Class 3 Review

Damian Thomas

2017-02-03

Topics

1. Jargon
   - Brackets
   - Other Symbols
2. R Data Structures
   - Atomic Vectors
   - Lists
   - Factors
   - Data Frames
3. Subsetting
   - Vectors
   - Lists
   - Data Frames
4. Iteration
   - Iterate by element
   - Iterate by index
   - Nested Loops
5. Functions
   - Single Parameter
   - No Parameters
   - Braces
   - Named Parameters
6. References

Brackets

Multiple symbols are called "brackets". To minimize confusion we will use the following terms to describe each[1]

Symbols	Names
()	"parenthesis", "round brackets", or "parens"
[]	"brackets" or "square brackets"
{}	"braces" or "curly brackets"
<>	"angle brackets" or "less than" & "greater than"

[1] Source: `https://en.wikipedia.org/wiki/Bracket`

Other Symbols

Symbols	Names
\	"backslash"
/	"slash" or "forward slash"
|	"pipe" or "vertical pipe"
!	"exclamation point" or "bang"

R Data Structures by Content Type and Number of Dimensions[2]

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

[2] Source: Advanced R (`http://adv-r.had.co.nz/`)

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Atomic vectors: Concept

Atomic Vector: one data type, one dimension

Atomic vectors: Concept

Atomic Vector: one data type, one dimension

Analogous to a...

- Single column in a data set
- Single time series with evenly spaced observations
- Mathematical vector in linear algebra
- Array data structure in computer science

The Combine Function: c()

- Purpose: Create vectors
- Input: one or more atomic vectors[3]
- Output: one atomic vector, with one type and one dimension

[3] Other data structures are permitted. See documentation: ?c()

The Combine Function: c(), examples

Separate inputs with commas (required) **and** spaces (preferred)

```
> c(1, 2, 3) # returns a numeric vector
```

[4] The data type is automatically changed

The Combine Function: c(), examples

Separate inputs with commas (required) **and** spaces (preferred)

```
> c(1, 2, 3) # returns a numeric vector
```

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

[4] The data type is automatically changed

The Combine Function: c(), examples

Separate inputs with commas (required) **and** spaces (preferred)

```
> c(1, 2, 3) # returns a numeric vector
```

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

Mixed types trigger coercion[4] to a single type.

```
> c("one", 2, 3.14, FALSE)  # returns a character vector
```

[4] The data type is automatically changed

The Combine Function: c(), examples

Separate inputs with commas (required) **and** spaces (preferred)

```
> c(1, 2, 3) # returns a numeric vector
```

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

Mixed types trigger coercion[4] to a single type.

```
> c("one", 2, 3.14, FALSE)  # returns a character vector
```

```
[1] "one"   "2"     "3.14"  "FALSE"
```

[4] The data type is automatically changed

The Combine Function: c(), examples

Separate inputs with commas (required) **and** spaces (preferred)

```
> c(1, 2, 3) # returns a numeric vector
```

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

Mixed types trigger coercion[4] to a single type.

```
> c("one", 2, 3.14, FALSE)  # returns a character vector
```

```
[1] "one"   "2"     "3.14"  "FALSE"
```

Vector inputs are concatenated

```
> foo <- c("T", "W")
> bar <- c("Th", "F")
> c("M", foo, c(bar, "S", "Su")) # returns 7 elements
```

[4] The data type is automatically changed

The Combine Function: c(), examples

Separate inputs with commas (required) **and** spaces (preferred)

```
> c(1, 2, 3) # returns a numeric vector
```

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

Mixed types trigger coercion[4] to a single type.

```
> c("one", 2, 3.14, FALSE)  # returns a character vector
```

```
[1] "one"   "2"     "3.14"  "FALSE"
```

Vector inputs are concatenated

```
> foo <- c("T", "W")
> bar <- c("Th", "F")
> c("M", foo, c(bar, "S", "Su")) # returns 7 elements
```

```
[1] "M"  "T"  "W"  "Th" "F"  "S"  "Su"
```

[4] The data type is automatically changed

Atomic Vectors, Example: Recent FRB Chairs

```
> surname <- c("Yellen", "Bernanke", "Greenspan")
```

Atomic Vectors, Example: Recent FRB Chairs

```
> surname <- c("Yellen", "Bernanke", "Greenspan")
> appointed <- c(2014, 2006, 1987)
```

Atomic Vectors, Example: Recent FRB Chairs

```
> surname <- c("Yellen", "Bernanke", "Greenspan")
> appointed <- c(2014, 2006, 1987)
> has_beard <- c(FALSE, TRUE, FALSE)
```

Atomic Vectors, Example: Recent FRB Chairs

What class & length will each vector have?

```
> str(surname)
> str(appointed)
> str(has_beard)
```

Atomic Vectors, Example: Recent FRB Chairs

What class & length will each vector have?

```
> str(surname)
> str(appointed)
> str(has_beard)
```

```
 chr [1:3] "Yellen" "Bernanke" "Greenspan"

 num [1:3] 2014 2006 1987

 logi [1:3] FALSE TRUE FALSE
```

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Lists: Concept

List: many types, one dimension

Lists: Concept

List: many types, one dimension

Analogous to...

