Data Types, Data Structures, and Subsetting

STAT 385 - James Balamuta

June 16, 2016

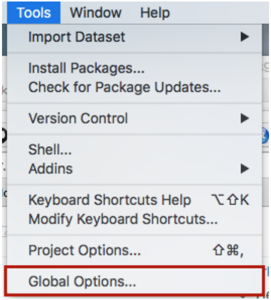
Overview

Today's Goals are:

1. Last minute RStudio tweaks
2. Writing Good Code
3. Data Types in R
4. Data Structures (Vectors)
5. Subsetting

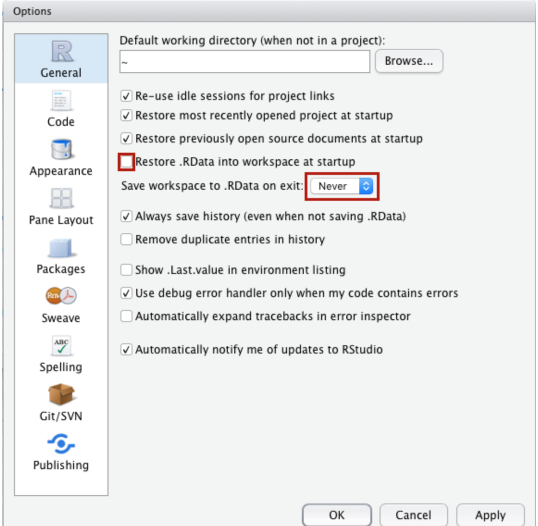
Preparing your environment

Navigate to Tools $\Rightarrow$ Global Options



Preparing your environment

- Uncheck `Restore .RData` into workspace at startup
- For `Save workspace to .RData on exit`, select the `Never` option from the dropdown.

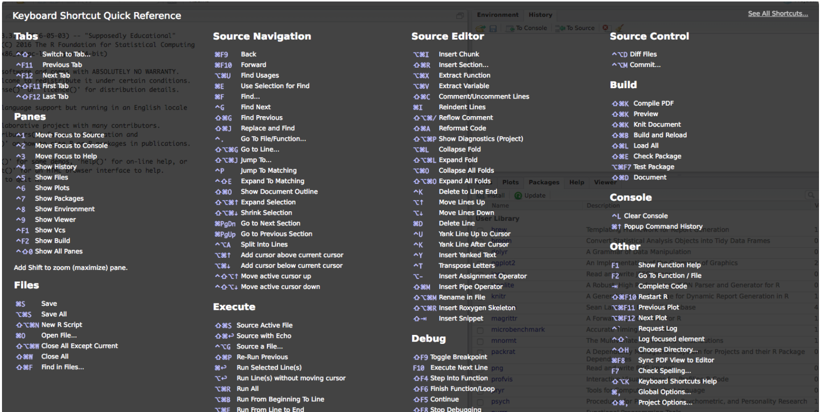


Know thine RStudio Environment

There are a *lot* of keyboard shortcuts in RStudio. These shortcuts are meant to speed up your work.

To view all the options, you must engage the keyboard shortcut that rules them all:

- Windows: `Alt` + `Shift` + `K`
- macOS: `Option` + `Shift` + `K`



My Favorites

1. Runs the current line and/or current selection from the editor to the console and runs it
   - Windows: `Ctrl` + `Enter`
   - macOS: `Cmd` + `Enter`
2. Comment multiple lines.
   - Windows: `Ctrl` + `Shift` + `C`
   - macOS: `Command` + `Shift` + `C`
3. Multicursor:
   - Windows: `Ctrl` + `Alt` + `Up` (or `Down`)
   - macOS: `Ctrl` + `Alt` + `Up` (or `Down`)
4. Reindent Code:
   - Windows: `Ctrl` + `L`
   - macOS: `Command` + `I`
5. Autocomplete command
   - Both: `Tab`

Writing Code

Good artists copy; great artists steal.
— Steve Jobs (quoted to Picaso but really T.S Elliot)

- To some degree, we will reinvent the wheel.
- In other cases, you may not.

Writing Code

Somethings to know about:

- All R code is immediately accessible by just typing the function name.

```
isTRUE
```

```
## function (x)
## identical(TRUE, x)
## <bytecode: 0x7fef8b846b90>
## <environment: namespace:base>
```

- Sometimes, we may need to pry into R using:
  - The `pryr` package's `fun_body()`
  - R's `getAnywhere()` or `methods()`
- GitHub has a fantastic search engine for code chunks.
  - Search via org:cran using either the name of a function or idea.

