Lecture 1: R basics

Falco J. Bargagli-Stoffi

03/06/2020

Introduction to R

Setting Working Directory

`R` is always pointed at a directory on your computer. You can find out which directory by running the `getwd` (get working directory) function; this function has no arguments. To change your working directory, use `setwd` and specify the path to the desired folder.

In the first block of code, I verify there is nothing in the work space, I check the current directory, and I specify a directory I want to go to, navigate further to the subfolder packageFiles.

```
ls()
```

```
## character(0)
```

```
getwd()
```

```
## [1] "G:/Il mio Drive/github/lab-data-science.github.io/r_code"
```

```
setwd("~/files")
```

Getting Help

Before asking others for help, it's generally a good idea for you to try to help yourself. `R` includes extensive facilities for accessing documentation and searching for help. There are also specialized search engines for accessing information about R on the internet, and general internet search engines can also prove useful

```
?plot # Opens the help page for the mean function
??plotting # Searches for topics containing words like "plotting"
??"regression model" # Searches for topics containing phrases like this
```

This is totally equivalent to the following:

```
help("plot")
help.search("plotting")
help.search("regression model")
```

Most functions have examples that you can run to get a better idea of how they work. Use the example function to run these. There are also some longer demonstrations of concepts that are accessible with the demo function:

```
example(plot)
```

```
demo() #list all demonstrations
demo(image)
```

