Introduction to Spatial Data Programming - R

Lesson 02
Vectors

Michael Dorman
Geography and Environmental Development
dorman@post.bgu.ac.il

Last updated: 2018-12-20 17:01:14

Ben-Gurion University of the Negev

Contents

- Using R code files
- Vector types
- Operations with vectors
- Functions with more than one argument
- Subsetting
- Missing values

Aims

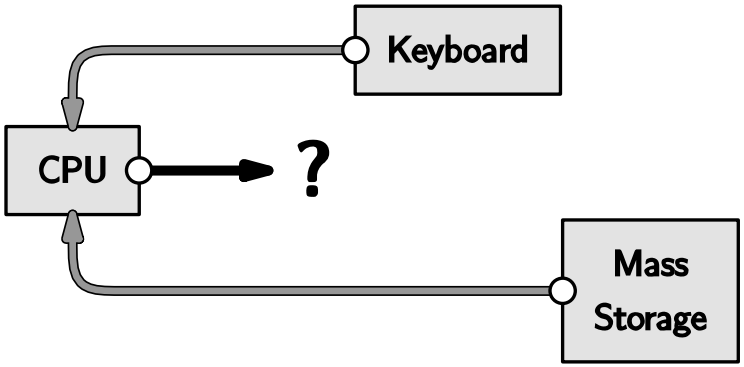
- Learning how to work with R code files
- Getting to know the simplest data structure in R, the vector
- Learning about subsetting, one of the fundamental operations with data

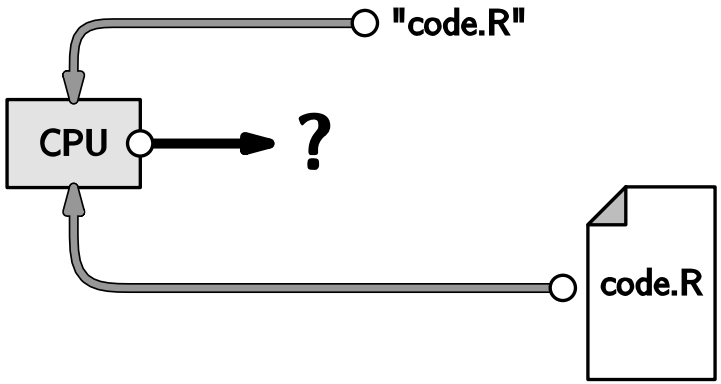
Using code files

- Computer code is stored as **plain text**
- When writing computer code, we must use a **plain text editor**, such as **Notepad++** or **RStudio**
- A **word processor**, such as **Microsoft Word**, is not a good choice for writing code, because -
  - Documents created with a Word Processor contain elements **other than** plain text (such as formatting), which are not processed, leading to confusion
  - Word processors can automatically **correct** "mistakes" thereby introducing unintended changes in our code, such as capitalizing: `max(1)` $\rightarrow$ `Max(1)`

Using code files

- Any plain text file can be used to store **R code**
- Conventionally, R code files have the `*.R` file **extension**
- **Complete** code files can be executed with `source`
- Selected **parts** of code can be executed by marking the section and pressing **Ctrl+Enter**
- Executing **one expression** can be done by standing inside it with the marker and pressing **Ctrl+Enter**





Using code files

- For example, we can run the file `volcano.R` which draws a 3D image of a volcano

```
source("volcano.R")
```

