Day 1 Morning

Contents

Why R? What is R anyway?

R is older than you, and its growth has been exponential. It has an enthusiastic community and a nice way of self-maintaining over time. Lots of research teams that didn't use to use R now do because it's: free, fast, and it's easy to solve problems without having an expert on-hand.

R is a rather high-level language, meaning that it's largely based on and built atop other languages. Usually when writing R code, you do so in a live R session, interactively. That helps you verify what you're doing as you do it. Once the code gets humming you might turn it into proper functions and document it for sharing (aka package it). And that's how R grows, via user-developed packages. Here's one from your dear instructor:

```r
# a bunch of stuff will print to the console
install.packages("HMDHFDplus")
# now you have the package:
library(HMDHFDplus)
```

Literally, it's that easy to get someone else's package, assuming that it's in the main repository. I think we might use this package for example data from time to time. Let's take a look at CRAN, Stack Overflow, and github.

There are many ways to access R, and Rstudio is the best one probably. Let's take a look around.

Data Types

First, let's looks at some of the basic kinds of data types and data objects that you'll frequently use in R. Think of data types as how values are stored, sort of like being explicit about the value format of a cell in a spreadsheet. Objects are like containers for values, and there are different kinds. Let's just get rolling.

You can use R to calculate stuff:

```r
.001 * exp(.07*70)
```

```
## [1] 0.1342898
```

```r
# functions use parentheses
```

But the value that R produces is not saved anywhere. But at least you got the answer. If you want to keep the result, then assign it:

```r
ans <- .001 * exp(.07*70)
ans
```

```
## [1] 0.1342898
```

```r
# when sqrt() gets fed ans, it's getting the value that ans evaluates to,
# *not* the above equation.
sqrt(ans)
```

```
## [1] 0.3664557
```

```r
# what kind of data is ans?
typeof(ans)
```

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

`ans` is an object. The value of `ans` is some sort of data type, `double` in this case

You can also use `=` to assign, but the arrow makes the direction of assignment explicit. Most R-folk use the arrow. If it's assigned, then you can use it like a number. You can change its value:

```r
ans <- 5
ans
```

```
## [1] 5
```

```r
# R is picky about naming:
Ans <- 6
ans == Ans
```

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

You always need to be exact when referring to named things. And FYI `==` is the way to *ask* if two things are equal. Anyway,R does stuff other than numbers, of course. Here are some character strings:

```r
# some character stuff
first.name <- "Tim"
last.name <-"Riffe"
paste(last.name, first.name, sep = ", ")
```

```
## [1] "Riffe, Tim"
```

```r
typeof(first.name)
```

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

R has many other data types, like dates `?Date`, factors (categorical vars for regressions) `?factor`, and different kinds of number formats, such as `integer`, `numeric`, `double`, and `complex`. Here are some tips: Don't bother coding stuff as `factor`, and instead use `character`, data types. Functions that expect `factor` data will correctly interpret `character` data, but sometimes not vice versa. Also, you rarely need to care about what number format you're using. Stuff tends to work itself out and you can afford to be lazy about it at this point in time. So, we're going to stick to `character` and `numeric` data for the rest of this tutorial.

Help files are easy to call up, and tend to be written in a systematic way. We'll use them very frequently. Seriously, very frequently.

You can coerce most data types and data objects to other data types and objects using `as.blabla`, where `blabla` is the destination data type or object.

```r
# to get a help file, just do
help("as.character")
# or
?as.character
as.character(1)
```

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

R also has your standard values, such as `NA` (all kinds of missing), `NaN` (not a number), `Inf`, `pi`, and others.

Data Objects

In R you can have tons of objects in your session, each callable by its name. Again, there are many kinds of objects, and you can invent new ones. We'll talk about `lists`, `vectors`, `data.frames`, `matrix`(c)es, and maybe `arrays`. Again, these data structures are like containers, with names.