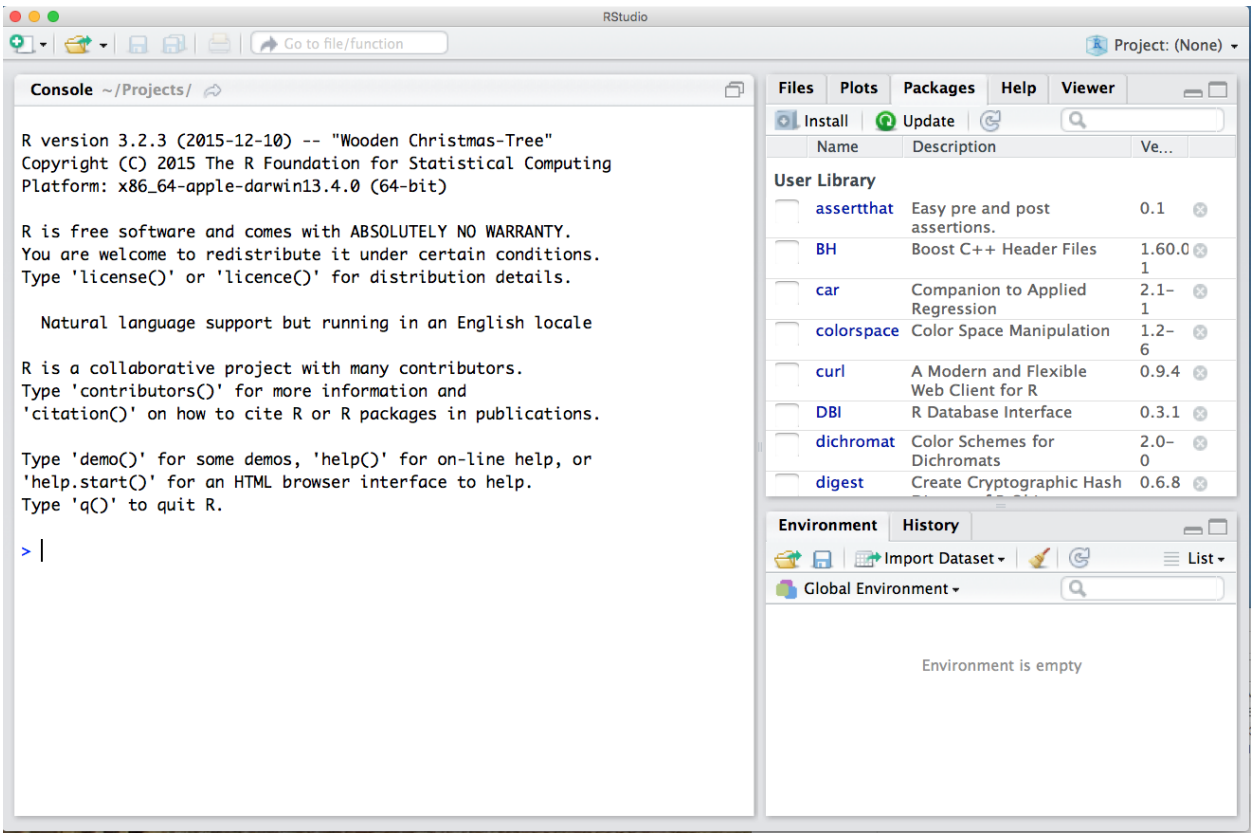
Libraries/Packages

A library is a piece of software that provides additional functionality to R, beyond what's contained in the basic installation. R has an enormous ecosystem of libraries (number in the tens of thousands) for various data-analysis tasks, ranging from the simple to the very sophisticated. The mosaic package was written specifically for use in statistics classrooms. We will use it along with a handful of other packages during this class, so you'll need to learn how to install them.

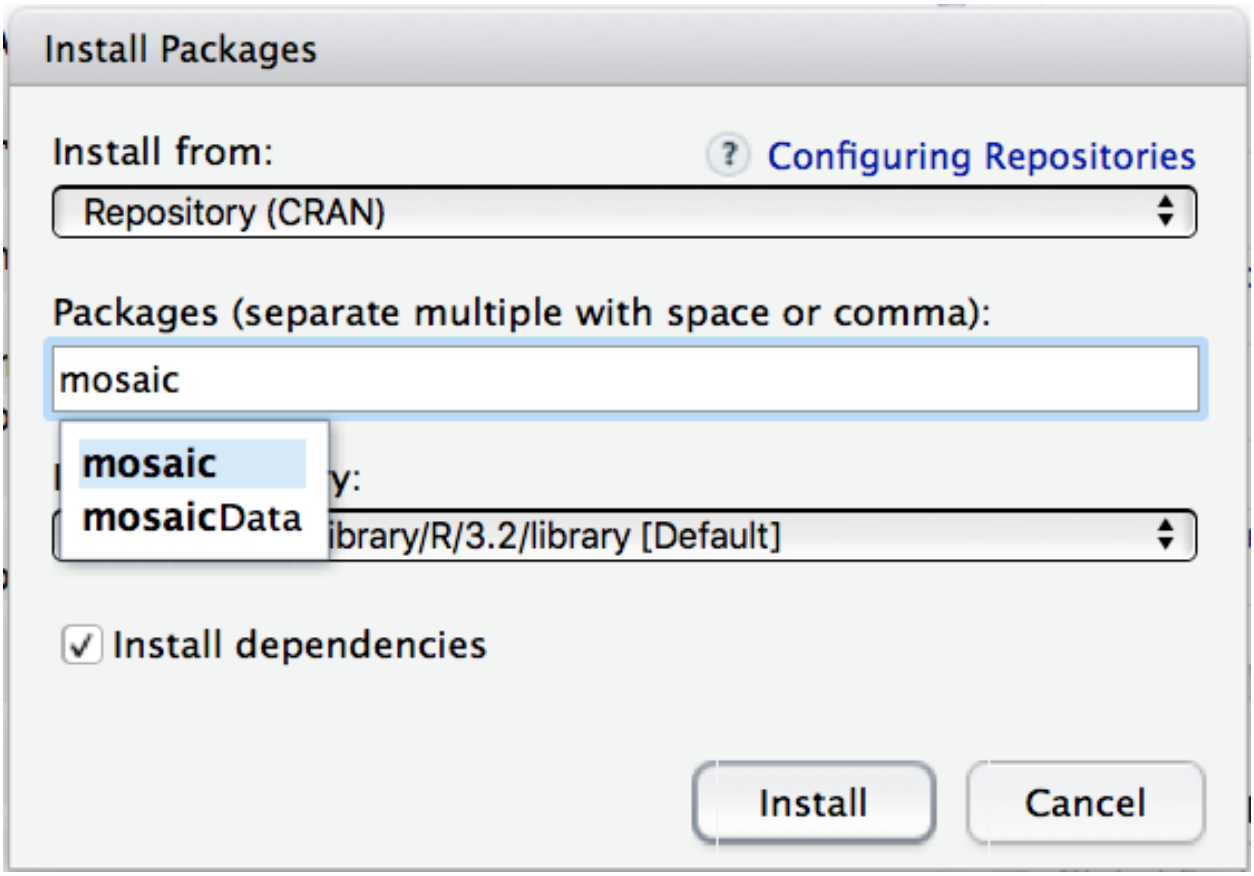
Conveniently, libraries, also called packages, are installed from within RStudio itself.

