(http://software-carpentry.org)

Understanding basic data types in R

- To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those.
- **Very Important** to understand because these are the things you will manipulate on a day-to-day basis in R. Most common source of frustration among beginners.
- Everything in `R` is an object.

`R` has 5 basic atomic classes

- logical (e.g., `TRUE`, `FALSE`)
- integer (e.g,, 2L, as.integer(3))
- numeric (real or decimal) (e.g, 2, 2.0, pi)
- complex (e.g, 1 + 0i, 1 + 4i)
- character (e.g, "a", "swc")

```
typeof() # what is it?
class() # what is it? (sorry)
storage.mode() # what is it? (very sorry)
length() # how long is it? What about two dimensional objects?
attributes() # does it have any metadata?
```

R also has many data structures. These include

- vector
- list
- matrix
- data frame
- factors (we will avoid these, but they have their uses)
- tables

Vectors

A vector is the most common and basic data structure in `R` and is pretty much the workhorse of R. Vectors can be of two types:

- atomic vectors
- lists

Atomic Vectors A vector can be a vector of characters, logical, integers or numeric.

Create an empty vector with `vector()`

```
x <- vector()
# with a pre-defined length
x <- vector(length = 10)
# with a length and type
vector("character", length = 10)
vector("numeric", length = 10)
vector("integer", length = 10)
vector("logical", length = 10)
```

The general pattern is `vector(class of object, length)`. You can also create vectors by concatenating them using the `c()` function.

Various examples:

```
x <- c(1, 2, 3)
```

x is a numeric vector. These are the most common kind. They are numeric objects and are treated as double precision real numbers. To explicitly create integers, add a `L` at the end.

```
x1 <- c(1L, 2L, 3L)
```

You can also have logical vectors.

```
y <- c(TRUE, TRUE, FALSE, FALSE)
```

(Don't use `T` and `F`!)

Finally you can have character vectors:

```
z <- c("Alec", "Dan", "Rob", "Rich")
```

Examine your vector

```
typeof(z)
length(z)
class(z)
str(z)
```

Question: Do you see property that's common to all these vectors above?

Add elements

```
z <- c(z, "Annette")
z
```

More examples of vectors

```
x <- c(0.5, 0.7)
x <- c(TRUE, FALSE)
x <- c("a", "b", "c", "d", "e")
x <- 9:100
x <- c(i+0i, 2+4i)
```

You can also create vectors as sequence of numbers

```
series <- 1:10
seq(10)
seq(1, 10, by = 0.1)
```

Other objects

`Inf` is infinity. You can have positive or negative infinity.

```
 1/0
# [1] Inf
 1/Inf
# [1] 0
```

`NaN` means Not a number. it's an undefined value.

```
0/0
NaN.
```

Each object has an attribute. Attributes can be part of an object of R. These include

- names
- dimnames
- length
- class
- attributes (contain metadata)

For a vector, `length(vector_name)` is just the total number of elements.

Vectors may only have one type

R will create a resulting vector that is the least common denominator. The coercion will move towards the one that's easiest to coerce to.

Guess what the following do without running them first

```
xx <- c(1.7, "a")
xx <- c(TRUE, 2)
xx <- c("a", TRUE)
```

This is called implicit coercion.

The coersion rule goes `logical` -> `integer` -> `numeric` -> `complex` -> `character`.

You can also coerce vectors explicitly using the `as.<class_name>`. Example

```
as.numeric()
as.character()
```

When you coerce an existing numeric vector with `as.numeric()`, it does nothing.

```
x <- 0:6
as.numeric(x)
as.logical(x)
as.character(x)
as.complex(x)
```

Sometimes coercions, especially nonsensical ones won't work.

```
x <- c("a", "b", "c")
as.numeric(x)
as.logical(x)
# both don't work
```

Sometimes there is implicit conversion

```
1 < "2"
# TRUE
"1" > 2
# FALSE
1 < "a"
# TRUE
```

Matrix

Matrices are a special vector in R. They are not a separate class of object but simply a vector but now with dimensions added on to it. Matrices have rows and columns.

```
m <- matrix(nrow = 2, ncol = 2)
m
dim(m)
same as
attributes(m)
```

Matrices are constructed columnwise.

```
m <- matrix(1:6, nrow=2, ncol =3)
```

Other ways to construct a matrix

```
m <- 1:10
dim(m) <- c(2,5)
```

This takes a vector and transform into a matrix with 2 rows and 5 columns.

Another way is to bind columns or rows using `cbind()` and `rbind()`.

```
x <- 1:3
y <- 10:12
cbind(x,y)
# or
rbind(x,y)
```

List

In R lists act as containers. Unlike atomic vectors, its contents are not restricted to a single mode and can encompass any data type. Lists are sometimes called recursive vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

List is a special vector. Each element can be a different class.

Create lists using `list` or coerce other objects using `as.list()`

```
x <- list(1, "a", TRUE, 1+4i)
```

```
x <- 1:10
x <- as.list(x)
length(x)
```

What is the class of `x[1]`? how about `x[[1]]`?

```
xlist <- list(a = "Rich FitzJohn", b = 1:10, data = head(iris))
```

what is the length of this object? what about its structure?

