Data Structures

Vectors

Let's start with vectors, the most common of R's data structures.

Atomic vectors

Atomic vectors are 1-dimensional data structures that must be all the same type

```
# Logical
logical_vector = c(TRUE, FALSE, TRUE)

# Integer
int_vector = c(1L, 3L, 6L)  # Notice the 'L' here

# Double
num_vector = c(3.4, 493.22, 239992.229)

# Character
char_vector = c("Do", "you", "love", "cats", "?")
```

A couple of points on atomic vectors:

- They are constructed using the `c()` function
- To specify an integer use the L suffix
- A vector must be the all the same type
- Use `typeof()` to see what type you have or an "is" function to check type

```
typeof(int_vector)
## [1] "integer"

is.character(char_vector)
## [1] TRUE

is.numeric(logical_vector)
## [1] FALSE
```

Lists

A list is like an atomic vector but each item in the list can be any type, including other lists, atomic vectors, data frames, or matrices. Use `list()` to make a list.

```
my_list = list("meow", 12, c(4, 5, 10), list("I have 10 cats", FALSE))

str(my_list)
## List of 4
##  $ : chr "meow"
##  $ : num 12
##  $ : num [1:3] 4 5 10
##  $ :List of 2
##   ..$ : chr "I have 10 cats"
##   ..$ : logi FALSE
```

Lists are very powerful and although confusing at first it is worth spending time learning how to use them. In particular when we come to the "apply" family of functions we will see how lists can make our lives much easier.

Factors

Ah, the dreaded factors! They are used to store categorical variables and although it is tempting to think of them as character vectors this is a dangerous mistake (you will get scratched, badly!).

Factors make perfect sense if you are a statistician designing a programming language (!) but to everyone else they exist solely to torment us with confusing errors.

A factor is really just an integer vector with an additional attribute, `levels()`, which defines the possible values.

```
crazy_factor = factor(c("up", "down", "down", "sideways", "up"))

print(crazy_factor)
## [1] up       down     down     sideways up
## Levels: down sideways up

levels(crazy_factor)
## [1] "down"     "sideways" "up"

as.integer(crazy_factor)
## [1] 3 1 1 2 3
```

But why not just use character vectors, you ask?

Believe it or not factor do have some useful properties. For example factors allow you to specify all possible values a variable may take even if those values are not in your dataset.

```
cool_animals = c("cat", "cat", "dog", "dog")

cool_animals_factor = factor(cool_animals, levels = c("cat", "dog", "bunny"))

table(cool_animals_factor)
## cool_animals_factor
##   cat   dog bunny
##     2     2     0
```

But for the most part factors are important for various statistics involving categorical variables, as you'll see for things like linear models. Love 'em or hate 'em, factors are integral to using R so better learn 'em.

Matrices

A matrix is a 2-dimensional vector and like atomic vector must be all of a single type.

```
mat = matrix(1:12, ncol = 4, nrow = 3)

print(mat)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
nrow(mat)
## [1] 3

ncol(mat)
## [1] 4
```

Matrices can have row and column names

```
colnames(mat) = c("A", "B", "C", "D")

rownames(mat) = c("cat", "tiger", "lion")

print(mat)
##       A B C  D
## cat   1 4 7 10
## tiger 2 5 8 11
## lion  3 6 9 12
```

Data frames

Data frames are very powerful data structures in R and will play a central role in most of the work you'll do. These are probably the most familiar data structures to most people as they resemble spreadsheets quite closely, at least on the surface.

You can think of data frame as a set of identically sized lists lined up together in columns with a few key features. Each column must be of the same type but column types can be any of the basic data types (character, integer, etc).

This makes them very useful for structured heterogeneous data, like what many of you generate in the lab everyday. However, it is very important to remember that they are not spreadsheets and to use them effectively you need to forget everything you've learned about Excel (which is probably a good idea anyway).

Here let's use a data frame that comes built in to R

```
head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

str(iris)
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Notice the $ notation, similar to what we saw for lists. We can use this to extract singe columns.

```
iris$Sepal.Length
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Alternatively,

```
iris[["Sepal.Length"]]
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

And now for some basic indexing.

```
# get the first 3 rows of the last 2 columns
iris[1:3, 4:5]
##   Petal.Width Species
## 1         0.2  setosa
## 2         0.2  setosa
## 3         0.2  setosa

# get the 10th row of the 'Petal.Width' column
iris[10, "Petal.Width"]
## [1] 0.1

# get the entire 4th row
iris[4, ]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 4          4.6         3.1          1.5         0.2  setosa
```

We'll do more indexing in the next session but for now here a couple of key points about indexing.

- The square brackets `[]` are used to index
- Row is first then column `[row, column]`
- Rows and columns can be indexed by names, with a character vector, although you'll see this much more commonly for columns, rather than rows.
- When indexing by integer, you can use a single value `3`, a vector `c(1, 3, 8)`, or a range `2:10`.
- Indexing in R starts at 1. This is different than most programming languages such as python where indexing begins at 0. For those new to programming this makes more sense but those with programming experience may find this frustrating.