Installing a library from within RStudio

Launch RStudio, and find the row of tabs for Files, Plots, Packages, etc. You can see this on the upper-right panel in the figure below.



Click on the Packages tab, and then on the button in the row just beneath it that says "Install". You should see a window pop up that looks like the one below. Type "mosaic" into the field where it says "Packages."



RStudio may autocomplete the available packages that begin with the string you've typed in. If so, select "mosaic" from the list that appears. IMPORTANT: make sure that the box next to "Install dependencies" is checked (as it is in the picture above). If it isn't, check it.

Click the Install button. Now a bunch of text will get dumped to the Console window, like this:

```
trying URL 'http://lib.stat.cmu.edu/R/CRAN/bin/macosx/mavericks/contrib/3.2/colo
rspace_1.2-6.tgz'
Content type 'application/x-gzip' length 396993 bytes (387 KB)
==================================================
downloaded 387 KB

trying URL 'http://lib.stat.cmu.edu/R/CRAN/bin/macosx/mavericks/contrib/3.2/minq
a_1.2.4.tgz'
Content type 'application/x-gzip' length 242427 bytes (236 KB)
==================================================
downloaded 236 KB

trying URL 'http://lib.stat.cmu.edu/R/CRAN/bin/macosx/mavericks/contrib/3.2/nlop
tr_1.0.4.tgz'
Content type 'application/x-gzip' length 932065 bytes (910 KB)
==================================================
downloaded 910 KB

trying URL 'http://lib.stat.cmu.edu/R/CRAN/bin/macosx/mavericks/contrib/3.2/Rcpp
Eigen_0.3.2.5.1.tgz'
Content type 'application/x-gzip' length 3931821 bytes (3.7 MB)
==================================================
downloaded 3.7 MB
```

This indicates that the mosaic library (and several other libraries upon which it depends) are being installed.

It's possible that RStudio will give you a prompt like the following:

```
Installing package into '/Users/james/Library/R/3.2/library'
(as 'lib' is unspecified)
also installing the dependencies 'colorspace', 'minqa', 'nloptr', 'RcppEigen', '
dichromat', 'munsell', 'labeling', 'lme4', 'SparseM', 'MatrixModels', 'stringi',
'assertthat', 'R6', 'Rcpp', 'magrittr', 'DBI', 'BH', 'digest', 'gtable', 'plyr',
'scales', 'pbkrtest', 'quantreg', 'stringr', 'curl', 'RColorBrewer', 'dplyr', 'g
gplot2', 'car', 'mosaicData', 'lazyeval', 'reshape2', 'readr', 'latticeExtra', '
ggdendro', 'gridExtra'


  There are binary versions available but the source versions are later:
       binary source needs_compilation
Rcpp   0.12.2 0.12.3              TRUE
digest  0.6.8  0.6.9              TRUE

Do you want to install from sources the packages which need compilation?
y/n: n
```

If so, just type "n" for no, and hit Enter to continue.

Eventually you will see that text has stopped appearing in the console, and that you have a blinking cursor next to a caret mark (>). When this happens, the mosaic package has been installed, and is now available to be loaded and used.

Some packages contain vignettes, which are short documents on how to use the packages. You can browse all vignettes on your machine using `browseVignettes`.

```
browseVignettes()
```

```
## starting httpd help server ... done
```

An alternative way (preferred), you can install a package by running the following function:

```
install.packages('')
```

If you want to install a package directly from GitHub (usually beta versions) you can run the following code:

```
library(devtools)
install_github("github_page/package_name")
```

Once you have installed a packages you can recall all its function by uploading it at the real beginning of your code using the `library()` function. By uploading a library you also upload all the libraries that are directly connected with it.

```
library(bartMachine)
```

```
## Loading required package: rJava
## Loading required package: bartMachineJARs
## Loading required package: car
## Loading required package: carData
## Loading required package: randomForest
## randomForest 4.6-14
## Type rfNews() to see new features/changes/bug fixes.
## Loading required package: missForest
## Loading required package: foreach
## Loading required package: itertools
## Loading required package: iterators
## Welcome to bartMachine v1.2.3! You have 0.48GB memory available.
##
## If you run out of memory, restart R, and use e.g.
## 'options(java.parameters = "-Xmx5g")' for 5GB of RAM before you call
## 'library(bartMachine)'.
```

```
library(purrr)
```

```
##
## Attaching package: 'purrr'
## The following objects are masked from 'package:foreach':
##
##     accumulate, when
## The following object is masked from 'package:car':
##
##     some
```

```
library(tibble)
```

