Introduction to R: **Vectors**

Day 1, Part B

In this lecture

1. Objects
2. Basic classes
3. Creating vectors
4. Testing/coercing classes
5. Missing values
6. Relational operators
7. Math on vectors
8. Summary statistics
9. Indexing
10. Factors
11. Matrices & arrays
12. Reassignment

Objects

Objects are how we store information (e.g., data, functions) in R.

Objects have three components:

1. Name
2. Value
3. Properties (class, dimension, etc.)

Objects

We create objects using an assignment operator (either <- or =) to assign a value to a name:

```
> my_office <- 424
> my_office
[1] 424
```

```
> my_office = 424
> my_office
[1] 424
```

Basic classes

Characters

Country iso3 codes: CAN, USA, MEX

Basic classes

Characters

Country iso3 codes: CAN, USA, MEX

Numerics

Population: 35.16, 318.9, 122.3

Basic classes

Characters

Country iso3 codes: CAN, USA, MEX

Numerics

Population: 35.16, 318.9, 122.3

Integers

Number of administrative units: 13, 51, 31

Basic classes

Characters

Country iso3 codes: CAN, USA, MEX

Numerics

Population: 35.16, 318.9, 122.3

Integers

Number of administrative units: 13, 51, 31

Logicals

Primary langugе is Spanish: FALSE, FALSE, TRUE

Vectors

A vector is an *ordered* collection of values that are all of the *same basic class*.

Vectors

A vector is an *ordered* collection of values that are all of the *same basic class*.

Vectors can be created manually using the combine function `c()`:

```
> iso3 <- c("CAN", "USA", "MEX")
> iso3
[1] "CAN" "USA" "MEX"
```

```
> pop <- c(35.16, 318.9, 122.3)
> pop
[1]  35.16 318.90 122.30
```

```
> admin1 <- c(13L, 51L, 31L)
> admin1
[1] 13 51 31
```

```
> spanish <- c(FALSE, FALSE, TRUE)
> spanish
[1] FALSE FALSE  TRUE
```

Creating vectors

There are lots of other functions which create vectors. Some useful ones:

`seq()` or `:` for numeric sequences:

```
> seq(from = 0, to = 27, by = 3)
 [1]  0  3  6  9 12 15 18 21 24 27
> 1990:2000
 [1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000
```

Creating vectors

There are lots of other functions which create vectors. Some useful ones:

`seq()` or `:` for numeric sequences:

```
> seq(from = 0, to = 27, by = 3)
 [1]  0  3  6  9 12 15 18 21 24 27
> 1990:2000
 [1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000
```

`rep()` for repeating elements of any class of vector:

```
> rep(c(1, 2, 3), each = 2)
[1] 1 1 2 2 3 3
> rep(c("red", "blue", "green"), times = 2)
[1] "red"   "blue"  "green" "red"   "blue"  "green"
```

Vector classes

We can find out the class of a vector using the `class()` command:

```
> class(iso3)
[1] "character"
> class(pop)
[1] "numeric"
> class(admin1)
[1] "integer"
> class(spanish)
[1] "logical"
```

Vector classes

We can also test to see if an object is of a particular class using `is.[class]()` commands:

```
> is.character(iso3)
[1] TRUE
> is.numeric(pop)
[1] TRUE
> is.integer(admin1)
[1] TRUE
> is.logical(spanish)
[1] TRUE
```

```
> is.numeric(iso3)
[1] FALSE
> is.integer(pop)
[1] FALSE
> is.numeric(admin1)
[1] TRUE
```

Coercion

In cases where it makes sense, it is possible to convert from one class of factors to another using `as.[class]()` commands:

```
> as.character(pop)
[1] "35.16" "318.9" "122.3"
> as.numeric(admin1)
[1] 13 51 31
> as.integer(spanish)
[1] 0 0 1
```

R often coerces vectors on the fly.

Missing values

R uses NA to denote missingness for all classes.

```
> rain <- c(F, F, F, F, NA, NA, NA)
```

Missing values

R uses `NA` to denote missingness for all classes.

```
> rain <- c(F, F, F, F, NA, NA, NA)
```

The function `is.na()` is used to test for missingness:

```
> is.na(rain)
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Missing values

R uses NA to denote missingness for all classes.