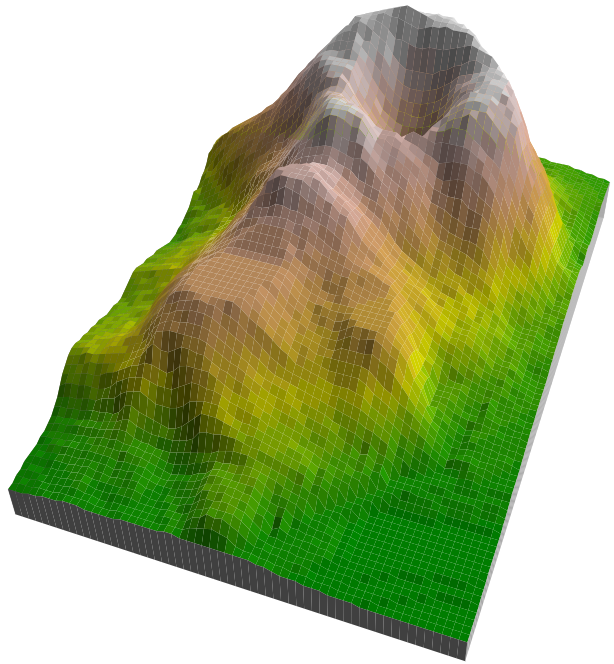


Figure 1: 3D image of the `volcano` dataset

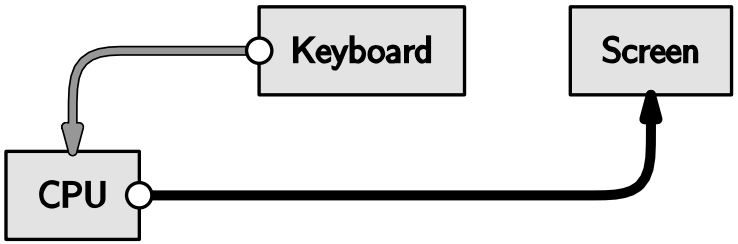
RStudio keyboard stortcuts

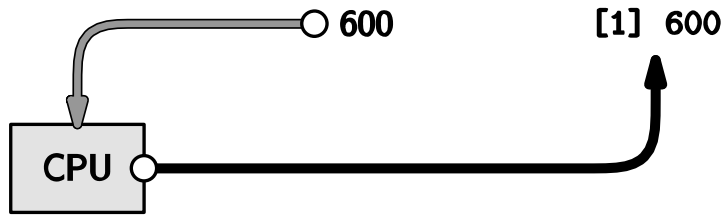
Table 1: RStudio keyboard shortcuts

Shortcut	Action
Alt+Shift+K	List of all shortcuts
Ctrl+1	Moving cursor to the code editor
Ctrl+2	Moving cursor to the console
Ctrl+Enter	Sending the current selection or line
Ctrl+Shift+P	Re-sending last selection
Ctrl+Alt+B	Sending from top to current line
Ctrl+Shift+C	Turn comment on or off
Tab	Auto-complete
Ctrl+D	Delete line
Ctrl+Shift+D	Duplicate line
Ctrl+F	Find and replace menu
Ctrl+S	Save

Assignment

- So far we have been using R by **typing** expressions into the command line and observing the result on screen
- That way R functions as a **"calculator"** - the results are not kept in computer memory



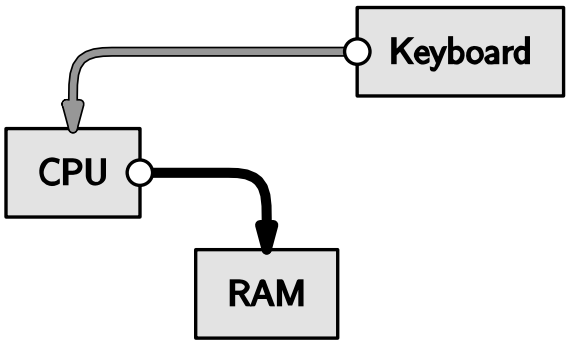


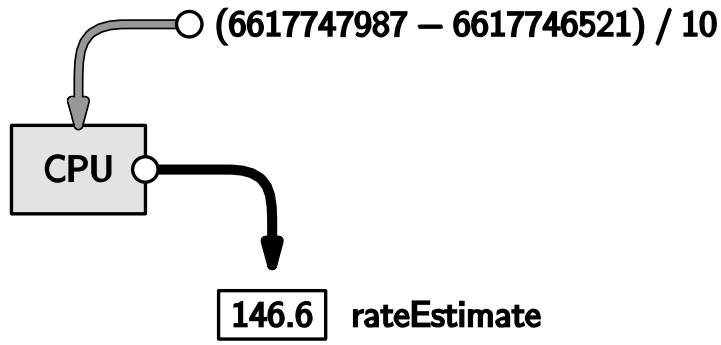
- Storing objects in the temporary computer memory (RAM) is called **assignment**
- Assignment is done using the **assignment operator**

