701596
3   TRUE bar 0.6199154
4  FALSE bar 0.2078104
5   TRUE bar 0.4912943
6  FALSE bar 0.1840306
7  FALSE bar 0.5438698
8   TRUE bar 0.5755350
9  FALSE bar 0.7557042
10 FALSE bar 0.8405729
```

Data Frames

```
(d4 <- data.frame(matrix(NA, 10, 3)))

   X1 X2 X3
1  NA NA NA
2  NA NA NA
3  NA NA NA
4  NA NA NA
5  NA NA NA
6  NA NA NA
7  NA NA NA
8  NA NA NA
9  NA NA NA
10 NA NA NA
```

Data Frames

Data frames are actually lists of vectors (representing the columns).

```
is.data.frame(d3)

[1] TRUE

is.list(d3)

[1] TRUE
```

Although they look like rectangular ”matrices”, from R’s perspective a data frame IS NOT a matrix.

```
is.matrix(d3)

[1] FALSE
```

Data Frames

We cannot treat a data frame like a matrix. E.g., matrix algebra doesn't work with data frames.

```
d1 %*% t(d2)

Error in d1 %*% t(d2):  requires numeric/complex matrix/vector arguments

as.matrix(d1) %*% t(as.matrix(d2))

     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] 2.01  1.02  4.03  3.04  6.05  5.06
[2,] 1.02  5.04  5.06  9.08  9.10 13.12
[3,] 4.03  5.06 10.09 11.12 16.15 17.18
[4,] 3.04  9.08 11.12 17.16 19.20 25.24
[5,] 6.05  9.10 16.15 19.20 26.25 29.30
[6,] 5.06 13.12 17.18 25.24 29.30 37.36
```

Factors

Factors are R’s way of repesenting nominal variables.

- We can create a factor using the `factor()` function.

```
(f1 <- factor(sample(1:3, 15, TRUE), labels = c("red", "yellow", "blue")))

 [1] red    yellow blue   red    blue   blue   yellow red
 [9] red    red    yellow blue   blue   blue   yellow
Levels: red yellow blue
```

Factors

Factors are integer vectors with a *levels* attribute and a *factor* class.

```
typeof(f1)

[1] "integer"

attributes(f1)

$levels
[1] "red"    "yellow" "blue"

$class
[1] "factor"
```

The levels are just group labels.

```
levels(f1)

[1] "red"    "yellow" "blue"
```

Factors

Even though a factor's data are represented by an integer vector, R does not consider factors to be interger/numeric data.

```
is.numeric(f1)

[1] FALSE

is.integer(f1)

[1] FALSE
```

Factors represent nominal variables, so we cannot do math with factors.

```
f1 + 1

 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

mean(f1)

[1] NA
```

PROGRAMMATIC DATA MANIPULATION

Tidyverse Solutions

The **dplyr** package provides many ways to subset data, but two functions are most frequently useful.

- `select()` : subset columns
- `filter()` : subset rows

```
library(dplyr)
```

Tidyverse Solutions

The **dplyr** package provides three primary transformation functions

- `recode()` : recode the levels of a variable
- `mutate()` : general purpose transformation & feature building

```
library(dplyr)
```

Tidyverse Solutions

The **dplyr** package provides three primary transformation functions

- `arrange()` : sort/order rows
- `relocate()` : move columns

```
library(dplyr)
```

What are pipes?

The %>% symbol represents the *pipe* operator.

- We use the pipe operator to compose functions into a *pipeline*.

The following code represents a pipeline.

```
firstBoys <-
  read_sav("../data/boys.sav") %>%
  head()
```

This pipeline replaces the following code.

```
firstBoys <- head(read_sav("../data/boys.sav"))
```

Why are pipes useful?

Let's assume that we want to:

1. Load data
2. Transform a variable
3. Filter cases
4. Select columns

Without a pipe, we may do something like this:

```
library(haven)
library(dplyr)

boys <- read_sav("../../data/boys.sav")
boys <- transform(boys, hgt = hgt / 100)
boys <- filter(boys, age > 15)
boys <- subset(boys, select = c(hgt, wgt, bmi))
```

Why are pipes useful?

With the pipe, we could do something like this:

```
library(haven)
library(dplyr)

boys <-
  read_sav("../../data/boys.sav") %>%
  transform(hgt = hgt / 100) %>%
  filter(age > 15) %>%
  subset(select = c(hgt, wgt, bmi))
```

With a pipeline, our code more clearly represents the sequence of steps in our analysis.

Benefits of Pipes

When you use pipes, your code becomes more readable.

- Operations are structured from left to right instead of in to out.
- You can avoid many nested function calls.
- You don't have to keep track of intermediate objects.
- It's easy to add steps to the sequence.

In RStudio, you can use a keyboard shortcut to insert the %>% symbol.

- Windows/Linux: *ctrl* + *shift* + *m*
- Mac: *cmd* + *shift* + *m*

What do pipes do?

Pipes compose R functions without nesting.

- `f(x)` becomes `x` %>% `f()`

```
mean(rnorm(10))
```

```
[1] 0.8904119
```

```
rnorm(10) %>% mean()
```

```
[1] 0.293541
```

What do pipes do?

Multiple function arguments are fine.

- `f(x, y)` becomes `x` `%>%` `f(y)`

```
cor(boys, use = "pairwise.complete.obs")

          hgt       wgt       bmi
hgt 1.0000000 0.6100784 0.1758781
wgt 0.6100784 1.0000000 0.8841304
bmi 0.1758781 0.8841304 1.0000000

boys %>% cor(use = "pairwise.complete.obs")

          hgt       wgt       bmi
hgt 1.0000000 0.6100784 0.1758781
wgt 0.6100784 1.0000000 0.8841304
bmi 0.1758781 0.8841304 1.0000000
```

What do pipes do?

Composing more than two functions is easy, too.

- `h(g(f(x)))` becomes `x` %>% `f` %>% `g` %>% `h`

```
max(na.omit(subset(boys, select = wgt)))

[1] 117.4

boys %>%
  subset(select = wgt) %>%
  na.omit() %>%
  max()

[1] 117.4
```

The Role of `.` in a Pipeline

In the expression `a` %>% `f(arg1, arg2, arg3)`, `a` will be "piped into" `f()` as `arg1`.

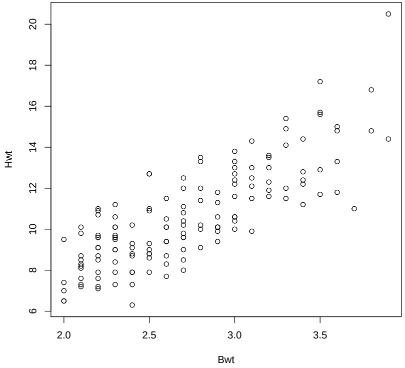
```
data(cats, package = "MASS")
cats %>% plot(Hwt ~ Bwt)

Error in text.default(x, y, txt, cex = cex, font = font):  invalid
mathematical annotation
```

Clearly, we have a problem if we pipe our data into the wrong argument.

- We can change this behavior with the `.` symbol.
- The `.` symbol acts as a placeholder for the data in a pipeline.

The Role of `.` in a Pipeline

```r
cats %>% plot(Hwt ~ Bwt, data = .)
```

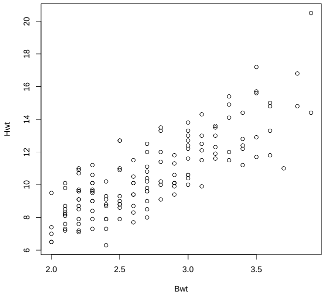


Exposition Pipe: %$%

There are several different flavors of pipe. The *exposition pipe*, %$%, is a particularly useful variant.

- The exposition pipe *exposes* the contents of an object to the next function in the pipeline.

```
cats %$% plot(Hwt ~ Bwt)
```



Performing a T-Test in a Pipeline

```
cats %$% t.test(Hwt ~ Sex)
```

```


Welch Two Sample t-test

data:  Hwt by Sex
t = -6.5179, df = 140.61, p-value = 1.186e-09
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.763753 -1.477352
sample estimates:
mean in group F mean in group M
       9.202128       11.322680
```

The above is equivalent to either of the following.

```
cats %>% t.test(Hwt ~ Sex, data = .)
t.test(Hwt ~ Sex, data = cats)
```

Storing the Results

We can use normal assignment to save the result of a pipeline.

```
catsTest <- cats %$% t.test(Bwt ~ Sex)

catsTest


Welch Two Sample t-test

data:  Bwt by Sex
t = -8.7095, df = 136.84, p-value = 8.831e-15
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.6631268 -0.4177242
sample estimates:
mean in group F mean in group M
       2.359574        2.900000
```

References