```
> rain <- c(F, F, F, F, NA, NA, NA)
```

The function is.na() is used to test for missingness:

```
> is.na(rain)
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Coercion of one vector type to another can cause missingness:

```
> as.numeric(iso3)
Warning: NAs introduced by coercion
[1] NA NA NA
```

Relational operators

You can make comparisons between a vector and a scalar using == (equal), != (not equal), > (greater than), and < (less than):

```
> iso3 == "USA"
[1] FALSE  TRUE FALSE
> pop > 300
[1] FALSE  TRUE FALSE
> admin1 < 20
[1]  TRUE FALSE FALSE
> spanish != TRUE
[1]  TRUE  TRUE FALSE
```

Relational operators

You can make comparisons between a vector and a scalar using == (equal), != (not equal), > (greater than), and < (less than):

```
> iso3 == "USA"
[1] FALSE  TRUE FALSE
> pop > 300
[1] FALSE  TRUE FALSE
> admin1 < 20
[1]  TRUE FALSE FALSE
> spanish != TRUE
[1]  TRUE  TRUE FALSE
```

You can also make comparisons between two vectors using these same operators:

```
> pop < admin1
[1] FALSE FALSE FALSE
```

Logical operators

The logical operators & (AND) and | (OR) can be used along with relational operators to create more complicated statements:

```
> iso3 == "USA" | iso3 == "CAN"
[1]  TRUE  TRUE FALSE
> admin1 > 20 & admin1 < 50
[1] FALSE FALSE  TRUE
> pop < 100 | pop > 300
[1]  TRUE  TRUE FALSE
```

Logical operators

The logical operators & (AND) and | (OR) can be used along with relational operators to create more complicated statements:

```
> iso3 == "USA" | iso3 == "CAN"
[1]  TRUE  TRUE FALSE
> admin1 > 20 & admin1 < 50
[1] FALSE FALSE  TRUE
> pop < 100 | pop > 300
[1]  TRUE  TRUE FALSE
```

The logical operator ! (NOT) negates or reverses any other logical statement

```
> pop < 200
[1]  TRUE FALSE  TRUE
> !(pop < 200)
[1] FALSE  TRUE FALSE
> !spanish
[1]  TRUE  TRUE FALSE
```

Recycling

If you try to make comparison between two vectors that are not the same length, R will ‘recycle’ the shorter vector:

```
> vec1 <- 1:6
> vec2 <- c(1, 10)
> vec1 > vec2
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

Recycling

If you try to make comparison between two vectors that are not the same length, R will ‘recycle’ the shorter vector:

```
> vec1 <- 1:6
> vec2 <- c(1, 10)
> vec1 > vec2
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```
> vec3 <- c(1, 10, 100, 1000)
> vec1 > vec3
Warning in vec1 > vec3: longer object length is not a
multiple of shorter object length
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

Vector math

All of the mathematical operators and functions we talked about earlier work element-wise on numeric (or integer) vectors:

```
> pop/admin1
[1] 2.704615 6.252941 3.945161
> log(pop)
[1] 3.559909 5.764878 4.806477
> sqrt(admin1)
[1] 3.605551 7.141428 5.567764
```

(note that recycling applies in this context too)

Summary statistics

R has many commands for calculating statistical summaries of data. For example:

```
> hi_temp <- c(77, 73, 75, 80, 79, 72, 72, 73, 72, 76, 80, 87,
+     90, 83, 84, 81, 80, 86, 95, 91, 73, 72, 79, 82, 94, 88, 72,
+     75, 81, 70, 71)
```

```
> mean(hi_temp)
[1] 79.45161
> sd(hi_temp)
[1] 7.159324
> quantile(hi_temp, c(0, 0.5, 1))
  0%  50% 100%
  70   79   95
```

Summary statistics

R has many commands for calculating statistical summaries of data. For example:

```
> hi_temp <- c(77, 73, 75, 80, 79, 72, 72, 73, 72, 76, 80, 87,
+     90, 83, 84, 81, 80, 86, 95, 91, 73, 72, 79, 82, 94, 88, 72,
+     75, 81, 70, 71)
```

```
> mean(hi_temp)
[1] 79.45161
> sd(hi_temp)
[1] 7.159324
> quantile(hi_temp, c(0, 0.5, 1))
  0%  50% 100%
  70   79   95
```

Some other useful commands: `var()`, `median()`, `min()`, `max()`, `range()`, `sum()`, `table()`.

Indexing by number

You can select one or more values from within a vector using the position number:

```
> hi_temp[1]
[1] 77
> hi_temp[c(1, 3, 10)]
[1] 77 75 76
> hi_temp[1:5]
[1] 77 73 75 80 79
> hi_temp[seq(1, 31, 7)]
[1] 77 73 84 72 81
> hi_temp[c(1, 1, 1)]
[1] 77 77 77
```

Indexing by logical

You can also select values from within a vector using logicals.

```
> iso3[c(FALSE, FALSE, TRUE)]
[1] "MEX"
> iso3[spanish == TRUE]
[1] "MEX"
> iso3[spanish]
[1] "MEX"
```

Indexing by logical

You can also select values from within a vector using logicals.

```
> iso3[c(FALSE, FALSE, TRUE)]
[1] "MEX"
> iso3[spanish == TRUE]
[1] "MEX"
> iso3[spanish]
[1] "MEX"
```