Assignment

- In an **assignment expression** we are storing an object, under a certain name, in the RAM
- This is an essential operation in programming, because it make **automation** possible - reaching the goal step by step, while storing intermediate products
- An assignment expression consists of -
  - The **expression** whose result we want to store
  - The assignment **operator**, = or <-
  - The **name** which will be assigned to the object
- For example -

```
rateEstimate = (6617747987 - 6617746521) / 10
```



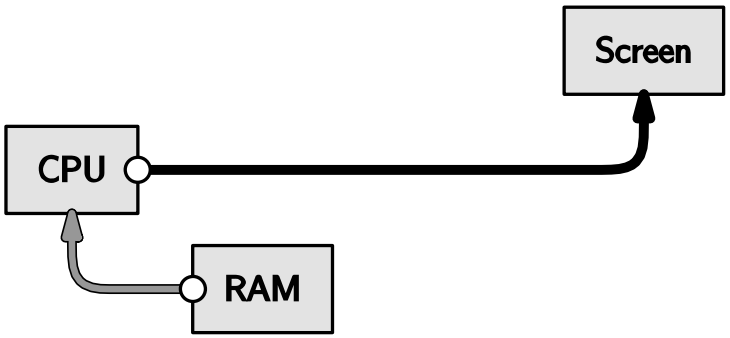


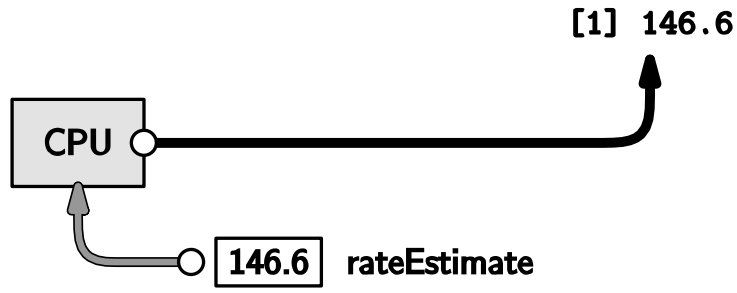
Assignment

- When we type an **object name** in the console, R **accesses** an object stored under that name in the RAM, and **calls** the `print` function on the object -

```
rateEstimate
## [1] 146.6
```

```
print(rateEstimate)
## [1] 146.6
```





Assignment

- What happens when we assign a new value to an **existing object**? The old object gets deleted, and its name is now pointing on the new value

```
x = 55
x
## [1] 55
```

```
x = "Hello"
x
## [1] "Hello"
```

Assignment

- Note the difference between the == and = operators!
- = is an **assignment** operator -

```
one = 1
two = 2
one = two
one
## [1] 2
two
## [1] 2
```

- == is a logical operator for **comparison** -

```
one = 1
two = 2
one == two
## [1] FALSE
```

Assignment

- Which user defined objects are currently **in memory**? The `ls` function returns a character vector with their names -

```
ls()
```

- Question: why do we write `ls()` and not `ls`?

Vector

- A vector is an **ordered** collection of values of the **same type**, such as -
  - **Numbers** - `interger` or `numeric`
  - **Text** - `character`
  - **Logical** - `logical`
- Recall that these are the same three types of constant values we saw in **Lesson 01**; in fact a constant value is a vector of length 1
- We can create an **empty vector** using the `vector` function, specifying -
  - `mode` - Type
  - `length` - Number of elements

```
v = vector(mode = "numeric", length = 10)
v
##  [1] 0 0 0 0 0 0 0 0 0 0
```

The c function

- Vectors can also be created with the c function, which **combines** the given vectors in the given order -

```
x = c(1, 2, 3)
x
## [1] 1 2 3
```

```
c(x, 5)
## [1] 1 2 3 5
```

- Another example, with `character` values -

```
y = c("cat", "dog", "mouse", "apple")
y
## [1] "cat"   "dog"   "mouse" "apple"
```

Vector subsetting

- We can access individual vector **elements** using the [operator and an index; in other words, to get a subset with an individual vector element -

```
y[2]
## [1] "dog"
```

```
y[3]
## [1] "mouse"
```

- Note: the index starts at 1!