- Single row in a data set
- Collection of features describing a single of entity
- Associative array structure in computer science

The List Function: `list()`

- Purpose: create lists
- Input: one or more objects (of any type)
- Output: one list object, with one or more types and one dimension

The List Function: `list()`, examples

Separate inputs with commas (required) and spaces (preferred)

```
> list("Yes", TRUE, 3.14, c(foo, bar)) # returns 4-elments
```

The List Function: `list()`, examples

Separate inputs with commas (required) and spaces (preferred)

```
> list("Yes", TRUE, 3.14, c(foo, bar)) # returns 4-elments
```

```
[[1]]
[1] "Yes"

[[2]]
[1] TRUE

[[3]]
[1] 3.14

[[4]]
[1] "T"  "W"  "Th" "F"
```

Lists, Example: Recent FRB Chairs

```
> current_chair <- list("Yellen", 2014, FALSE)
```

Lists, Example: Recent FRB Chairs

```
> current_chair <- list("Yellen", 2014, FALSE)
> prior_chair   <- list("Bernanke", 2006, TRUE)
```

Lists, Example: Recent FRB Chairs

```
> current_chair <- list("Yellen", 2014, FALSE)
> prior_chair   <- list("Bernanke", 2006, TRUE)
> earlier_chair <- list("Greenspan", 1987, FALSE)
```

Lists, Example: Recent FRB Chairs

```
> current_chair <- list("Yellen", 2014, FALSE)
> prior_chair   <- list("Bernanke", 2006, TRUE)
> earlier_chair <- list("Greenspan", 1987, FALSE)
```

Assign names to list elements

```
> listnames <- c("surname", "appointed", "has_beard")
> names(current_chair) <- listnames
> names(prior_chair)   <- listnames
> names(earlier_chair) <- listnames
```

Lists, Example: Recent FRB Chairs

What class & length will each list have?

Lists, Example: Recent FRB Chairs

What class & length will each list have?

```
> str(current_chair)
> str(prior_chair)
> str(earlier_chair)
```

Lists, Example: Recent FRB Chairs

What class & length will each list have?

```
> str(current_chair)
> str(prior_chair)
> str(earlier_chair)
```

```
List of 3
 $ surname  : chr "Yellen"
 $ appointed: num 2014
 $ has_beard: logi FALSE

List of 3
 $ surname  : chr "Bernanke"
 $ appointed: num 2006
 $ has_beard: logi TRUE

List of 3
 $ surname  : chr "Greenspan"
 $ appointed: num 1987
 $ has_beard: logi FALSE
```

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Factors: Concept

Factor Vector: one type, one dimension, contents restricted to a specified set of discrete values

Factors: Concept

Factor Vector: one type, one dimension, contents restricted to a specified set of discrete values
Analogous to...

- Discrete variable with few values, such as sex or employment status

The Factor Function: `factor()`

- Purpose: create factor vectors
- Input: any atomic vector (required), a vector of possible values (optional), and a vector of corresponding labels (optional)[5]
- Output: an integer vector with embedded attributes to define the set of possible levels and their labels

[5] Refer to the documentation `?factor()`

The Factor Function: `factor()`, examples

Raw data

```
> rps_data <- c("Rock", "Paper", "Scissors", "Rock",
+               "Scissors", "Paper", "Paper",
+               "Scissors", "Rock", "Rock", "Paper")
```

The Factor Function: `factor()`, examples

Raw data

```
> rps_data <- c("Rock", "Paper", "Scissors", "Rock",
+               "Scissors", "Paper", "Paper",
+               "Scissors", "Rock", "Rock", "Paper")
```

Convert to a factor

```
> rps <- factor(rps_data,
+               levels = c("Rock", "Paper", "Scissors"))
```

The Factor Function: `factor()`, examples

Raw data

```
> rps_data <- c("Rock", "Paper", "Scissors", "Rock",
+               "Scissors", "Paper", "Paper",
+               "Scissors", "Rock", "Rock", "Paper")
```

Convert to a factor

```
> rps <- factor(rps_data,
+               levels = c("Rock", "Paper", "Scissors"))
```

View object structure – data stored as integer values

```
> str(rps)
```

```
 Factor w/ 3 levels "Rock","Paper",..: 1 2 3 1 3 2 2 3 1 1
```

The Factor Function: `factor()`, examples

Subset

```
> rps[3]
```

```
[1] Scissors
Levels: Rock Paper Scissors
```

The Factor Function: `factor()`, examples

Subset

```
> rps[3]
```

```
[1] Scissors
Levels: Rock Paper Scissors
```

Attempt to overwrite with invalid level

```
> rps[3] <- "foo"
```

The Factor Function: `factor()`, examples

Subset

```
> rps[3]
```

```
[1] Scissors
Levels: Rock Paper Scissors
```

Attempt to overwrite with invalid level

```
> rps[3] <- "foo"
```

The result is NA

```
> rps[3]
```

```
[1] <NA>
Levels: Rock Paper Scissors
```

The Factor Function: `factor()`, examples

View all attributes of the factor

```
> attributes(rps)

$levels
[1] "Rock"     "Paper"    "Scissors"

$class
[1] "factor"
```

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Data Frame: Concept

Data Frame: many types, two dimensions

Data Frame: Concept

Data Frame: many types, two dimensions
Analogous to...