`list` objects

Lists are the most general and flexible object in R, with the drawback that they can be awkward to work with. How so?

```r
mylist <- list(a = 1:5,
               b = "my_name",
               c = pi,
               d = list(a = rnorm(1000), b = .001),
               e = matrix(0,5,5))
str(mylist)
```

```
## List of 5
##  $ a: int [1:5] 1 2 3 4 5
##  $ b: chr "my_name"
##  $ c: num 3.14
##  $ d:List of 2
##   ..$ a: num [1:1000] 0.192 0.288 -1.077 0.613 -0.726 ...
##   ..$ b: num 0.001
##  $ e: num [1:5, 1:5] 0 0 0 0 0 0 0 0 0 0 ...
```

```
# you can access the parts in different ways:
mylist[["a"]]
```

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

```
mylist[[1]]
```

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

```
mylist$c
```

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

```
# you can remove parts like so:
mylist$c <- NULL
# you can over-write like so:
mylist$e <- 1
str(mylist)
```

```
## List of 4
##  $ a: int [1:5] 1 2 3 4 5
##  $ b: chr "my_name"
##  $ d:List of 2
##   ..$ a: num [1:1000] 0.192 0.288 -1.077 0.613 -0.726 ...
##   ..$ b: num 0.001
##  $ e: num 1
```

Lots going on there. Lists are very free-form. When I use R I see them in two common circumstances: 1) I have a list of identically structured objects where I want to calculate the same thing for each element of the list (these might be lifetables) 2) you use a statistical function in R that returns tons of output in the form of a list.

`vector` objects

While lists are the most free-form, vectors are the most basic. Vectors must always be of the same data type, and each element of a vector can only be a single value, not an entire object.

```
v1 <- runif(100)
v1
```

```
##   [1] 0.288419098 0.146081639 0.996151696 0.864810443 0.170558668
##   [6] 0.861750679 0.669742129 0.915669606 0.330907233 0.015840261
##  [11] 0.723645854 0.583032412 0.559595000 0.705202486 0.675904841
##  [16] 0.454814174 0.465145672 0.576981214 0.694483037 0.127073229
##  [21] 0.747754973 0.355003510 0.907095322 0.303523937 0.917982118
##  [26] 0.283172311 0.456333756 0.305618000 0.670088039 0.661974895
##  [31] 0.486451278 0.299213637 0.928568031 0.682662136 0.300859137
##  [36] 0.436669271 0.110397772 0.483113957 0.248746562 0.210202758
##  [41] 0.299876504 0.722280214 0.517376783 0.980300563 0.133837648
##  [46] 0.608300640 0.269742784 0.214177520 0.920520799 0.427333477
##  [51] 0.585820915 0.215897786 0.718560599 0.774749447 0.392225366
##  [56] 0.638893075 0.373269672 0.247141801 0.598917974 0.272155100
##  [61] 0.399716632 0.877286481 0.342261393 0.190106143 0.459400402
##  [66] 0.858524390 0.350570860 0.809392425 0.744238899 0.599173242
##  [71] 0.247328370 0.581613892 0.002706292 0.269373149 0.302645145
##  [76] 0.863265288 0.112704045 0.480083400 0.791248904 0.962706168
##  [81] 0.258578456 0.586511562 0.205815116 0.451923266 0.656724387
##  [86] 0.468943157 0.100376264 0.204493413 0.507441122 0.469253805
##  [91] 0.201379773 0.453093924 0.109677749 0.721128895 0.896972051
##  [96] 0.723492972 0.073467502 0.703816590 0.474340262 0.327406675
```

```
length(v1)
```

```
## [1] 100
```

```
mean(v1)
```

```
## [1] 0.493718
```

```
# this is how to make one by hand:
v2 <- c(1,5,8)

# replace an element:
v1[1:3] <- v2
# or
v1[length(v1)] <- pi

# cut down:
v1 <- v1[20:30]
# or explicitly:
v1 <- v1[-c(1:3)]
v1 # not much left!
```

```
## [1] 0.9070953 0.3035239 0.9179821 0.2831723 0.4563338 0.3056180 0.6700880
## [8] 0.6619749
```

```
# and they can be appended like so:
v3 <- c(v1,v2,rnorm(8))
v3
```

```
##  [1]  0.90709532  0.30352394  0.91798212  0.28317231  0.45633376
##  [6]  0.30561800  0.67008804  0.66197490  1.00000000  5.00000000
## [11]  8.00000000  0.88514449  0.93815057 -0.48777996  0.05790437
## [16]  0.52892650 -0.52824406 -0.11814821  0.29974399
```

If you're doing matrix stuff, vectors are treated as column vectors by default. Let's move on.