Data Types

To a computer, each *variable* must have a specific kind of *data type*.

Definition:

Data type is a description that indicates the type of data that can object can hold.

It is important that the data are matched with the apporiate type.

Supported Data Types

The different types of data supported by `R` are as follows:

- Numeric (double/float)
  - Examples: `-2`, `0.0`, `6.1`, `41.234`
- Integer
  - Examples: `-2L`,`0L`, `3L`, `10L`
- Complex
  - Examples: `-1 + 2i`, `0 + 0i`, `1 - 2i`
- Logical (boolean)
  - Examples: `TRUE` (`T`) and `FALSE` (`F`)
  - As a side note: `NA` (missing value) is also considered logical.
- Character
  - Examples: `"Hello"`, `"World"`, `"of Statistics"`, `"1 + 1"`
- Factor
  - Example: `Levels: 'May','Jun','Jul'`
- Ordered Factors:
  - Example: `Levels: 1 < 4 < 6`

Storing Information in a Variable

- While using R, you may wish to call a calculation at a later time. In such cases, it is ideal to store the computation in a **variable**.
- You do *not* need to specify the variables data type in advance as R will handle that for you. This is good and bad for various reasons that will be covered next under **coercion**.

Sample Assignments:

```r
a = 1      # Assign 1 to `a`
b = 2      # Assign 2 to `b`

d = a + b # Assign the sum of `a` and `b` to `d`.
```

Storing Missingness in a Variable

> *An NA is the presence of an absence. Don't forget that some missing values are the absence of a presence*
> *— Hadley Wickham twitter*

Within R, there is a specific type that handles "missingness". The type is `NA`.

Storing Missingness in a Variable

There are `NA` values for many data types. However, all you will really need is `NA`.

```r
NA              # Logical
```

```
## [1] NA
```

```r
NA_integer_     # Integer
```

```
## [1] NA
```

```r
NA_real_        # Double
```

```
## [1] NA
```

```r
NA_character_ # Character
```

```
## [1] NA
```

Built-in constants

R has a few pre-defined variables that will make life easier.
As a result, you no longer have to google: order of the alphabet.

```
LETTERS      # Uppercase alphabet
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
letters      # Lowercase alphabet
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Built-in constants

```
month.abb   # Abbreviated Month Name
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Se
## [12] "Dec"
```

```
month.name  # Full Month Name
```

```
##  [1] "January"   "February"  "March"     "April"     "Ma
##  [6] "June"      "July"      "August"    "September" "Oc
## [11] "November"  "December"
```

```
pi          # Pi
```

```
## [1] 3.141593
```

Be Warned of the Redefine

You **can** overide these variables on a per session basis.
So, be **careful** when using them. e.g.

```
pi          # Initial Value
```

```
## [1] 3.141593
```

```
pi = 3.14 # Modified the equation
pi          # View new value
```

```
## [1] 3.14
```

Keep this in mind during later when we work on debugging code. . . *wink*.

Preview Value during Assignment

Previously, we opted to assign and then output the variable contents.
e.g.

```r
life = 42
life
```

```
## [1] 42
```

By enclosing the assignment within paranthesis, e.g. `()`, we can omit one line:

```r
(life = 42)
```

```
## [1] 42
```

Style Guide

- When writing code, it is important, especisally in large organizations, to have a **consistent** style.
- Did you know, UIUC has its own identity standards?
- For instance, to use UIUC's I-logo, university personnel and vendors must adhere to the correct usage case:

Correct	Incorrect

Style Guide

- In that vein, organizations, like Google, have created internal style guides for code.
- Now, style guides are not per say the *best* practices to use end of discussion.
- Instead, they serve to unify the code written by a bunch of different individuals.

Class Style Guide

- For the most part, we will follow Google's style advice, which is also used by Hadley Wickham.
- The main exception to this principle is the avoidance of using the -> and <- assignment operators outside of piping (more later). Of course, you can also switch this by using formatR by Yihui Xie (knitr author).

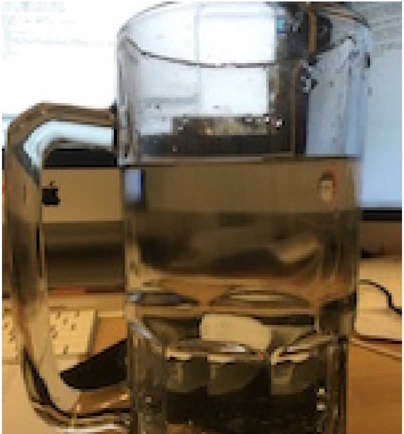
```
x = 1  # Good