Vector subsetting

- Another example -

```
counts = c(2, 0, 3, 1, 3, 2, 9, 0, 2, 1, 11, 2)
counts[4]
## [1] 1
```

- Components of an expression for accessing a vector element -

symbol:	counts[4]
square brackets:	counts[4]
index:	counts[4]

Vector subsetting

- We can also **assign** new values into a vector subset -

```
x = c(1, 2, 3)
x
## [1] 1 2 3
```

```
x[2] = 300
x
## [1]   1 300   3
```

- Note: in this example we made an assignment into a subset with a single element; the same way, we can make an assignment into a subset of any length (see below)

Calling functions on a vector

- There are various functions for calculating vector **properties** -

```
x = c(1, 6, 3, -8, 2)
```

```
length(x)   # Number of elements
## [1] 5
min(x)      # Minimum
## [1] -8
max(x)      # Maximum
## [1] 6
range(x)    # Minimum, maximum
## [1] -8  6
mean(x)     # Average
## [1] 0.8
sum(x)      # Sum
## [1] 4
```

Calling functions on a vector

- Other functions operate on **each** vector element, returning a vector of results having the same length as the input -

```
sqrt(x)
## [1] 1.000000 2.449490 1.732051      NaN 1.414214
```

```
abs(x)
## [1] 1 6 3 8 2
```

- Question: why does the result of the first expression contain `NaN`?

The recycling rule

- Binary operations applied on two vectors are done **element-by-element**, and a vector of the results is returned -

```
c(1, 2, 3) * c(10, 20, 30)
## [1] 10 40 90
```

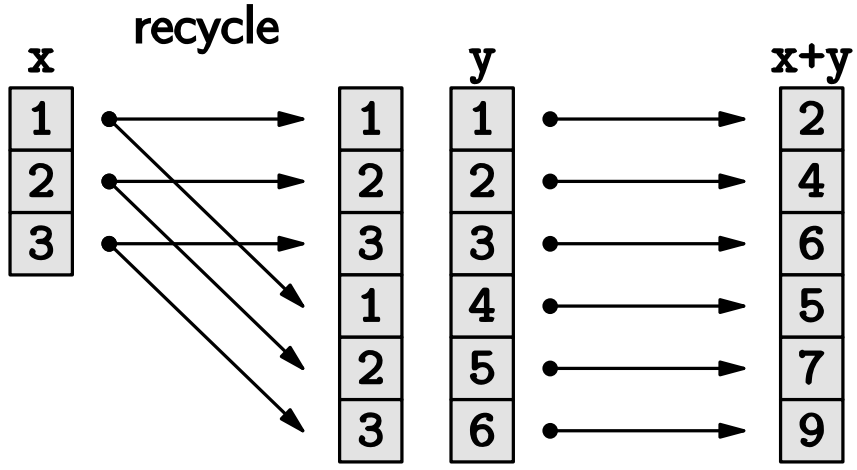
- What happens when the input vector lengths do not match? The shorter vector gets **"recycled"**

The recycling rule

- For example, when one of the vectors is of length 3 and the other vector is of length 6, then the sorter (of length 3) is repeated 2 times until it matches the longer vector -

```
c(1, 2, 3)           + c(1, 2, 3, 4, 5, 6)
## [1] 2 4 6 5 7 9
```

```
c(1, 2, 3, 1, 2, 3) + c(1, 2, 3, 4, 5, 6)
## [1] 2 4 6 5 7 9
```



The recycling rule

- When one of the vectors is of length 1 and the other is of length 4, the shorter vector (of length 1) is replicated 4 times -

```
2               * c(1, 2, 3, 4)
## [1] 2 4 6 8
```

```
c(2, 2, 2, 2) * c(1, 2, 3, 4)
## [1] 2 4 6 8
```

The recycling rule

- When one of the vectors is of length 2 and the other is of length 4, the shorter vector (of length 2) is replicated 2 times -

```
c(10, 100)           + c(1, 2, 3, 4)
## [1]  11 102  13 104
```

```
c(10, 100, 10, 100) + c(1, 2, 3, 4)
## [1]  11 102  13 104
```

The recycling rule

- When longer vector length is not a multiple of the shorter one recycling is **"incomplete"** and we get a warning message -

```
c(1, 2)    * c(1, 2, 3)
## Warning in c(1, 2) * c(1, 2, 3): longer object
## length is not a multiple of shorter object length
## [1] 1 4 3
```

```
c(1, 2, 1) * c(1, 2, 3)
## [1] 1 4 3
```