- A bundle of columns (atomic vectors) with the same length
- A data set
  - each row represents an observation
  - each column represents a variable
- Multiple draws from a population with associated measures

The Data Frame Function: `data.frame()`

- Purpose: Creates data frames
- Input: one or more atomic vectors or factors of equal length[6]
- Output: one data frame, with input vectors arranged as columns of data

[6] Shorter vectors will be recycled to meet this criteria

The data frame function: data.frame(), examples

Recall the vectors created earlier:

```
> surname
[1] "Yellen"    "Bernanke"  "Greenspan"
> appointed
[1] 2014 2006 1987
> has_beard
[1] FALSE  TRUE FALSE
```

The data frame function: data.frame(), examples

Construct a data frame using vectors

```
> frb_chair <- data.frame(surname,
+                         appointed,
+                         has_beard)
```

The data frame function: data.frame(), examples

Construct a data frame using vectors

```
> frb_chair <- data.frame(surname,
+                         appointed,
+                         has_beard)

> frb_chair

    surname appointed has_beard
1    Yellen       2014     FALSE
2  Bernanke       2006      TRUE
3 Greenspan       1987     FALSE
```

The data frame function: data.frame(), examples

Examine attributes: object type and dimensions

```
> class(frb_chair)
> nrow(frb_chair)
> ncol(frb_chair)  # same as length(frb_chair)
> str(frb_chair)
```

The data frame function: data.frame(), examples

Examine attributes: object type and dimensions

```
> class(frb_chair)
> nrow(frb_chair)
> ncol(frb_chair)  # same as length(frb_chair)
> str(frb_chair)
```

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

```
[1] 3
```

```
[1] 3
```

```
'data.frame': 3 obs. of  3 variables:
 $ surname  : Factor w/ 3 levels "Bernanke","Greenspan",..
 $ appointed: num  2014 2006 1987
 $ has_beard: logi  FALSE TRUE FALSE
```

The data frame function: data.frame(), examples

Use function option `stringsAsFactors = FALSE` to disable automatic conversion of character vectors to factors

```
> frb_chair <- data.frame(surname,
+                         appointed,
+                         has_beard,
+                         stringsAsFactors = FALSE)
```

The data frame function: data.frame(), examples

Use function option `stringsAsFactors = FALSE` to disable automatic conversion of character vectors to factors

```
> frb_chair <- data.frame(surname,
+                         appointed,
+                         has_beard,
+                         stringsAsFactors = FALSE)

> frb_chair

    surname appointed has_beard
1    Yellen       2014     FALSE
2  Bernanke       2006      TRUE
3 Greenspan       1987     FALSE
```

The data frame function: data.frame(), examples

Examine attributes again

```
> str(frb_chair)
```

The data frame function: data.frame(), examples

Examine attributes again

```
> str(frb_chair)

'data.frame': 3 obs. of  3 variables:
 $ surname  : chr  "Yellen" "Bernanke" "Greenspan"
 $ appointed: num  2014 2006 1987
 $ has_beard: logi  FALSE TRUE FALSE
```

Subsetting

Subsetting operators

- `[` *bracket*
- `[[` *double bracket*
- `$` *dollar sign*

Subsetting

Subsetting operators

- `[` *bracket*
- `[[` *double bracket*
- `$` *dollar sign*

These operators enable **access** to the data stored in R data structures for:

- Retrieving data from an object
- Changing data in an object
- Deleting data from an object

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Subsetting Vectors: use [

How [works

- Given `x`, an atomic vector with `length(x) == n`,
- And `i`, a vector of *positive* integers, with `all(i <= n)`.
- Then, `x[i]` will return the elements in `x` indexed by `i`.
- *In other words*, "return the `i`-th element(s) from `x`"

Subsetting Vectors: use [

Example 1: "x" is the days of the week, "i" is 5

```
> dow <- c("S", "M", "T", "W", "Th", "F", "Sa")
```

Subsetting Vectors: use [

Example 1: "x" is the days of the week, "i" is 5

```
> dow <- c("S", "M", "T", "W", "Th", "F", "Sa")
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 1: "x" is the days of the week, "i" is 5

```
> dow <- c("S", "M", "T", "W", "Th", "F", "Sa")
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[5] # retrieve the 5th element
```

Subsetting Vectors: use [

Example 1: "x" is the days of the week, "i" is 5

```
> dow <- c("S", "M", "T", "W", "Th", "F", "Sa")
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[5] # retrieve the 5th element
[1] "Th"
```

Subsetting Vectors: use [

Example 1: "x" is the days of the week, "i" is 5

```
> dow <- c("S", "M", "T", "W", "Th", "F", "Sa")
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[5] # retrieve the 5th element
[1] "Th"
```

Easy. Let's do another.

Subsetting Vectors: use [

Example 2: "x" is the days of the week, "i" is 2, 4, 6

Subsetting Vectors: use [

Example 2: "x" is the days of the week, "i" is 2, 4, 6

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 2: "x" is the days of the week, "i" is 2, 4, 6

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[c(2, 4, 6)]
```

Subsetting Vectors: use [

Example 2: "x" is the days of the week, "i" is 2, 4, 6

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[c(2, 4, 6)]
[1] "M" "W" "F"
```

Subsetting Vectors: use [

Example 2: "x" is the days of the week, "i" is 2, 4, 6

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[c(2, 4, 6)]
[1] "M" "W" "F"
```

No problem.