x <- 1 # Bad
```

Heterogeneity and Homogenous Data Structures

R is **unique** in that it provides heterogeneous structures that enable data types to be mixed.

However, there are certain data types that must homogeneous or of the same type.

Heterogeneous	Homogeneous
	

Heterogeneity vs. Homogenous Data Structures

The different data structures are as follows:

Dimensions	Homogeneous	Heterogeneous
1d	`atomic vector`	`list`
2d	`matrix`	`data.frame`
nd	`array`	

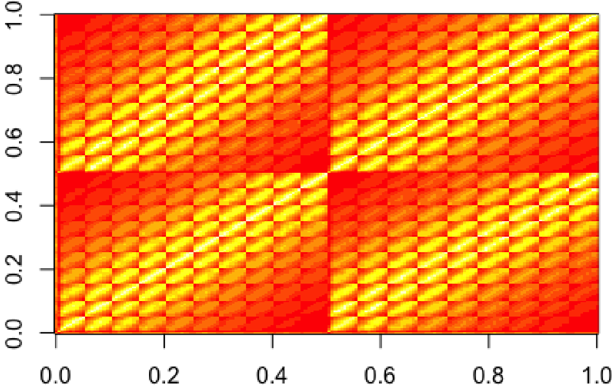
For the moment, we are going to focus on `vector` structures and `homogeneous` data.

Questions:

1. When might we need to use `n`-dimensions of data?
2. What happens if we mix one data type with another?
3. Which data structure could potentially rule them all? (e.g. be the parent)

3D Data Examples

Strength of the Relationship



3D Data Examples

Viewing Purchase decision of Customers by Part and Store.

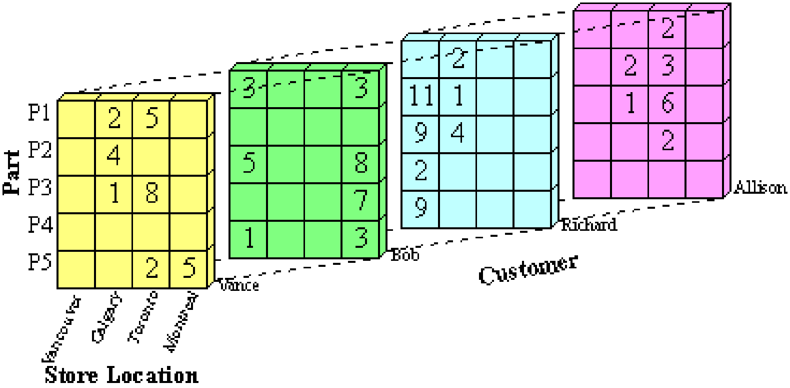


Figure 3: Cube Traits

Atomic Vectors

The majority of work done in R uses **atomic vectors** as building blocks.
To create an empty atomic vector we can use:

```r
a = numeric()   # Numeric
b = integer()   # Integer
d = character() # String
e = complex()   # Complex Number
f = logical()   # Boolean
g = factor()    # Factor
```

This creates an atomic vector of length 0 of a specific type.

```r
length(a) # Number of elements contained in the vector
```

```
## [1] 0
```

Atomic Vectors

`**Notes:**`

1. We can only store **one** specific data type per atomic vector (hence, homogeneous).
2. The length of `0` is only problematic in the case of a `factor`.

Vectors Initialization

To create a vector of length `n`, simply:

```
n = 20L            # Store a number
a = numeric(n)     # Create a double
a                  # View entries
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
length(a)          # Verify the length
```

```
## [1] 20
```

Vectors Initialization

Alternatively, if the values for the vector are already known, the vector can be created using:

```
(a2 = c(-1,2,4,5,1,6,41,31,23))
```

```
## [1] -1  2  4  5  1  6 41 31 23
```

```
(b2 = c(1L,2L,3L))
```

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

Notes:

- `c()` can also be used to concatenate different (add together) objects
- Avoid naming functions or variables with `c`!

Atomic Vector Depth

Atomic vectors must always be *flat*.
That is, if you nest different atomic vectors with concatenation, `c()`, the resulting atomic vector should always have dimension 1:

```r
c(1, c(2, c(3, 4)))  # Nested concatenation
```

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

```r
c(1, 2, 3, 4)        # Traditional construction
```

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

Atomic Vector Properties

Each atomic vector has its own properties, in the case of `a`, we have:

```r
typeof(a)          # Determine the type of `a`
```

```
## [1] "double"
```

```r
typeof(b2)         # Determine the type of `b2`
```

```
## [1] "integer"
```

Atomic Vector Properties

Attributes, which will cover in depth later, can be viewed as a way to add additional data or metadata.

```
attributes(a)     # Access metadata of `a`
```

```
## NULL
```

Note: The initial vector does not have any other attributes associated with it. We can add some using:

```
# Set metadata of `a` to include parameter sample with val
attr(a,"sample") = "Statistics"
attributes(a)      # Access metadata of `a`
```

```
## $sample
## [1] "Statistics"
```

Identifying an Atomic Vector

To identify the type of an atomic vector, you can create your own check statement or use a built-in function:

```
is.character(letters) # Checks for characters
is.double(1.2:4.4)    # Check for doubles
is.integer(1L:4L)     # Checks for integers
is.logical(c(T,F))    # Checks for booleans
is.atomic(1:4)        # Checks for atomic vector
```

Note: Do not use: `is.vector()`! You will be disappointed in the results.

Mixing Homogeneous Types

As hinted to earlier, it is not ideal to mix homogenous types.
Consider two atomic vectors:

```
a = runif(5)       # Five random numbers from within [0,1]
b = letters[1:5]   # First 5 letters of the alphabet
```

The structure of these vectors are:

```
str(a)             # View the structure of `a`
```

```
##  num [1:5] 0.3321 0.1702 0.2327 0.0687 0.6898
```

```
str(b)             # View the structure of `b`
```

```
##  chr [1:5] "a" "b" "c" "d" "e"
```

Mixing Homogeneous Types

If we merge them, we get:

```r
d = c(a, b)          # Concatenate `a` and `b`

str(d)               # View the structure of `d`
```

```
##  chr [1:10] "0.332097182050347" "0.170185891445726" ...
```

Notes: - Merging with `characters` yields `numeric`

Converting Between Homogeneous Types

Sometimes, you may have data that has been given to you in `character` form but really should be `numeric` and vice versa. To move between formats you can use:

```
(x = c(TRUE, FALSE, FALSE, TRUE))
```

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

```
as.numeric(x)
```

```
## [1] 1 0 0 1
```

Atomic Vector Access

We can access each element using the [operator.

```r
a[1]      # Access and print first element
```

```
## [1] 0.3320972
```

```r
a[1] = 2 # Accesss and assign new value to first element
a[1]      # Access and print first element
```

```
## [1] 2
```

Note: All vectors in *R* start with index 1.

Naming Atomic Vector Values

Sometimes, it might be helpful to label what each value in an Atomic vector is apart of:
e.g.

```
x = c("sphynx" = -1, "calypso" = 2, "doomsday" = 0,
      "life" = 42, "nine" = 9)
```

As a result, each component can be referred to by a name:

```
x[c("doomsday", "life")]
```

```
## doomsday      life
##         0        42
```

Naming Atomic Vector Values

All of the names within the vector can be known with:

```r
names(x)
```

```
## [1] "sphynx"    "calypso"   "doomsday" "life"      "nine"
```

Subsetting with Vectors

Sometimes, we may wish to only look at a specific piece of data. We can get elements at specific positions with:

```
x[c(2, 4)]
```

```
## calypso    life
##       2      42
```

Or, we can remove elements at specific positions with:

```
x[-c(2, 4)]
```

```
##    sphynx doomsday      nine
##        -1        0         9
```

Subsetting with Vectors

Notes:

1. Indexes must be all positive or all negative.
2. You cannot use names to remove an element.