The recycling rule

- The recycling rule applies to many operators and functions, such as all arithmetic operators -

```
x = c(1, 2, 3)
y = c(4, 5, 6)
```

```
x + y
## [1] 5 7 9
```

```
x - y
## [1] -3 -3 -3
```

```
x * y
## [1]  4 10 18
```

```
x / y
## [1] 0.25 0.40 0.50
```

Consecutive and repetitive vectors

- Other than the `vector` and `c` functions, there are three commonly used methods for creating **consecutive** or **repetitive** vectors -
  - The `:` operator
  - The `seq` function
  - The `rep` function

Consecutive vectors

- The : operator is used to create a vector of consecutive vectors in steps of 1 or -1 -

```
1:10   # Steps of 1
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
55:43  # Steps of -1
##  [1] 55 54 53 52 51 50 49 48 47 46 45 44 43
```

Consecutive vectors

- The `seq` function creates a vector of consecutive values in any step size -
  - `from` - Where to start
  - `to` - When to end
  - `by` - Step size
- For example -

```
seq(from = 100, to = 150, by = 10)
## [1] 100 110 120 130 140 150
```

```
seq(from = 100, to = 80, by = -5)
## [1] 100  95  90  85  80
```

Function calls

- Using the `seq` function we will demonstrate three properties of function calls
- **First**, we can omit parameter names as long as the arguments are passed in the default order -

```
seq(from = 5, to = 10, by = 1)
## [1]  5  6  7  8  9 10
```

```
seq(5, 10, 1)
## [1]  5  6  7  8  9 10
```

Function calls

- **Second**, we can use any argument order as long as parameter names are specified -

```
seq(to = 10, by = 1, from = 5)
## [1]  5  6  7  8  9 10
```

```
seq(by = 1, from = 5, to = 10)
## [1]  5  6  7  8  9 10
```

```
seq(from = 5, by = 1, to = 10)
## [1]  5  6  7  8  9 10
```

Function calls

- **Third**, we can omit parameters that have a default value in the function definition -

```
seq(5, 10, 1)
## [1]  5  6  7  8  9 10
```

```
seq(5, 10)
## [1]  5  6  7  8  9 10
```

- To find out what are the parameters of a particular function, their order or their default values, we can look into the documentation -

```
?seq
```

Repetitive vectors

- The `rep` function **replicates** its argument to create a repetitive vector -
  - `x` - What to replicate
  - `times` - How many time to repeat `x`
  - `each` - How many times to repeat each element of `x`

```
rep(x = 22, times = 10)
##  [1] 22 22 22 22 22 22 22 22 22 22
```

```
rep(x = 22, each = 10)
##  [1] 22 22 22 22 22 22 22 22 22 22
```

Repetitive vectors

- The `x` argument can be a vector of **length >1**
- Note the difference between the `times` and the `each` **parameters** -

```
rep(x = c(18, 0, 9), times = 3)
## [1] 18  0  9 18  0  9 18  0  9
```

```
rep(x = c(18, 0, 9), each = 3)
## [1] 18 18 18  0  0  0  9  9  9
```

Vector subsetting

- So far we created vector subsets using a `numeric` index which consists of a **single** value, such as -

```
x = c(43, 85, 10)
x[3]
## [1] 10
```

- We can also use a vector of **length** $>$**1** as an index -

```
x[1:2]
## [1] 43 85
```

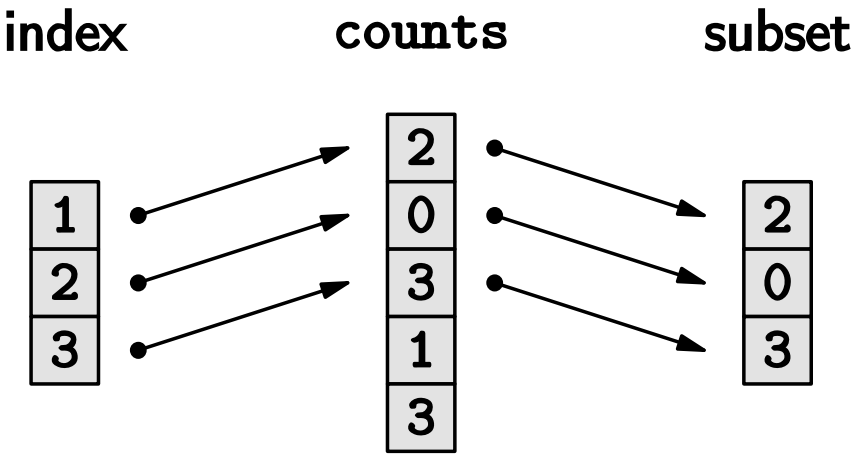
- Note that the vector does not need to be consecutive, and can include repetitions -

```
x[c(1, 1, 3, 2)]
## [1] 43 43 10 85
```