List can contain as many lists nested inside.

```
temp <- list(list(list(list())))
temp
is.recursive(temp)
```

Lists are extremely useful inside functions. You can "staple" together lots of different kinds of results into a single object that a function can return.

It doesn't print out like a vector. Prints a new line for each element.

Elements are indexed by double brackets. Single brackets will still return another list.

Factors

Factors are special vectors that represent categorical data. Factors can be ordered or unordered and are important when for modelling functions such as `lm()` and `glm()` and also in plot methods.

Factors can only contain pre-defined values.

Factors are pretty much integers that have labels on them. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings. Some string methods will coerce factors to strings, while others will throw an error.

Sometimes factors can be left unordered. Example: male, female

Other times you might want factors to be ordered (or ranked). Example: low, medium, high.

Underlying it's represented by numbers 1,2,3.

They are better than using simple integer labels because factors are what are called self describing. male and female is more descriptive than 1s and 2s. Helpful when there is no additional metadata.

Which is male? 1 or 2? You wouldn't be able to tell with just integer data. Factors have this information built in.

Factors can be created with `factor()`. Input is a character vector.

```
x <- factor(c("yes", "no", "no", "yes", "yes"))
x
```

`table(x)` will return a frequency table.

`unclass(x)` strips out the class information.

In modelling functions, important to know what baseline levels is. This is the first factor but by default the ordering is determined by alphabetical order of words entered. You can change this by specifying the levels.

```
x <- factor(c("yes", "no", "yes"), levels = c("yes", "no"))
```

Data frame

A data frame is a very important data type in R. It's pretty much the de facto data structure for most tabular data and what we use for statistics.

data frames can have additional attributes such as `rownames()`. This can be useful for annotating data, like subject*id or sample*id. But most of the time they are not used.

e.g. `rownames()` useful for annotating data. subject names. other times they are not useful.

- Data frames Usually created by read.csv and read.table.
- Can convert to `matrix` with `data.matrix()`
- Coercion will force and not always what you expect.

- Can also create with `data.frame()` function.

With and data frame, you can do `nrow(df)` and `ncol(df)` rownames are usually 1..n.

Combining data frames

```
df <- data.frame(id = letters[1:10], x = 1:10, y = rnorm(10))
> df
   id  x          y
1   a  1 -0.3913992
2   b  2 -0.8607609
3   c  3  1.1234612
4   d  4 -0.8283688
5   e  5 -0.8785586
6   f  6  0.2116839
7   g  7 -0.3795995
8   h  8 -0.5992272
9   i  9  0.3203085
10  j 10  0.2901185
```

`cbind(df, data.frame(z = 4))`

When you combine column wise, only row numbers need to match. If you are adding a vector, it will get repeated.

Useful functions `head()` - see first 5 rows `tail()` - see last 5 rows `dim()` - see dimensions `nrow()` - number of rows `ncol()` - number of columns `str()` - structure of each column `names()` - will list column names for a data.frame (or any object really).

A data frame is a special type of list where every element of a list has same length.

See that it is actually a special list:

```
> is.list(iris)
[1] TRUE
> class(iris)
[1] "data.frame"
 >
```

Naming objects

Other R objects can also have names not just true for data.frames. Adding names is helpful since it's useful for readable code and self describing objects.

```
x <- 1:3
names(x) <- c("rich", "daniel", "diego")
x
```

Lists can also have names.

```
x <- as.list(1:10)
names(x) <- letters[seq(x)]
x
```

Finally matrices can have names and these are called `dimnames`

```
m <- matrix(1:4, nrow = 2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
# first element = rownames
# second element = colnames
```

Missing values

denoted by `NA` and/or `NaN` for undefined mathematical operations.

```
is.na()
is.nan()
```

check for both.

NA values have a class. So you can have both an integer NA and a missing character NA.

NaN is also NA. But not the other way around.

```
x <- c(1,2, NA, 4, 5)
```

`is.na(x)` returns logical. shows third

`is.nan(x)` # none are NaN.

```
x <- c(1,2, NA, NaN, 4, 5)
```

`is.na(x)` shows 2 TRUE. `is.nan(x)` shows 1 TRUE

Missing values are very important in R, but can be very frustrating for new users.

What do these do? What should they do? `1 == NA NA == NA`

How can we do that sort of comparison?

Diagnostic functions in R

Super helpful functions

- `str()` Compactly display the internal structure of an R object. Perhaps the most uesful diagnostic function in R.
- `names()` Names of elements within an object
- `class()` Retrieves the internal class of an object

- `mode()` Get or set the type or storage mode of an object.
- `length()` Retrieve or set the dimension of an object.
- `dim()` Retrieve or set the dimension of an object.
- `R --vanilla` - Allows you to start a clean session of R. A great way to test whether your code is reproducible.
- `sessionInfo()` Print version information about R and attached or loaded packages.
- `options()` Allow the user to set and examine a variety of global options which affect the way in which R computes and displays its results.

`str()` is your best friend

`str` is short for structure. You can use it on any object. Try the following:

```
x <- 1:10
class(x)
mode(x)
str(x)
```