Subsetting Vectors: use [

How `[` works with `-i`

- Given `x`, an atomic vector with `length(x) == n`,
- And `i`, a vector of *positive* integers, with `all(i <= n)`.
- Then, `x[-i]` will return all of the elements in `x`, *excluding* those indexed by `abs(i)`.
- *In other words*, "return everything minus the `i`-th element(s) from `x`"

Subsetting Vectors: use [

Example 3: "x" is the days of the week, "i" is -5

Subsetting Vectors: use [

Example 3: "x" is the days of the week, "i" is -5

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 3: "x" is the days of the week, "i" is -5

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[-5] # exclude the 5th element
```

Subsetting Vectors: use [

Example 3: "x" is the days of the week, "i" is -5

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[-5] # exclude the 5th element
[1] "S"  "M"  "T"  "W"  "F"  "Sa"
```

Subsetting Vectors: use [

Example 3: "x" is the days of the week, "i" is -5

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[-5] # exclude the 5th element
[1] "S"  "M"  "T"  "W"  "F"  "Sa"
```

Light work.

Subsetting Vectors: use [

Example 4: "x" is the days of the week, "i" is a negative integer vector

Subsetting Vectors: use [

Example 4: "x" is the days of the week, "i" is a negative integer vector

```
> print(dow)
```

```
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 4: "x" is the days of the week, "i" is a negative integer vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[-c(2, 4, 6)]  # negate i to exclude
```

Subsetting Vectors: use [

Example 4: "x" is the days of the week, "i" is a negative integer vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[-c(2, 4, 6)]  # negate i to exclude
[1] "S"  "T"  "Th" "Sa"
```

Subsetting Vectors: use [

Example 4: "x" is the days of the week, "i" is a negative integer vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[-c(2, 4, 6)]  # negate i to exclude
[1] "S"  "T"  "Th" "Sa"
```

Bored yet?

Subsetting Vectors: use [

How `[` works with *logical* `i`

- Given `x`, an atomic vector with `length(x) == n`,
- And `i`, a vector of *logical* values, with `length(i) == n`.
- Then, `x[i]` will return all of the elements in `x`, corresponding to the `TRUE` values in `i`.
- *In other words*, "Take every element in `x` where `i` is `T`"

Subsetting Vectors: use [

Example 5: "x" is the days of the week, "i" is logical vector

Subsetting Vectors: use [

Example 5: "x" is the days of the week, "i" is logical vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 5: "x" is the days of the week, "i" is logical vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[c(FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE)]  # kee
```

Subsetting Vectors: use [

Example 5: "x" is the days of the week, "i" is logical vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[c(FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE)]  # kee
[1] "M"  "T"  "W"  "Th" "F"
```

Subsetting Vectors: use [

Example 5: "x" is the days of the week, "i" is logical vector

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> dow[c(FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE)]  # kee
[1] "M"  "T"  "W"  "Th" "F"
```

Easy, but tedious.

Subsetting Vectors: use [

How `[` works with an *expression* `i` that resolves to logical values.

- Given `x`, an atomic vector with `length(x) == n`,
- And `i`, *an expression that returns a vector of logical* values, with `length(i) == n`.
- Then, `x[i]` will return all of the elements in `x`, corresponding to the `TRUE` values in `i`.
- *In other words*, "Take every element in `x` where `i` is `T`"

Subsetting Vectors: use [

Example 6: "x" is the days of the week, "i" is a logical expression

Subsetting Vectors: use [

Example 6: "x" is the days of the week, "i" is a logical expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 6: "x" is the days of the week, "i" is a logical expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> nchar(dow) == 1  # logical expression
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

Subsetting Vectors: use [

Example 6: "x" is the days of the week, "i" is a logical expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> nchar(dow) == 1  # logical expression
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
> dow[nchar(dow) == 1]  # keep 1-letter days
```

Subsetting Vectors: use [

Example 6: "x" is the days of the week, "i" is a logical expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> nchar(dow) == 1  # logical expression
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
> dow[nchar(dow) == 1]  # keep 1-letter days
[1] "S" "M" "T" "W" "F"
```

Subsetting Vectors: use [

Example 6: "x" is the days of the week, "i" is a logical expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> nchar(dow) == 1  # logical expression
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
> dow[nchar(dow) == 1]  # keep 1-letter days
[1] "S" "M" "T" "W" "F"
```

Now we're getting somewhere. One more.

Subsetting Vectors: use [

How `[` works with an *expression* `i` that resolves to integer values.

- Given `x`, an atomic vector with `length(x) == n`,
- And `i`, *an expression that returns a vector of integers*, with `all(i <= n)`.
- Then, `x[i]` will return all of the elements in `x`, indexed by `i`.
- *In other words*, "return the `i`-th element(s) from `x`"

Subsetting Vectors: use [

Example 7: "x" is the days of the week, "i" is an integer expression

Subsetting Vectors: use [

Example 7: "x" is the days of the week, "i" is an integer expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

Example 7: "x" is the days of the week, "i" is an integer expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> grep('S', dow) # where can I find an S?
[1] 1 7
```

Subsetting Vectors: use [

Example 7: "x" is the days of the week, "i" is an integer expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> grep('S', dow) # where can I find an S?
[1] 1 7
> dow[grep('S', dow)]  # keep S-letter days
```

Subsetting Vectors: use [

Example 7: "x" is the days of the week, "i" is an integer expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> grep('S', dow) # where can I find an S?
[1] 1 7
> dow[grep('S', dow)]  # keep S-letter days
[1] "S"  "Sa"
```

Subsetting Vectors: use [

Example 7: "x" is the days of the week, "i" is an integer expression

```
> print(dow)
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
> grep('S', dow) # where can I find an S?
[1] 1 7
> dow[grep('S', dow)]  # keep S-letter days
[1] "S"  "Sa"
```

Wheel of Fortune.