This is generally how you select values that meet a certain criteria:

```
> pop > 300
[1] FALSE  TRUE FALSE
> pop[pop > 300]
[1] 318.9
> iso3[pop > 300]
[1] "USA"
```

Indexing by logical

This also one way to find missing (or non missing) values:

```
> rain[is.na(rain)]
[1] NA NA NA
> rain[!is.na(rain)]
[1] FALSE FALSE FALSE FALSE
```

Indexing by logical

This also one way to find missing (or non missing) values:

```
> rain[is.na(rain)]
[1] NA NA NA
> rain[!is.na(rain)]
[1] FALSE FALSE FALSE FALSE
```

More complicated relational/logical statements can also be used:

```
> iso3[pop > 100 & admin1 < 50]
[1] "MEX"
```

Factors

Factors are a special type of vector that combine integers and characters. They are usually used for storing categorical data.

Factors can be created from a numeric vector by providing labels:

```
> sex <- c(1, 1, 2)
> sex
[1] 1 1 2
```

```
> factor_sex <- factor(sex, levels = c(1, 2), labels = c("Male",
+     "Female"))
> factor_sex
[1] Male   Male   Female
Levels: Male Female
```

Factors

Factors can also be created from character vectors:

```
> sex <- c("Male", "Male", "Female")
> sex
[1] "Male"   "Male"   "Female"
```

```
> factor_sex <- factor(sex, levels = c("Male", "Female"))
> factor_sex
[1] Male   Male   Female
Levels: Male Female
```

Factors

Factors can also be created from character vectors:

```
> sex <- c("Male", "Male", "Female")
> sex
[1] "Male"   "Male"   "Female"
```

```
> factor_sex <- factor(sex, levels = c("Male", "Female"))
> factor_sex
[1] Male   Male   Female
Levels: Male Female
```

Some helpful factor-related functions: `is.factor()`, `levels()`, `nlevels()`

Matrices

Matrices are essentially two-dimensional vectors.

They are usually created by assigning dimensions to a vector:

```
> mat <- matrix(1:6, nrow = 2, ncol = 3, byrow = T)
> mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Matrices

There are functions for testing if an object is a matrix, and for finding its dimensions:

```
> is.matrix(mat)
[1] TRUE
> nrow(mat)
[1] 2
> ncol(mat)
[1] 3
> dim(mat)
[1] 2 3
```

Matrices

There are functions for testing if an object is a matrix, and for finding its dimensions:

```
> is.matrix(mat)
[1] TRUE
> nrow(mat)
[1] 2
> ncol(mat)
[1] 3
> dim(mat)
[1] 2 3
```

And also functions for carrying out matrix algebra: `t()`, `solve()`, `det()`, `diag()`, `kronecker()`, `chol()`, etc.

Arrays

For more than 2 dimensions, R has arrays.

Like matrices, these are usually constructed by assigning a vector dimensions:

```
> ary <- array(1:24, dim = c(3, 4, 2))
> ary
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Reassignment

We've been carrying out operations on vectors and printing to screen:

```
> as.character(pop)
[1] "35.16" "318.9" "122.3"
```

This causes no permanent change to the object – R carries out the operation and prints the results, but doesn't change anything stored in memory:

```
> class(pop)
[1] "numeric"
> pop
[1]  35.16 318.90 122.30
```

Reassignment

To make a change permanent, we use one of the assignment operators to reassign the value of the object:

```
> pop <- as.character(pop)
> class(pop)
[1] "character"
> pop
[1] "35.16" "318.9" "122.3"
```

A quick note about object names...

Object names MUST start with an alphabetic character (a-z, A-Z). After the first character, objects names can contain alphabetic characters (a-z, A-Z), digits (0-9), underscores (_), and periods (.).

A quick note about object names...

Object names MUST start with an alphabetic character (a-z, A-Z). After the first character, objects names can contain alphabetic characters (a-z, A-Z), digits (0-9), underscores (_), and periods (.).

There are MANY conventions to choose from. Some common ones:

- `all_lower_with_underscores`
- `all.lower.with.periods`
- `CamelCase`

A quick note about object names. . .

Object names MUST start with an alphabetic character (a-z, A-Z). After the first character, objects names can contain alphabetic characters (a-z, A-Z), digits (0-9), underscores (_), and periods (.).

There are MANY conventions to choose from. Some common ones:

- `all_lower_with_underscores`
- `all.lower.with.periods`
- `CamelCase`

The key is consistency. And finding a balance between names that are short enough to type (repeatedly) quickly, but informative enough to be useful. (Some helpful advice on writing readable R code can be found here: http://adv-r.had.co.nz/Style.html)