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.

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

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

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

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

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

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

```r
1:5
```

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

```r
1.2:5.3
```

```
[1] 1.2 2.2 3.2 4.2 5.2
```

Generating Vectors

```r
rep(33, 4)
```

```
[1] 33 33 33 33
```

```r
rep(1:3, 3)
```

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

```r
rep(y3, each = 2)
```

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

```r
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 default 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

```r
(d3 <- data.frame(a = sample(c(TRUE, FALSE), 8, replace = TRUE),
                  b = sample(c("foo", "bar"), 8, replace = TRUE),
                  c = runif(8)
                  )
)
```

```
      a   b         c
1 FALSE bar 0.3218011
2  TRUE bar 0.5110387
3  TRUE foo 0.8472829
4 FALSE foo 0.1928677
5  TRUE bar 0.4708232
6 FALSE bar 0.2701596
7 FALSE bar 0.6199154
8  TRUE bar 0.2078104
```

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] yellow red    blue   yellow red    yellow blue   red
 [9] blue   blue   yellow red    red    red    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
```

Data Manipulation

Base R Subsetting

In Base R, we typically use three operators to subset objects:

- `[]`
- `[[]]`
- `$`

Which of these operators we choose to use (and how we implement the chosen operator) will depend on two criteria:

- What type of object are we trying to subset?
- How much of the original typing do we want to keep in the subset?

Atomic Data Objects

To subset vectors and matrices, we can use either `[]` or `[[]]` .

```
(x <- rnorm(8))

[1]  1.17766957 -0.16448325  0.94412165  2.07070102
[5]  1.10427674  0.39615080 -0.09361605  0.13621028

x[1:3]

[1]  1.1776696 -0.1644832  0.9441216

x[2]

[1] -0.1644832

x[c(2, 5, 7)]

[1] -0.16448325  1.10427674 -0.09361605

x[c(TRUE, FALSE)]

[1]  1.17766957  0.94412165  1.10427674 -0.09361605
```

Atomic Data Objects

The `[[]]` operator can only select a single element.

```
x[[2]]

[1] -0.1644832

x[[1:3]]

Error in x[[1:3]]:  attempt to select more than one element in vectorIndex
```

Atomic Data Objects

To subset matrices, we need to differentiate the dimensions.

```
(y <- matrix(x, 6, 4))

           [,1]        [,2]        [,3]        [,4]
[1,]  1.1776696 -0.09361605  1.10427674  0.94412165
[2,] -0.1644832  0.13621028  0.39615080  2.07070102
[3,]  0.9441216  1.17766957 -0.09361605  1.10427674
[4,]  2.0707010 -0.16448325  0.13621028  0.39615080
[5,]  1.1042767  0.94412165  1.17766957 -0.09361605
[6,]  0.3961508  2.07070102 -0.16448325  0.13621028

y[2, 2]

[1] 0.1362103

y[1:3, 1]

[1]  1.1776696 -0.1644832  0.9441216
```

Atomic Data Objects

We can select sub-matrices and mix different indexing styles.

```
y[1:2, c(2, 4)]

            [,1]      [,2]
[1,] -0.09361605 0.9441216
[2,]  0.13621028 2.0707010

y[c(1:3, 5), c(FALSE, TRUE, TRUE, FALSE)]

            [,1]        [,2]
[1,] -0.09361605  1.10427674
[2,]  0.13621028  0.39615080
[3,]  1.17766957 -0.09361605
[4,]  0.94412165  1.17766957
```

Atomic Data Objects

Leaving the rows or columns section empty will return all rows or columns, respectively.

```
y[ , 2]

[1] -0.09361605  0.13621028  1.17766957 -0.16448325
[5]  0.94412165  2.07070102

y[2:5, ]

           [,1]       [,2]        [,3]        [,4]
[1,] -0.1644832  0.1362103  0.39615080  2.07070102
[2,]  0.9441216  1.1776696 -0.09361605  1.10427674
[3,]  2.0707010 -0.1644832  0.13621028  0.39615080
[4,]  1.1042767  0.9441216  1.17766957 -0.09361605
```

Atomic Data Objects

The `[[]]` operator can still select only a single element.

```
y[[2, 2]]

[1] 0.1362103

y[[1:3, 2]]

Error in y[[1:3, 2]]:  attempt to select more than one element in get1index
```

Lists

We can use all three operators to subset lists.

```
l4$people

[1] "Bob"   "Alice" "Suzy"

l4[1]

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

l4[["people"]]

[1] "Bob"   "Alice" "Suzy"
```

Lists

As expected, we cannot select multiple list elements with `[[]]`.

```
l4[1:2]

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

$money
[1] 0

l4[[1:2]]

[1] "Alice"
```

Lists

The relative behavior of `[]` and `[[]]` is more important for lists.

```
(tmp1 <- l4[1])

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

class(tmp1)

[1] "list"

(tmp2 <- l4[[1]])

[1] "Bob"   "Alice" "Suzy"

class(tmp2)

[1] "character"
```

Data Frames

We can subset the columns of a data frame using list semantics.

```
d3$a
```

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

```
d3[1]
```

```
      a
1 FALSE
2  TRUE
3  TRUE
4 FALSE
5  TRUE
6 FALSE
7 FALSE
8  TRUE
```

Data Frames

```
d3["a"]

      a
1 FALSE
2  TRUE
3  TRUE
4 FALSE
5  TRUE
6 FALSE
7 FALSE
8  TRUE

d3[["a"]]

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

Data Frames

We can also use matrix-style subsetting.