If you want to use a single function from a package, without uploading it you can use the following option:

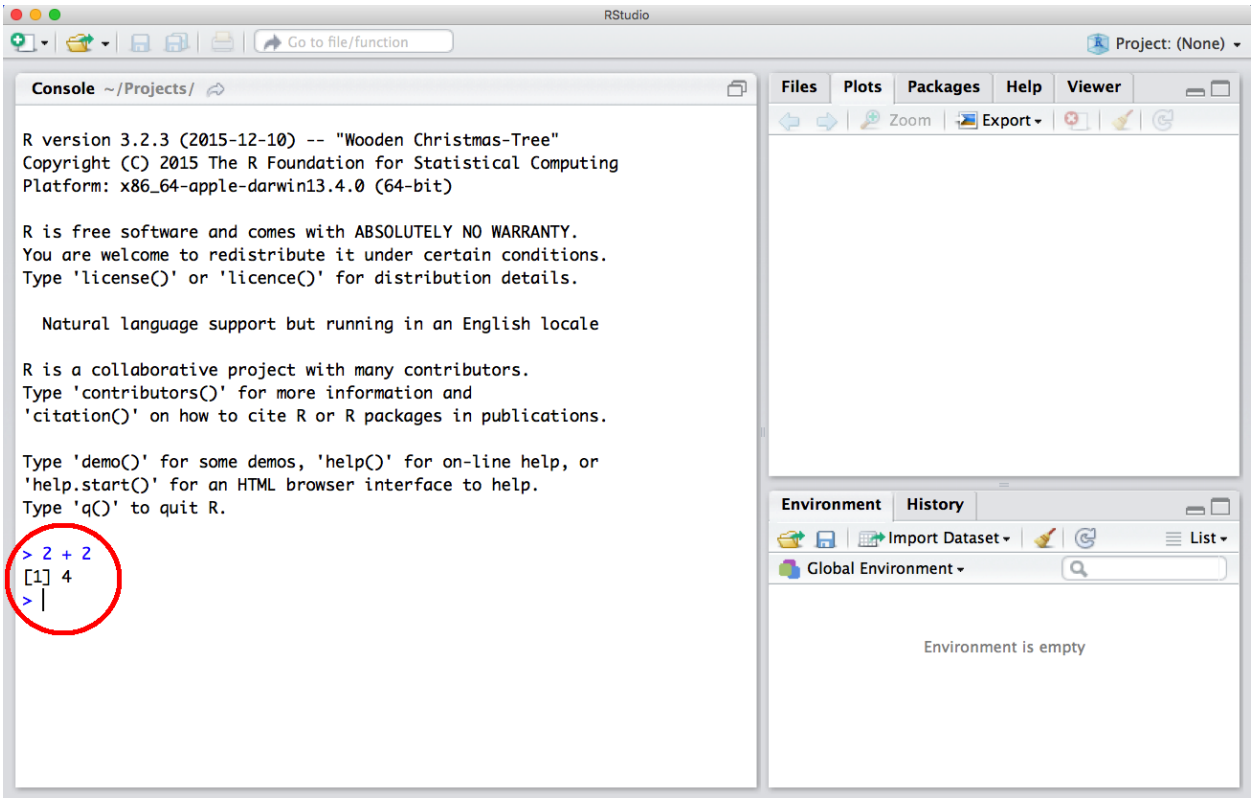
```
bartMachine::bartMachine()
```

Using R as a calculator

This is a "hello world" walk-through the usage of `R` as a calculator.

Let's start with the simplest thing of all that you can do with R: use it as a scientific calculator.

Open up RStudio; on the left-hand side is the Console window with a little caret mark (>), where you can type in commands directly and see the results of computations (circled in red below).



