Vectors

The most basic unit available in R to store data are *vectors*. Complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

Creating vectors

We can create vectors using the function `c`, which stands for concatenate. We use `c` to *concatenate* entires in the following way:

```
codes <- c(380, 124, 818)
codes
```

```
## [1] 380 124 818
```

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variables names.

```
country <- c("italy", "canada", "egypt")
```

Note that if you type

```
country <- c(italy, canada, egypt)
```

you recieve an error becuase the variables `italy`, `canada` and `egypt` are not defined: R looks for variables with those names and returns an error.

Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

The object `codes` continues to be a numeric vector:

```
class(codes)
```

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

but with names

```r
names(codes)
```

```
## [1] "italy"  "canada" "egypt"
```

If the use of strings without quotes looks confusing, know that you can use the quotes as well

```r
codes <- c("italy" = 380, "canada" = 124, "egypt" = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

There is no difference between this call and the previous one: one of the many ways R is quirky compared to other languages.

We can also assign names using the `names` function:

```r
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
codes
```

```
##  italy canada  egypt
##    380    124    818
```

Sequences

Another useful function for creating vectors generates sequences

```r
seq(1, 10)
```

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

The first argument defines the start, and the second the end. The default is to go up in increments of 1, but a third argument let's us tell it how much to jump by:

```r
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

If we want consecutive integers we can use the following shorthand

```r
1:10
```

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

Note that when we use this function, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
```

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

However, note that as soon as we create something that's not an integer the class changes:

```
class(seq(1, 10))
```

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

```
class(seq(1, 10, 0.5))
```

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

Subsetting

We use square brackets to access specific elements of a list. For the vector `codes` we defined above, we can access the second element using

```
codes[2]
```

```
## canada
##    124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
```

```
## italy egypt
##   380   818
```

The sequences defined above are particularly useful if we want to access, say, the first two elements

```
codes[1:2]
```

```
##  italy canada
##    380    124
```

If the elements have names, we can also access the entries using these names. Here are two examples.

```
codes["canada"]
```

```
## canada
##    124
```

```
codes[c("egypt","italy")]
```

```
## egypt italy
##   818   380
```

Coercion

In general, *coercion* is an attempt by R to be flexible with data types. When an entry does not match the expected, R tries to guess what we meant before throwing an error. This can also lead to confusion. Failing to understand *coercion* can drive a programmer crazy when attempting to code in R since it behaves quite diffently from most other languages in this regard. Let's learn about it with some examples.

We said that elements of a vector must be all of the same type. So if we try to combine, say, numbers and characters you might expect an error

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at `x` and its class:

```
x
```

```
## [1] "1"      "canada" "3"
```

```
class(x)
```

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

R *coerced* the data into characters. It guessed that because you put a character string in the vector you meant the 1 and 3 to actually be character strings `"1"` and `"3"`. The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to force a specific coercion. For example you can turn numbers into characters with

```
x <- 1:5
y <- as.character(x)
y
```

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

And you can turn it back with `as.numeric`.

```
as.numeric(y)
```

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

This function is actually quite useful as datasets that include numbers as character strings are common.

Not Availables (NA)

When these coercion functions encounter an impossible case it gives us a warning and turns the entry into a special value called an `NA` for "not available". For example:

```r
x <- c("1", "b", "3")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  3
```

R does not have any guesses for what number you want when you type `b` so it does not try.

Note that as a data scientist you will encounter `NA`s often as they are used for missing data, a common problem in real-life datasets.