Subsetting Vectors: use [

What happens when we do this?

```
> dow[TRUE]
```

Subsetting Vectors: use [

What happens when we do this?

```
> dow[TRUE]
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

Subsetting Vectors: use [

What happens when we do this?

```
> dow[TRUE]
```

```
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

What happens when we do this?

```
> dow[c(1:5, 5:1)]
```

Subsetting Vectors: use [

What happens when we do this?

```
> dow[TRUE]
```

```
[1] "S"  "M"  "T"  "W"  "Th" "F"  "Sa"
```

What happens when we do this?

```
> dow[c(1:5, 5:1)]
```

```
 [1] "S"  "M"  "T"  "W"  "Th" "Th" "W"  "T"  "M"  "S"
```

Subsetting Vectors: use [

How `[` works when `x` is a named vector.

- Given `x`, a named atomic vector with `length(x) == n`,
- And `j`, a character vector, with `all(j %in% names(x))`.
- Then, `x[j]` will return all of the elements in `x`, where `names(x) %in% j`.
- *In other words*, "return the element(s) named `j`"

Subsetting Vectors: use [

Example 8: "x" is the days of the week, with named elements, and "j" is a character expression

Subsetting Vectors: use [

Example 8: "x" is the days of the week, with named elements, and "j" is a character expression

```
> names(dow) <- c("Funday", "Motovation",
+                 "Transformation",
+                 "Wayback", "Throwback",
+                 "Friday", "Social")
```

Subsetting Vectors: use [

Example 8: "x" is the days of the week, with named elements, and "j" is a character expression

```
> names(dow) <- c("Funday", "Motovation",
+                 "Transformation",
+                 "Wayback", "Throwback",
+                 "Friday", "Social")

> str(dow)

 Named chr [1:7] "S" "M" "T" "W" "Th" "F" "Sa"
 - attr(*, "names")= chr [1:7] "Funday" "Motovation" "Trans
```

Subsetting Vectors: use [

Example 8: "x" is the days of the week, with named elements, and "j" is a character expression

```
> names(dow) <- c("Funday", "Motovation",
+                 "Transformation",
+                 "Wayback", "Throwback",
+                 "Friday", "Social")

> str(dow)

 Named chr [1:7] "S" "M" "T" "W" "Th" "F" "Sa"
 - attr(*, "names")= chr [1:7] "Funday" "Motovation" "Trans

> c("Wayback", "Throwback")
```

Subsetting Vectors: use [

Example 8: "x" is the days of the week, with named elements, and "j" is a character expression

```
> names(dow) <- c("Funday", "Motovation",
+                 "Transformation",
+                 "Wayback", "Throwback",
+                 "Friday", "Social")

> str(dow)

 Named chr [1:7] "S" "M" "T" "W" "Th" "F" "Sa"
 - attr(*, "names")= chr [1:7] "Funday" "Motovation" "Trans

> c("Wayback", "Throwback")

> dow[c("Wayback", "Throwback")]
```

Subsetting Vectors: use [

Example 8: "x" is the days of the week, with named elements, and "j" is a character expression

```
> names(dow) <- c("Funday", "Motovation",
+                 "Transformation",
+                 "Wayback", "Throwback",
+                 "Friday", "Social")

> str(dow)

 Named chr [1:7] "S" "M" "T" "W" "Th" "F" "Sa"
 - attr(*, "names")= chr [1:7] "Funday" "Motovation" "Trans

> c("Wayback", "Throwback")

> dow[c("Wayback", "Throwback")]

  Wayback Throwback
      "W"       "Th"
```

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Subsetting Lists: [[and $

How `[[` works - retrieve one element only, by name or by position

- Given `x`, a list with `length(x) == n`,
- And `j`, a *single* character value where `j %in% names(x)`,
- Or `j`, a *single* integer value where `j <= n`.
- Then, `x[[j]]` will return the *single* element in `x`, named or indexed by `j`.
- *In other words*, "return the element named `j` or the `j`th element"

Subsetting Lists: [[and $

```
> student <- list(universty = "Howard",
+                 major     = "Economics",
+                 GPA       = 3.2)
```

Subsetting Lists: [[and $

```
> student <- list(universty = "Howard",
+                 major     = "Economics",
+                 GPA       = 3.2)

> student[[3]]          # 3rd part of student object
```

Subsetting Lists: [[and $

```
> student <- list(universty = "Howard",
+                 major     = "Economics",
+                 GPA       = 3.2)

> student[[3]]          # 3rd part of student object

[1] 3.2
```

Subsetting Lists: [[and $

```
> student <- list(universty = "Howard",
+                 major     = "Economics",
+                 GPA       = 3.2)

> student[[3]]          # 3rd part of student object

[1] 3.2

> student[["GPA"]]      # GPA of student
```