Vector subsetting

- Another example -

```
counts = c(2, 0, 3, 1, 3)
counts[1:3]
## [1] 2 0 3
```



Vector subsetting

- Another example -

```
counts = c(2, 0, 3, 1, 3, 2, 9, 0, 2, 1, 11, 2)
counts[c(1:3, 7:9)]
## [1] 2 0 3 9 0 2
```

symbol: `counts[c(1:3, 7:9)]`
square brackets: `counts[c(1:3, 7:9)]`
index: `counts[c(1:3, 7:9)]`

Vector subsetting

- For the next examples, let's create a vector of all **even** numbers between 1 and 100 -

```
x = seq(2, 100, 2)
x
##  [1]   2   4   6   8  10  12  14  16  18  20  22
## [12]  24  26  28  30  32  34  36  38  40  42  44
## [23]  46  48  50  52  54  56  58  60  62  64  66
## [34]  68  70  72  74  76  78  80  82  84  86  88
## [45]  90  92  94  96  98 100
```

- Question: what is the meaning of the numbers in square brackets when printing the vector?

Vector subsetting

- How many **elements** does `x` have?

```
length(x)
## [1] 50
```

- What is the value of the **last** element in `x`?

```
x[50]
## [1] 100
```

```
x[length(x)]
## [1] 100
```

- Question: which of the last two expressions is preferable and why?

Vector subsetting

- How can we get the **entire** vector using subsetting with a `numeric` index?

```
x[1:length(x)]
##  [1]   2   4   6   8  10  12  14  16  18  20  22
## [12]  24  26  28  30  32  34  36  38  40  42  44
## [23]  46  48  50  52  54  56  58  60  62  64  66
## [34]  68  70  72  74  76  78  80  82  84  86  88
## [45]  90  92  94  96  98 100
```

Vector subsetting

- How can we get the entire vector **except for** the last element?

```
x[1:(length(x)-1)]
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30
## [16] 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60
## [31] 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90
## [46] 92 94 96 98
```

- Question: how can we get a reversed vector using a `numeric` index?

Vector subsetting

- When requesting an index **beyond** vector length we get `NA` -

```
x[55]
## [1] NA
```

```
x[1:80]
##  [1]   2   4   6   8  10  12  14  16  18  20  22
## [12]  24  26  28  30  32  34  36  38  40  42  44
## [23]  46  48  50  52  54  56  58  60  62  64  66
## [34]  68  70  72  74  76  78  80  82  84  86  88
## [45]  90  92  94  96  98 100  NA  NA  NA  NA  NA
## [56]  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
## [67]  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
## [78]  NA  NA  NA
```

- Reminder: `NA` is a special value meaning *Not Available*

The recycling rule

- Earlier, we saw the recycling rule with **arithmetic operators**
- The rule also applies to **assignment** -

```
counts = c(2, 0, 3, 1, 3, 2, 9, 0, 2, 1, 11, 2)
counts[c(1:3, 7:9)]
## [1] 2 0 3 9 0 2
```

```
counts[c(1:3, 7:9)] = NA
counts
##  [1] NA NA NA  1  3  2 NA NA NA  1 11  2
```

```
counts[c(1:3, 7:9)] = c(NA, 99)
counts
##  [1] NA 99 NA  1  3  2 99 NA 99  1 11  2
```

Logical vectors

- The third common type of vectors are `logical` vectors
- A **logical vector** is composed of logical values: `TRUE` and `FALSE`
- For example -

```
c(TRUE, FALSE, FALSE)
## [1]  TRUE FALSE FALSE
```

```
rep(TRUE, 7)
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Logical vectors

- Usually, we will not be creating `logical` vectors **"manually"**, but through applying a **logical operator** on another vector -