`data.frame` objects

`data.frame`s are very very common in R. These are like tables in a spreadsheet. Each column is the same length, and you can index by both columns and rows. The columns do not need to be of the same type of data.

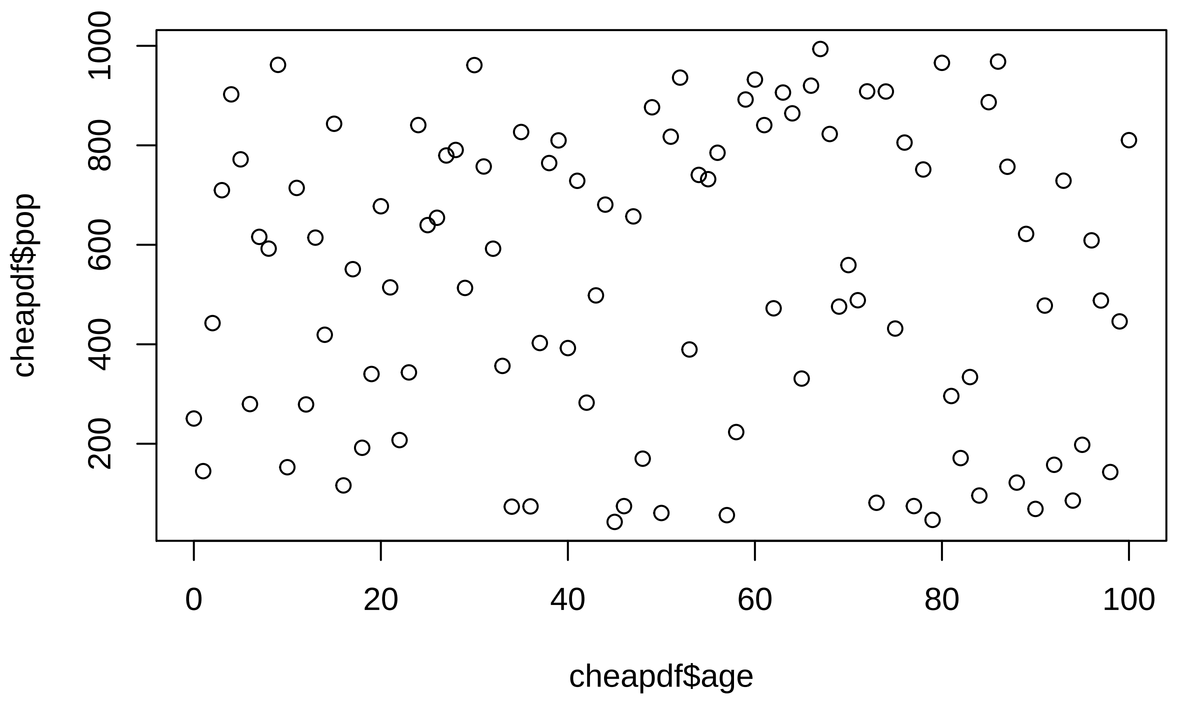
```
# = to assign in functions!!!
cheapdf <- data.frame(age = 0:100, sex = "f", pop = runif(101)*1e3)
# look at the top:
head(cheapdf)
```

```
##   age sex      pop
## 1   0   f 250.7132
## 2   1   f 144.9498
## 3   2   f 442.5534
## 4   3   f 709.7614
## 5   4   f 902.4335
## 6   5   f 771.7918
```

```
tail(cheapdf)
```

```
##     age sex      pop
## 96   95   f 197.9068
## 97   96   f 608.9053
## 98   97   f 487.9869
## 99   98   f 143.1200
## 100  99   f 445.9940
## 101 100   f 810.3885
```

```
# you can index in many ways:
plot(cheapdf$age, cheapdf$pop)
```



```
# index by value
cheapdf$pop[cheapdf$age > 90]
```

```
##  [1] 477.75006 157.69711 728.82421  85.75179 197.90685 608.90530 487.98686
##  [8] 143.12002 445.99399 810.38850
```

```
# index by position:
cheapdf[1:5,3]
```

```
## [1] 250.7132 144.9498 442.5534 709.7614 902.4335
```

```
# bottom right corner
cheapdf[nrow(cheapdf),ncol(cheapdf)]
```

```
## [1] 810.3885
```

```
# or some combo (you can access it like a list too)
cheapdf[[1]]
```

```
##   [1]   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
##  [18]  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33
##  [35]  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
##  [52]  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67
##  [69]  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
##  [86]  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