Subsetting Lists: [[and $

```
> student <- list(universty = "Howard",
+                 major     = "Economics",
+                 GPA       = 3.2)

> student[[3]]          # 3rd part of student object

[1] 3.2

> student[["GPA"]]      # GPA of student

[1] 3.2
```

Subsetting Lists: [[and $

How $ works - retrieve one element only, by name

- Given `x`, a list with `length(x) == n`,
- And `j`, a *single* character value where `j %in% names(x)`.
- Then, `x$j` will return the *single* element in `x`, named `j`.
- *In other words*, "return the element named `j`"

Subsetting Lists: [[and $

```
> student$GPA          # student's GPA
```

Subsetting Lists: [[and $

```
> student$GPA          # student's GPA
[1] 3.2
```

R Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
*n*d	Array	-

Subsetting Data Frames: [, [[and $

All three subset operators work for data frames.

- $ – get a single column vector by name
- [[– get a single column vector by name or by position
- [– get a subset of rows and/or a subset of columns
  - retrieve rows by position or logical condition
  - select columns by name, position, or logical condition

Subsetting Data Frames: $

Recall the existing data frame:

```
> str(frb_chair)

'data.frame': 3 obs. of  3 variables:
 $ surname  : chr  "Yellen" "Bernanke" "Greenspan"
 $ appointed: num  2014 2006 1987
 $ has_beard: logi  FALSE TRUE FALSE

> print(frb_chair)

    surname appointed has_beard
1    Yellen       2014     FALSE
2  Bernanke       2006      TRUE
3 Greenspan       1987     FALSE
```

Subsetting Data Frames: $

Retrieve the $j-**th** column vector in the data frame, where `j` is the name of a column

```
> frb_chair$name
```

Subsetting Data Frames: $

Retrieve the $j-**th** column vector in the data frame, where `j` is the name of a column

```
> frb_chair$name
```

```
NULL
```

Subsetting Data Frames: [[

Retrieve the j-**th** column vector in the data frame, where `j` is the name of a column

```
> frb_chair[["year"]]
```

Subsetting Data Frames: [[

Retrieve the j-**th** column vector in the data frame, where `j` is the name of a column

```
> frb_chair[["year"]]
```

```
NULL
```

Subsetting Data Frames: [[

Retrieve the [[j-**th**]] column vector in the data frame, where `j` is the position of a column

```
> frb_chair[[3]]
```

Subsetting Data Frames: [[

Retrieve the [[j-**th**]] column vector in the data frame, where `j` is the position of a column

```
> frb_chair[[3]]
```

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

Subsetting Data Frames: [

Retrieve the `i`-**th** rows of the `j`-**th** columns in the data frame, where `i` and `j` are the positions of the rows and columns, respectively.

```
> frb_chair[2, 1]
```

Subsetting Data Frames: [

Retrieve the `i`-**th** rows of the `j`-**th** columns in the data frame, where `i` and `j` are the positions of the rows and columns, respectively.

```
> frb_chair[2, 1]
```

```
[1] "Bernanke"
```

Subsetting Data Frames: [

Retrieve the `i`-**th** rows of the `j`-**th** columns in the data frame, where `i` and `j` are vectors of positions for the rows and columns, respectively.

```
> frb_chair[c(2,3), c(1,3)]
```

Subsetting Data Frames: [

Retrieve the `i`-**th** rows of the `j`-**th** columns in the data frame, where `i` and `j` are vectors of positions for the rows and columns, respectively.

```
> frb_chair[c(2,3), c(1,3)]
```

```
    surname has_beard
2  Bernanke      TRUE
3 Greenspan     FALSE
```

Subsetting Data Frames: [

Retrieve the `j`-**th** columns in the data frame, `j` is a character vector with the names of the desired columns.

```
> frb_chair[, c("surname", "has_beard")]
```

Subsetting Data Frames: [

Retrieve the `j`-**th** columns in the data frame, `j` is a character vector with the names of the desired columns.

```
> frb_chair[, c("surname", "has_beard")]
```

```
    surname has_beard
1    Yellen     FALSE
2  Bernanke      TRUE
3 Greenspan     FALSE
```

Subsetting Data Frames: [

Retrieve the `i`-**th** rows in the data frame, where `i` is a logical vector with `TRUE` values at corresponding positions for the desired rows.

```
> frb_chair[ frb_chair$appointed >= 2000, ]
```

Subsetting Data Frames: [

Retrieve the `i`-**th** rows in the data frame, where `i` is a logical vector with `TRUE` values at corresponding positions for the desired rows.

```
> frb_chair[ frb_chair$appointed >= 2000, ]
```

```
   surname appointed has_beard
1   Yellen       2014     FALSE
2 Bernanke       2006      TRUE
```

Subsetting Data Frames: [

Mix and match!

```
> i <- frb_chair$has_beard == TRUE | frb_chair$appointed <
> j <- names(frb_chair)
> frb_chair[i, j]
```

Subsetting Data Frames: [

Mix and match!

```
> i <- frb_chair$has_beard == TRUE | frb_chair$appointed <
> j <- names(frb_chair)
> frb_chair[i, j]
```

```
    surname appointed has_beard
2  Bernanke       2006      TRUE
3 Greenspan       1987     FALSE
```

The subset Function: subset()