```
x = 1:5
x
## [1] 1 2 3 4 5
```

```
x >= 3
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

- Note how the recycling rule applies to logical operators as well

Logical vectors

- When arithmetic operations are applied to a `logical` vector, the `logical` vector is **converted** to a numeric one, where `TRUE` becomes `1` and `FALSE` becomes `0`
- For example -

```
sum(x >= 3)
## [1] 3
```

```
mean(x >= 3)
## [1] 0.6
```

- Question: what is the meaning of the values `3` and `0.6` in the above example?

Logical vectors

- We can use a `logical` vector as an **index** for subsetting -

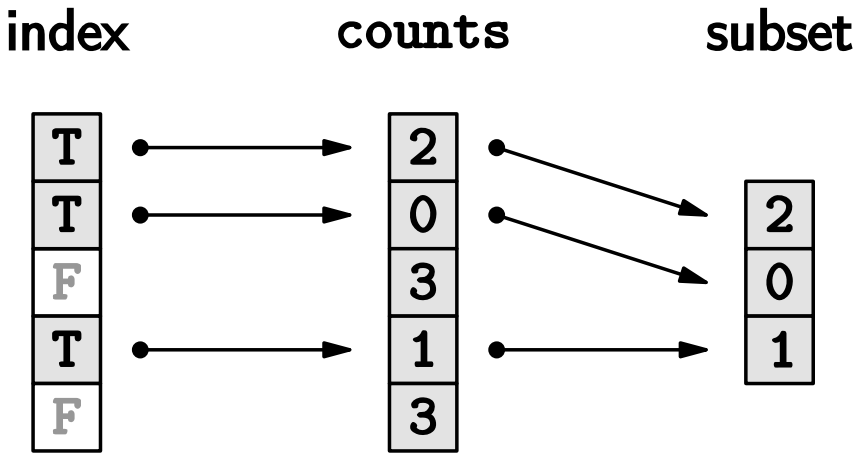
```
counts = c(2, 0, 3, 1, 3)
```

```
counts < 3
## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

```
counts[counts < 3]
## [1] 2 0 1
```

Logical vectors

- The logical vector `counts<3` specifies whether **to include** each of the elements of `counts` in the resulting subset



Logical vectors

- Using a `logical` index -

```
x = 1:5
```

```
x[x >= 3]
## [1] 3 4 5
```

```
x[x != 2]
## [1] 1 3 4 5
```

```
x[x > 4 | x < 2]
## [1] 1 5
```

```
x[x > 4 & x < 2]
## integer(0)
```

- Question: what happened in the last expression?

Logical vectors

- The next example is slightly more complex: we select the elements of `z` whose square is larger than 8 -

```
z = c(5, 2, -3, 8)
z[z^2 > 8]
## [1]  5 -3  8
```

- `z^2` gives a vector of squared `z` values (2 is recycled)

```
z^2
## [1] 25  4  9 64
```

- Each of the squares is compared to 8 (8 is recycled)

```
z^2 > 8
## [1]  TRUE FALSE  TRUE  TRUE
```

- Finally, the latter `logical` vector is used for subsetting `z`

Missing values

- The `is.na` function is used to detect **missing** (`NA`) values in a vector -
  - Accepts a vector of **any type**
  - Returns a **logical** vector with `TRUE` in place of `NA` values and `FALSE` in place of non-`NA` values
- For example -

```
x = c(28, 58, NA, 31, 39, NA, 9)
x
## [1] 28 58 NA 31 39 NA  9
```

```
is.na(x)
## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

Missing values

- Many functions that **summarize** vector properties, such as `mean`, have the `na.rm` parameter
- The `na.rm` parameter can be used to **exclude** `NA` values from the calculation
- The default is `na.rm=FALSE`, meaning that `NA` values are **not** excluded
- For example -

```
x = c(28, 58, NA, 31, 39, NA, 9)
```

```
mean(x)
## [1] NA
mean(x, na.rm = TRUE)
## [1] 33
```

- Question 1: why do we get `NA` in the first expression?
- Question 2: what will be the result of `length(x)`?