```
# eliminate a column
cheapdf[[2]] <- NULL

# stick one on:
cheapdf <- cbind(cheapdf,sex="f")
head(cheapdf)
```

```
##   age      pop sex
## 1   0 250.7132   f
## 2   1 144.9498   f
## 3   2 442.5534   f
## 4   3 709.7614   f
## 5   4 902.4335   f
## 6   5 771.7918   f
```

```
# always good to have dims in mind!
dim(cheapdf)
```

```
## [1] 101   3
```

`matrix` objects:

And finally, matrices. Matrices are very useful. They are like `data.frames` except they must always be of the same dimension, and there are only two useful ways of indexing in them (position, and named-dimension):

```r
mat  <- matrix(runif(36), ncol = 6)
mat
```

```
##           [,1]      [,2]       [,3]       [,4]      [,5]        [,6]
## [1,] 0.3459027 0.4487298 0.65626155 0.54740879 0.8630221 0.009465768
## [2,] 0.8632833 0.3491009 0.97294322 0.04798972 0.1340614 0.839671262
## [3,] 0.5824243 0.3614737 0.48315250 0.21820828 0.0979110 0.375958305
## [4,] 0.9746127 0.7851550 0.04558556 0.96583478 0.6821245 0.120157854
## [5,] 0.8263700 0.6103037 0.48058362 0.89936945 0.4770391 0.031820896
## [6,] 0.7654827 0.4422878 0.75577882 0.65954520 0.4630585 0.651787262
```

```r
# assign some fake names:
dimnames(mat) <- list(1:6,letters[1:6])
mat
```

```
##           a         b          c          d         e           f
## 1 0.3459027 0.4487298 0.65626155 0.54740879 0.8630221 0.009465768
## 2 0.8632833 0.3491009 0.97294322 0.04798972 0.1340614 0.839671262
## 3 0.5824243 0.3614737 0.48315250 0.21820828 0.0979110 0.375958305
## 4 0.9746127 0.7851550 0.04558556 0.96583478 0.6821245 0.120157854
## 5 0.8263700 0.6103037 0.48058362 0.89936945 0.4770391 0.031820896
## 6 0.7654827 0.4422878 0.75577882 0.65954520 0.4630585 0.651787262
```

```r
# add it up
sum(mat)
```

```
## [1] 18.83387
```

```r
# index it: (rows then columns)
mat[1:2,3:4]
```

```
##           c          d
## 1 0.6562615 0.54740879
## 2 0.9729432 0.04798972
```

```r
# elementwise operations
mat / 2
```

```
##           a         b          c          d          e           f
## 1 0.1729513 0.2243649 0.32813077 0.27370439 0.43151105 0.004732884
## 2 0.4316417 0.1745505 0.48647161 0.02399486 0.06703069 0.419835631
## 3 0.2912122 0.1807369 0.24157625 0.10910414 0.04895550 0.187979152
## 4 0.4873063 0.3925775 0.02279278 0.48291739 0.34106225 0.060078927
## 5 0.4131850 0.3051518 0.24029181 0.44968473 0.23851956 0.015910448
## 6 0.3827414 0.2211439 0.37788941 0.32977260 0.23152924 0.325893631
```