- Purpose: subset data frames
- Input: a data frame, a logical expression for rows (optional), and a numeric or character vector identifying columns (optional)[7]
- Output: a data frame, with optional subsets

Equivalent to the `[` operator. Subset a data frame based on a logical vector, `i`, for desired the rows and a character vector, `j`, for the desired variable names.

```
> subset(frb_chair, subset = i, select = j)
```

[7] See documentation `?subset`

Subsetting Data Frames: subset()

```
> subset(frb_chair,
+        subset = c(TRUE, TRUE, FALSE),
+        select = c("surname", "appointed"))
```

Subsetting Data Frames: subset()

```
> subset(frb_chair,
+        subset = c(TRUE, TRUE, FALSE),
+        select = c("surname", "appointed"))

   surname appointed
1   Yellen       2014
2 Bernanke       2006
```

Subsetting Data Frames: subset()

```
> subset(frb_chair,
+        frb_chair$has_beard == TRUE | frb_chair$appointed
```

Subsetting Data Frames: subset()

```
> subset(frb_chair,
+        frb_chair$has_beard == TRUE | frb_chair$appointed

   surname appointed has_beard
1   Yellen       2014     FALSE
2 Bernanke       2006      TRUE
```

Subsetting Data Frames: subset()

```
> i <- frb_chair$has_beard == TRUE | frb_chair$appointed <
> j <- names(frb_chair)
> subset(frb_chair, i, j)
```

Subsetting Data Frames: subset()

```
> i <- frb_chair$has_beard == TRUE | frb_chair$appointed <
> j <- names(frb_chair)
> subset(frb_chair, i, j)

    surname appointed has_beard
2  Bernanke       2006      TRUE
3 Greenspan       1987     FALSE
```

Iterate over vector elements

Recall the surname vector:

```
> surname
[1] "Yellen"    "Bernanke"  "Greenspan"
```

Iterate over vector elements

Recall the surname vector:

```
> surname
[1] "Yellen"    "Bernanke"  "Greenspan"
> for ( s in surname ) {
+     print(s)
+ }
```

Iterate over vector elements

Recall the surname vector:

```
> surname
[1] "Yellen"    "Bernanke"  "Greenspan"
> for ( s in surname ) {
+     print(s)
+ }
[1] "Yellen"
[1] "Bernanke"
[1] "Greenspan"
```

Iterate over vector indices

```
> for ( i in 1:length(surname) ) {
+     print(surname[i])
+ }
```

Iterate over vector indices

```
> for ( i in 1:length(surname) ) {
+     print(surname[i])
+ }
[1] "Yellen"
[1] "Bernanke"
[1] "Greenspan"
```

Nested Loops

Recall the `frb_chair` data frame:

```
> frb_chair
```

```
    surname appointed has_beard
1    Yellen       2014     FALSE
2  Bernanke       2006      TRUE
3 Greenspan       1987     FALSE
```

Nested Loops

```
> for ( i in 1:nrow(frb_chair) ) {
+     for ( j in 1:ncol(frb_chair) ) {
+         message <- paste(i, j, frb_chair[i, j])
+         print(message)
+     }
+ }
```

Nested Loops

```
[1] "1 1 Yellen"
[1] "1 2 2014"
[1] "1 3 FALSE"
[1] "2 1 Bernanke"
[1] "2 2 2006"
[1] "2 3 TRUE"
[1] "3 1 Greenspan"
[1] "3 2 1987"
[1] "3 3 FALSE"
```

Create a function with one parameter

A function is just an object with instructions in it. The instructions tell the function how to manipulate objects.

```
> f <- function(x) {
+     2*x
+ }
```

Create a function with one parameter

A function is just an object with instructions in it. The instructions tell the function how to manipulate objects.

```
> f <- function(x) {
+     2*x
+ }
```

What happens when we print the object f?

```
> f
```

Create a function with one parameter

A function is just an object with instructions in it. The instructions tell the function how to manipulate objects.

```
> f <- function(x) {
+     2*x
+ }
```

What happens when we print the object f?

```
> f
```

```
function(x) {
    2*x
}
```

...R prints the instructions embedded in `f`: *"I take one input object and make a private copy called x. I will multiply my x by 2, then return the result"*

Call a function with one argument

```
> x <- c(1, 2, 3)
> f(x)
```

Call a function with one argument

```
> x <- c(1, 2, 3)
> f(x)

[1] 2 4 6
```

Call a function with one argument

```
> x <- c(1, 2, 3)
> f(x)
```

```
[1] 2 4 6
```

What if we run it without arguments?

```
> x <- c(1, 2, 3)
> f()
```

Call a function with one argument

```
> x <- c(1, 2, 3)
> f(x)
```

```
[1] 2 4 6
```

What if we run it without arguments?

```
> x <- c(1, 2, 3)
> f()
```

```
> # Error in f() : argument "x" is missing, with no defaul
```

Why doesn't that work? Why didn't `f` see the `x`? It's right there!

Create a function with no parameters

```
> g <- function() {
+     2*x
+ }
```

Create a function with no parameters

```
> g <- function() {
+     2*x
+ }
```

What happens when we print the object g?

```
> g
```

Create a function with no parameters

```
> g <- function() {
+     2*x
+ }
```

What happens when we print the object g?

```
> g
```

```
function() {
    2*x
}
```

...R prints the instructions embedded in g: *"I will find an x somewhere. I will multiply it x by 2, then return the result"*

Call a function with no parameters

```
> x <- c(1, 2, 3)
> g(x)
```

Call a function with no parameters

```
> x <- c(1, 2, 3)
> g(x)

> # Error in g(x) : unused argument (x)
```

g doesn't know what it's supposed to do with that.