Missing values

```
x = c(28, 58, NA, 31, 39, NA, 9)
x
## [1] 28 58 NA 31 39 NA  9
```

- Question: how can we replace the `NA` values in the above vector with the mean of its non-`NA` values?

The `any` and `all` functions

- Sometimes we want to figure out whether -
  - A `logical` vector contains **at least one** `TRUE` value
  - A `logical` vector is **entirely** composed of `TRUE` values
- We can use the `any` and `all` functions, respectively, for that

The any and all functions

- The any function returns TRUE if **at least one** of the input vector values is TRUE, otherwise it returns FALSE -

```
x = 1:7
x
## [1] 1 2 3 4 5 6 7
```

```
x > 5
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
any(x > 5)
## [1] TRUE
```

```
x > 88
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
any(x > 88)
## [1] FALSE
```

The any and all functions

- The `all` function returns `TRUE` if **all** of the input vector values are `TRUE`, otherwise it returns `FALSE` -

```
x = 1:7
x
## [1] 1 2 3 4 5 6 7
```

```
x > 5
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
all(x > 5)
## [1] FALSE
```

```
x > 0
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
all(x > 0)
## [1] TRUE
```

The `which` function

- The `which` function **converts** a `logical` vector to a `numeric` one with the indices of `TRUE` values
- That way we can find out the **index** of values that satisfy a given criterion
- For example -

```
x = c(2, 6, 2, 3, 0, 1)
x
## [1] 2 6 2 3 0 1
```

```
x > 2.3
## [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
```

```
which(x > 2.3)
## [1] 2 4
```

The `which.min` and `which.max` functions

- Related functions `which.min` and `which.max` return the **index** of the (first) **minimal** or **maximal** value in a vector
- For example -

```
x = c(2, 6, 2, 3, 0, 1, 6)
x
## [1] 2 6 2 3 0 1 6
```

```
which.min(x)
## [1] 5
```

```
which.max(x)
## [1] 2
```

- Question: how can we get **all** indices of the minimal or maximal value?

The order function

- The order function returns **ordered** vector **indices**, based on the order of vector **values**
- In other words, order gives the index of the smallest value, the index of the second smallest value, and so on

```
x = c(2, 6, 2, 3, 0, 1, 6)
x
## [1] 2 6 2 3 0 1 6
```

```
order(x)
## [1] 5 6 1 3 4 2 7
```

- We can also get the **reverse** order with decreasing=TRUE -

```
order(x, decreasing = TRUE)
## [1] 2 7 4 1 3 6 5
```

- Question: how can we *sort* a vector using order?

The paste function

- The paste function is used to **"paste"** text values
- Its sep parameter determines the separating character(s), with default sep=" "

```
paste("There are", "5", "books.")
## [1] "There are 5 books."
paste("There are", "5", "books.", sep = "_")
## [1] "There are_5_books."
```

- Non-character vectors are converted to character before pasting -

```
paste("There are", 80, "books.")
## [1] "There are 80 books."
```

The paste and paste0 functions

- The recycling rule applies in paste too -

```
paste("image", 1:5, ".tif", sep = "")
## [1] "image1.tif" "image2.tif" "image3.tif"
## [4] "image4.tif" "image5.tif"
```

- The paste0 function is a shortcut for paste with sep="" -

```
paste0("image", 1:5, ".tif")
## [1] "image1.tif" "image2.tif" "image3.tif"
## [4] "image4.tif" "image5.tif"
```

Exercise 1 - Submission date **2018-11-25**

Introduction to Spatial Data Programming

Execrise 1

The R environment & Vectors

Last updated: 2018-11-09 17:11:36

Question 1

- On R start-up, several sample datasets are loaded
- Two of them are -
  - `state.name` - Names of US states
  - `state.area` - Corresponding state area size, in square Miles
- **Create** a vector of state area sizes in square kilometers
- **Print** a subset of state names for states whose areas are larger than 400,000 square kilometers
- Note: you need to convert from square miles to square kilometers

```
## [1] "Alaska"     "California" "Texas"
```

(50 points)

Question 2

- Use the `state.name` and `state.area` vectors and write an expression that **prints** the name of the state which has the smallest area in the US
- Note: do not use the values `39` or `"Rhode Island"` your the expression

```
## [1] "Rhode Island"
```