Try typing some basic mathematical expressions directly into the console and verify that the output looks right. To execute any command, like `2+2`, just type it into the console and hit Enter.

```
2 + 2
```

```
## [1] 4
```

```
3*7
```

```
## [1] 21
```

```
20/4
```

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

```
2^3
```

```
## [1] 8
```

```
sqrt(16)
```

```
## [1] 4
```

```
log10(100)
```

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

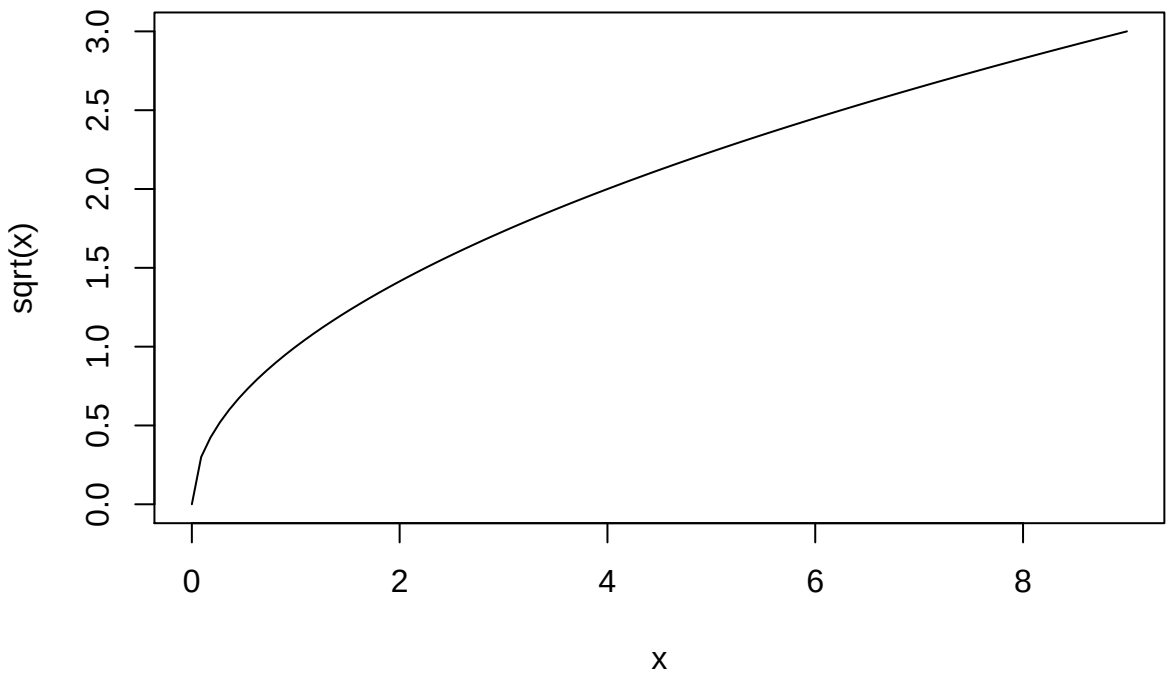
R obeys the usual order of operations for mathematical expressions. Therefore use parentheses appropriately to convey your meaning:

```
(4 + 2) * 3
```

```
## [1] 18
```

You can also use R as a graphing calculator. Try typing the following command:

```
curve(sqrt(x), from=0, to=9)
```



You should get a plot of the square-root function from $x = 0$ to $x = 9$ that pops up on your screen.