```
d3[1:5, 1:2]

      a   b
1 FALSE bar
2  TRUE bar
3  TRUE foo
4 FALSE foo
5  TRUE bar

d3[c(1, 3, 5, 7), letters[2:3]]

    b         c
1 bar 0.3218011
3 foo 0.8472829
5 bar 0.4708232
7 bar 0.6199154
```

Data Frames

The list-style subsetting can have advantages.

```
(tmp1 <- d3[ , 2])

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

(tmp2 <- d3[2])

    b
1 bar
2 bar
3 foo
4 foo
5 bar
6 bar
7 bar
8 bar
```

Data Frames

Single columns are returned as $N \times 1$ data frames, rather than N-element vectors.

```r
class(tmp1)
```

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

```r
class(tmp2)
```

```
[1] "data.frame"
```

Overwriting Values

We also use subsetting syntax to overwrite values in an R object.

```
x[2:3] <- NA
x

[1]  1.17766957          NA          NA  2.07070102
[5]  1.10427674  0.39615080 -0.09361605  0.13621028

l4$people <- "None"
l4

$people
[1] "None"

$money
[1] 0

$logical
[1] FALSE
```

Overwriting Values

```
y[1:3, 2:4] <- -1
print(y, digits = 3)

       [,1]   [,2]   [,3]    [,4]
[1,]  1.178 -1.000 -1.000 -1.0000
[2,] -0.164 -1.000 -1.000 -1.0000
[3,]  0.944 -1.000 -1.000 -1.0000
[4,]  2.071 -0.164  0.136  0.3962
[5,]  1.104  0.944  1.178 -0.0936
[6,]  0.396  2.071 -0.164  0.1362
```

```
d4      <- d3
d4[1:2] <- rgamma(nrow(d4) * 2, 10)
print(d4, digits = 3)

      a     b     c
1  9.99 11.53 0.322
2 10.35 10.23 0.511
3 16.20  5.39 0.847
4 16.81  8.01 0.193
5 15.30  8.24 0.471
6 12.66  8.96 0.270
7  9.95 10.17 0.620
8 11.74 12.88 0.208
```

Tidyverse Subsetting

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

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

```
library(dplyr)
```

Subsetting Columns: `select()`

The `dplyr::select()` function provides a very intuitive syntax for variable selection and column-wise subsetting.

```
select(d3, a, b)

      a   b
1 FALSE bar
2  TRUE bar
3  TRUE foo
4 FALSE foo
5  TRUE bar
6 FALSE bar
7 FALSE bar
8  TRUE bar
```

```
select(d3, -a)

    b         c
1 bar 0.3218011
2 bar 0.5110387
3 foo 0.8472829
4 foo 0.1928677
5 bar 0.4708232
6 bar 0.2701596
7 bar 0.6199154
8 bar 0.2078104
```

Subsetting Rows

The `dplyr::filter()` function provides easy row subsetting:

```
filter(d3, c > 0.5)

      a   b         c
1  TRUE bar 0.5110387
2  TRUE foo 0.8472829
3 FALSE bar 0.6199154
```

```
filter(d3, c > 0.15, b == "foo")

      a   b         c
1  TRUE foo 0.8472829
2 FALSE foo 0.1928677
```

We can achieve the same effect via logical indexing in Base R:

```
d3[d3$c > 0.5, ]

      a   b         c
2  TRUE bar 0.5110387
3  TRUE foo 0.8472829
7 FALSE bar 0.6199154
```

```
d3[d3$c > 0.15 & d3$b == "foo", ]

      a   b         c
3  TRUE foo 0.8472829
4 FALSE foo 0.1928677
```

Base R Variable Transformations

There is nothing very special about the process of transforming variables in Base R.

```r
d4   <- d3
d4$d <- scale(d4$c)
d4$e <- !d4$a
d4

      a   b         c          d     e
1 FALSE bar 0.3218011 -0.4771996  TRUE
2  TRUE bar 0.5110387  0.3557774 FALSE
3  TRUE foo 0.8472829  1.8358417 FALSE
4 FALSE foo 0.1928677 -1.0447329  TRUE
5  TRUE bar 0.4708232  0.1787590 FALSE
6 FALSE bar 0.2701596 -0.7045129  TRUE
7 FALSE bar 0.6199154  0.8350260  TRUE
8  TRUE bar 0.2078104 -0.9789586 FALSE
```

```r
d4   <- d3
d4$c <- scale(d4$c, scale = FALSE)
d4$a <- as.numeric(d4$a)
d4

  a   b           c
1 0 bar -0.10841126
2 1 bar  0.08082628
3 1 foo  0.41707055
4 0 foo -0.23734472
5 1 bar  0.04061085
6 0 bar -0.16005280
7 0 bar  0.18970305
8 1 bar -0.22240197
```

Tidyverse Variable Transformations

The `mutate()` function from **dplyr** is the workhorse of Tidyverse transformation functions.