```r
# some convenient functions:
colSums(mat)
```

```
##        a        b        c        d        e        f
## 4.358076 2.997051 3.394305 3.338356 2.717217 2.028861
```

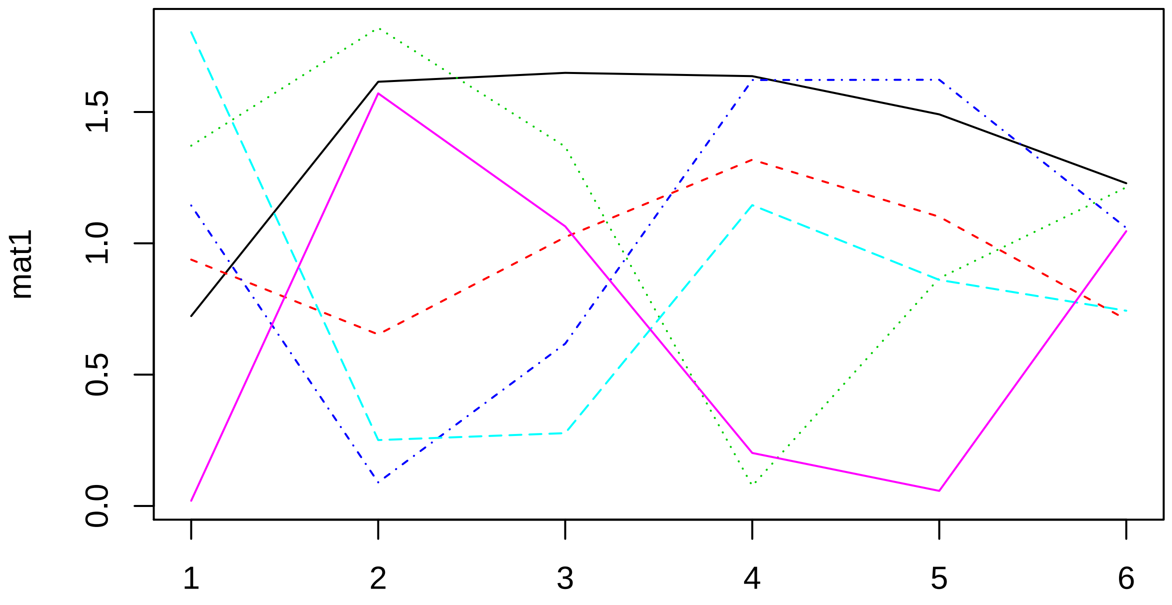
```r
rowMeans(mat)
```

```
##         1         2         3         4         5         6
## 0.4784651 0.5345083 0.3531880 0.5955784 0.5542478 0.6229900
```

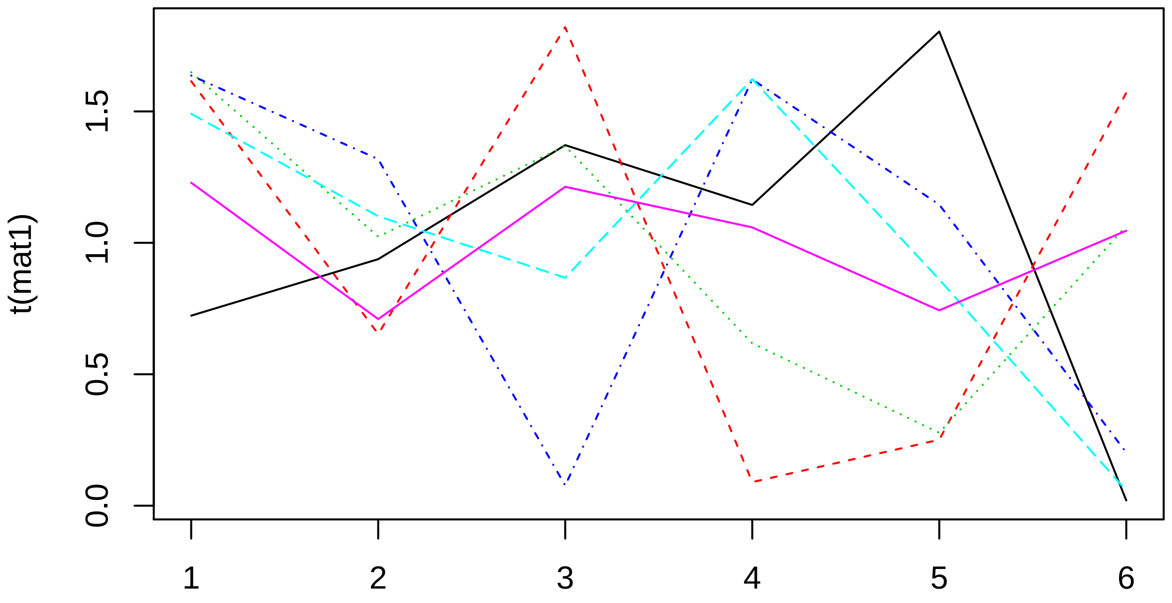
```r
# vectors go into the rows, elementwise by default:
mat1 <- mat / rowMeans(mat)
rowSums(mat1)
```

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

```r
matplot(mat1,type='l')
```



```r
matplot(t(mat1),type='l')
```



That’s enough basics for now