Call a function with no parameters

```
> x <- c(1, 2, 3)
> g(x)

> # Error in g(x) : unused argument (x)
```

g doesn't know what it's supposed to do with that.
What if we run g without arguments?

```
> x <- c(1, 2, 3)
> g()
```

Call a function with no parameters

```
> x <- c(1, 2, 3)
> g(x)

> # Error in g(x) : unused argument (x)
```

g doesn't know what it's supposed to do with that.
What if we run g without arguments?

```
> x <- c(1, 2, 3)
> g()

[1] 2 4 6
```

Why does that work? How did g know about the x?

Lets talk about {}

In R, {} group expressions or code blocks[8]. When used along with a function or a for loop they also imply a separate environment[9]

[8] Multiple lines of code
[9] memory space

Lets talk about {}

In R, {} group expressions or code blocks[8]. When used along with a function or a for loop they also imply a separate environment[9] What is the output here?

```
> x <- c(1, 2, 3)
> f <- function(x) {
+     y <- 1
+     x + y
+ }
> f(x)
> print(y)
```

[8] Multiple lines of code

[9] memory space

Lets talk about {}

In R, {} group expressions or code blocks[8]. When used along with a function or a for loop they also imply a separate environment[9] What is the output here?

```
> x <- c(1, 2, 3)
> f <- function(x) {
+     y <- 1
+     x + y
+ }
> f(x)
> print(y)
```

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

```
> # Error in print(y) : object 'y' not found
```

[8] Multiple lines of code

[9] memory space

Create a function with a named parameter, and a default value

Use a named parameter, with a default value

```
> g <- function(x = NA) {
+     2*x
+ }
```

g: *"I will assume my x is NA until given a value. I will multiply my x by 2, then return the result"*

Create a function with a named parameter, and a default value

Use a named parameter, with a default value

```
> g <- function(x = NA) {
+     2*x
+ }
```

g: *"I will assume my x is NA until given a value. I will multiply my x by 2, then return the result"* What if we run g without arguments?

```
> x <- c(1, 2, 3)
> g()
```

Create a function with a named parameter, and a default value

Use a named parameter, with a default value

```
> g <- function(x = NA) {
+     2*x
+ }
```

g: *"I will assume my x is NA until given a value. I will multiply my x by 2, then return the result"* What if we run g without arguments?

```
> x <- c(1, 2, 3)
> g()
```

```
[1] NA
```

g doesn't know about the `x` anymore. It has it's own parameter now.

Call a function with a named parameter

Use a named parameter, to refer to an environment variable

```
> y <- c(1, 2, 3)
> g(x = y)
```

Call a function with a named parameter

Use a named parameter, to refer to an environment variable

```
> y <- c(1, 2, 3)
> g(x = y)
```

```
[1] 2 4 6
```

g: *"I will copy the public y to my x. I will multiply my x by 2, then return the result"*

Call a function with a named parameter

Use a named parameter, to refer to an environment variable

```
> y <- c(1, 2, 3)
> g(x = y)
```

```
[1] 2 4 6
```

g: *"I will copy the public y to my x. I will multiply my x by 2, then return the result"*

```
> x <- c(1, 2, 3)
> g(y)
```

```
[1] 2 4 6
```

Same thing, the argument in the first position is copied to the internal x object

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }
```

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }

> f(x = 3, y = 2, z = 1)
```

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }

> f(x = 3, y = 2, z = 1)

[1] 1
```

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }
```

```
> f(x = 3, y = 2, z = 1)
```

```
[1] 1
```

Named parameters can be moved around

```
> f(x = 3, y = 2, z = 1) == f(z = 1, y = 2, x = 3)
```

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }
```

```
> f(x = 3, y = 2, z = 1)
```

```
[1] 1
```

Named parameters can be moved around

```
> f(x = 3, y = 2, z = 1) == f(z = 1, y = 2, x = 3)
```

```
[1] TRUE
```

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }
```

```
> f(x = 3, y = 2, z = 1)
```

```
[1] 1
```

Named parameters can be moved around

```
> f(x = 3, y = 2, z = 1) == f(z = 1, y = 2, x = 3)
```

```
[1] TRUE
```

Unnamed parameters cannot

```
> f(1, 2, 3) == f(3, 2, 1)
```

Create a function with multiple named parameters, and default values

```
> f <- function(x = NA, y = NA, z = NA) {
+     (x - y) * z
+ }
```

```
> f(x = 3, y = 2, z = 1)
```

```
[1] 1
```

Named parameters can be moved around

```
> f(x = 3, y = 2, z = 1) == f(z = 1, y = 2, x = 3)
```

```
[1] TRUE
```

Unnamed parameters cannot

```
> f(1, 2, 3) == f(3, 2, 1)
```

```
[1] FALSE
```

References

R

- http://adv-r.had.co.nz/
- http://tryr.codeschool.com/

Code Style

- https://google.github.io/styleguide/Rguide.xml
- http://adv-r.had.co.nz/Style.html