```
mutate(d3, d = rbinom(nrow(d3), 1, c))

      a   b         c d
1 FALSE bar 0.3218011 0
2  TRUE bar 0.5110387 1
3  TRUE foo 0.8472829 1
4 FALSE foo 0.1928677 0
5  TRUE bar 0.4708232 1
6 FALSE bar 0.2701596 0
7 FALSE bar 0.6199154 1
8  TRUE bar 0.2078104 0
```

```
mutate(d3,
       d = rbinom(nrow(d3), 1, c),
       e = d * c
       )

      a   b         c d         e
1 FALSE bar 0.3218011 1 0.3218011
2  TRUE bar 0.5110387 0 0.0000000
3  TRUE foo 0.8472829 1 0.8472829
4 FALSE foo 0.1928677 0 0.0000000
5  TRUE bar 0.4708232 1 0.4708232
6 FALSE bar 0.2701596 0 0.0000000
7 FALSE bar 0.6199154 1 0.6199154
8  TRUE bar 0.2078104 0 0.0000000
```

Sorting & Ordering

To sort a single vector, the best option is the Base R `sort()` function.

```
sort(d3$c)

[1] 0.1928677 0.2078104 0.2701596 0.3218011 0.4708232
[6] 0.5110387 0.6199154 0.8472829

sort(d3$c, decreasing = TRUE)

[1] 0.8472829 0.6199154 0.5110387 0.4708232 0.3218011
[6] 0.2701596 0.2078104 0.1928677
```

To sort the rows of a data frame according to the order of one of its columns, the `dplyr::arrange()` works best.

- You can use the Base R `order()` function to achieve similar results.
- The behavior of `order()` is (extremely) unintuitive.

Tidyverse Ordering

Using `dplyr::arrange()` could not be simpler.

```r
arrange(d3, a)
```

```
      a   b         c
1 FALSE bar 0.3218011
2 FALSE foo 0.1928677
3 FALSE bar 0.2701596
4 FALSE bar 0.6199154
5  TRUE bar 0.5110387
6  TRUE foo 0.8472829
7  TRUE bar 0.4708232
8  TRUE bar 0.2078104
```

```r
arrange(d3, -c)
```

```
      a   b         c
1  TRUE foo 0.8472829
2 FALSE bar 0.6199154
3  TRUE bar 0.5110387
4  TRUE bar 0.4708232
5 FALSE bar 0.3218011
6 FALSE bar 0.2701596
7  TRUE bar 0.2078104
8 FALSE foo 0.1928677
```

```r
arrange(d3, -a, c)
```

```
      a   b         c
1  TRUE bar 0.2078104
2  TRUE bar 0.4708232
3  TRUE bar 0.5110387
4  TRUE foo 0.8472829
5 FALSE foo 0.1928677
6 FALSE bar 0.2701596
7 FALSE bar 0.3218011
8 FALSE bar 0.6199154
```

Pipes

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 <-
    readRDS("boys.rds") %>%
    head()
```

This pipeline replaces the following code.

```
firstBoys <- head(readRDS("boys.rds"))
```

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:

```r
library(dplyr)

boys <- readRDS("../../../data/boys.rds")
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:

```
boys <-
    readRDS("../../../data/boys.rds") %>%
    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] 0.05223587

rnorm(10) %>% mean()

[1] 0.04802827
```

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`.

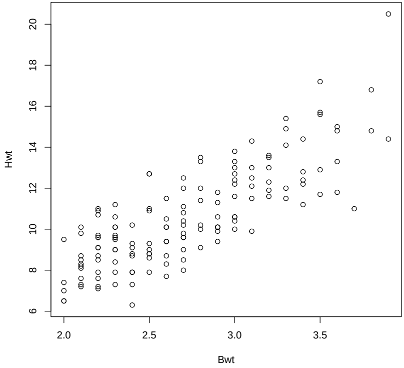
```
data(cats, package = "MASS")
cats %>% plot(Hwt ~ Bwt)

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

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

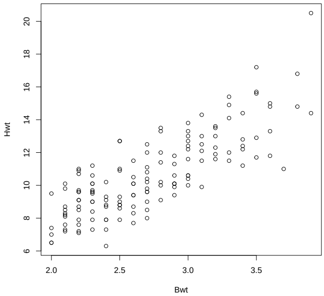


Exposition Pipe: %$%

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 between group F and group M
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)
```