```
x[-c("doomsday", "life")]
# Error in -c("doomsday", "life") :
# invalid argument to unary operator
```

Edge Subsetting Cases

If an index is not included, then the entire vector will be displayed

```
x[]    # All terms
```

```
##    sphynx   calypso doomsday      life      nine
##        -1         2        0        42         9
```

If you specify 0, there will be an empty vector:

```
x[0]   # Empty Vector
```

```
## named numeric(0)
```

Edge Subsetting Cases

If the element is out of bounds, then you will receive an `NA` vector

```
x[9]  # Only one NA returned
```

```
## <NA>
##   NA
```

```
x[NA] # All terms are NA
```

```
## <NA> <NA> <NA> <NA> <NA>
##   NA   NA   NA   NA   NA
```

Changing Element order withing Atomic Vectors

More often then not, you will want to know the progression of elements either in an increasing or decreasing form.
To do so, use:

```
x[order(x)]                        # Ascending Order
```

```
##    sphynx doomsday  calypso     nine     life
##        -1        0        2        9       42
```

```
x[order(x, decreasing = T)]    # Descending Order
```

```
##      life     nine  calypso doomsday   sphynx
##        42        9        2        0       -1
```

Changing Element order withing Atomic Vectors

Alternatively, you can sort the vector

```r
sort(x)
```

```
##    sphynx doomsday  calypso      nine      life
##        -1        0        2         9        42
```

Shortcuts

Vectors have many convenient short cuts.

1. `x:y` operator allows for the generation of an integer vector.
   - `x = 1:5` generates a vector of length 5 that contains $1, 2, 3, \ldots, 5$ and assigns it to `x`.
2. `*x`,`/x`,`+x` operators allow the vector to be modified by a term.

Shortcuts

```
2 * x         # Multiply all values by two
```

```
## [1]  2  4  6  8 10
```

```
x / 3         # Divide all values by three
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667
```

```
x + 1         # Add one to all values.
```

```
## [1] 2 3 4 5 6
```

```
2 * (x - 1) # Subtract one from all values and then multip
```

```
## [1] 0 2 4 6 8
```

Summing an atomic vector

There are two ways we can go about adding up the contents within an atomic vector.
We can use a loop:

```
x = 1:5                       # Create initial vector
sumx = 0                      # Create a sum value

for(i in seq_along(x)){  # Create an index vector
   sumx = sumx + x[i]    # Access each [i] and sum over it
}
```

Summing an atomic vector

Or, we can use a vectorized function:

```
sumx_v2 = sum(x)           # Use a vectorized calculation
```

We will need to verify if the calculation is the same:

```
all.equal(sumx, sumx_v2) # Verify equality
```

```
## [1] TRUE
```

Other helpful shortcuts

- `rep()` function provides a way to replicate values throughout the vector.

```r
# Generates a vector of length `5` that contains only 1
rep(1,5)
```

```
## [1] 1 1 1 1 1
```

Other helpful shortcuts

- `seq()` function provides a way to create a sequence of values.
  - **Note:** This approach is considerably slower than using `1:5` due to the methods genericness.

```
# Generates a vector of length `5` that contains 1, 2, 3,
seq(1,5)
```

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

Other helpful shortcuts

- `seq(from,to,by)` function provides a way to create a sequence of values with a specific incrementer.

```
# Generates a vector of length `10` containing 0.0, 0.1, .
seq(0, 1, by = 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Other helpful shortcuts

For the next set of shortcuts, note the following assignments:

```r
y = c(1,5,6,2,4) # Create a vector
n = length(y)    # Obtain the Length
```

Other helpful shortcuts

- `seq_len()` function provides a way to create a vector based on length.

```
# Generates a vector starting at `1` and going to `n`.
seq_len(n)
```

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

Note: We will see an example later where this is used to protect a looping counter.

Other helpful shortcuts

- `seq_along()` function provides a way to obtain an index for each element in the vector.

```
# Generates a vector starting at `1` and
# going to `length(y)`.
seq_along(y)
```

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

Note: We will see an example later where this is used to protect a looping counter.

A note on shortcuts

Warning: If the term being multiplied is another vector, R will recycle values in the previous vector.
Take for example:

```
x           # Original
```

```
## [1] 1 2 3 4 5 6
```

```
c(-1,1)*x # Recycled values of x
```

```
## [1] -1  2 -3  4 -5  6
```

A note on shortcuts

This is helpful when constructing confidence intervals. Consider the Confidence Intervals for Proportions Formula:

$$Estimate \pm MOE \tag{1}$$

$$\hat{p} \pm z_{\alpha/2} \left(\sqrt{\frac{\hat{p}\,(1-\hat{p})}{n}} \right) \tag{2}$$

How could we use this property to avoid recomputing the MOE?