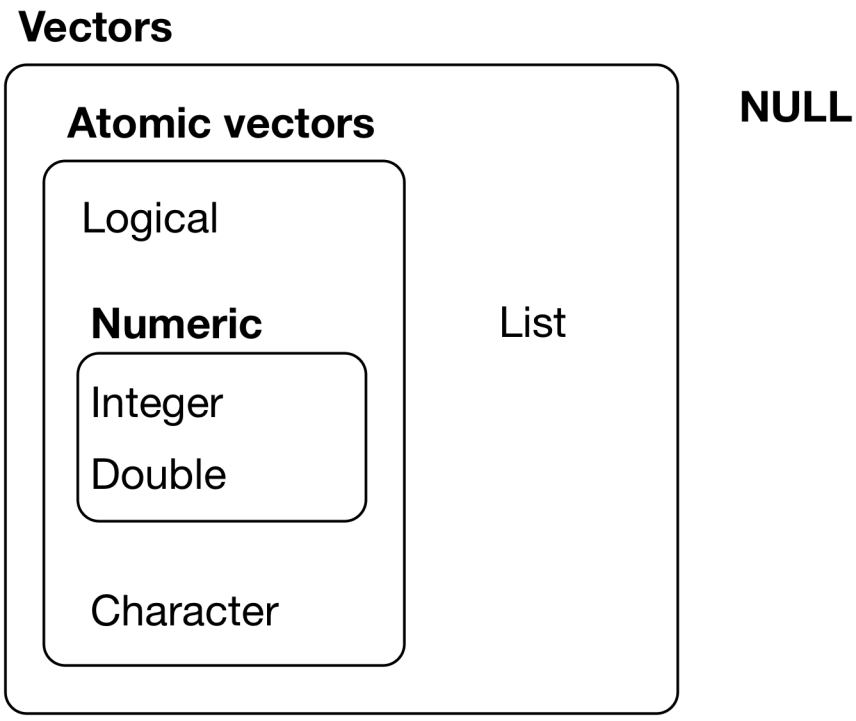
Vectors

There are two types of vectors in R.

1. Atomic vectors, of which there are six types: logical, integer, double, character, complex, and raw. Integer and double vectors are collectively known as numeric vectors.
2. Lists, which are sometimes called recursive vectors because lists can contain other lists.

The main difference between atomic vectors and lists is that atomic vectors are homogeneous, while lists can be heterogeneous. There's one other related object: NULL. NULL is often used to represent the absence

of a vector (as opposed to `NA` which is used to represent the absence of a value in a vector). `NULL` typically behaves like a vector of length 0.



Every vector has two key properties:

1. its *type*, which you can determine with the `typeof()` function.
2. its *length*, which you can determine with the `length` function.

```
typeof(letters)
```

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

```
typeof(1:10)
```

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

```
length(1:10)
```

```
## [1] 10
```

Vectors can also contain arbitrary additional metadata in the form of attributes. These attributes are used to create augmented vectors which build on additional behaviour. There are three important types of augmented vector:

1. factors are built on top of integer vectors;
2. dates and date-times are built on top of numeric vectors;
3. data frames and tibbles are built on top of lists.

This lecture will introduce you to these important vectors from simplest to most complicated. You'll start with atomic vectors, then build up to lists, and finish off with augmented vectors.

Important types of atomic vector

The four most important types of atomic vector are logical, integer, double, and character. Raw and complex are rarely used during a data analysis, so we won't discuss them here.

Logical

Logical vectors are the simplest type of atomic vector because they can take only three possible values: FALSE, TRUE, and NA. Logical vectors are usually constructed with comparison operators, as described in comparisons. You can also create them by hand with c():

```
1:10 == 7
```

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

```
c(TRUE, TRUE, FALSE, NA)
```

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

Numeric

Integer and double vectors are known collectively as numeric vectors. In R, numbers are doubles by default. To make an integer, place an L after the number:

```
typeof(1)
```

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

```
typeof(1L)
```

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

The distinction between integers and doubles is not usually important, but there are two important differences that you should be aware of:

1. Doubles are approximations. Doubles represent floating point numbers that can not always be precisely represented with a fixed amount of memory. This means that you should consider all doubles to be approximations. This behaviour is common when working with floating point numbers: most calculations include some approximation error. Instead of comparing floating point numbers using ==, you should use dplyr::near() which allows for some numerical tolerance.
2. Integers have one special value: NA, while doubles have four: NA, NaN, Inf and -Inf. All three special values NaN, Inf and -Inf can arise during division:

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf  NaN  Inf
```

Avoid using == to check for these other special values. Instead use the helper functions is.finite(), is.infinite(), and is.nan():

	0	Inf	NA	NaN
is.finite()	x			
is.infinite()		x		
is.na()			x	x
is.nan()				x

Character

Character vectors are the most complex type of atomic vector, because each element of a character vector is a string, and a string can contain an arbitrary amount of data.

R uses a global string pool. This means that each unique string is only stored in memory once, and every use of the string points to that representation. This reduces the amount of memory needed by duplicated strings. You can see this behaviour in practice with \texttt{pryr::object_size()}:

```
x <- "This is a reasonably long string."
pryr::object_size(x)
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
## 152 B
```

```
y <- rep(x, 1000)
pryr::object_size(y)
```

```
## 8.14 kB
```

y doesn't take up 1000x as much memory as x, because each element of y is just a pointer to that same string. A pointer is 8 bytes, so 1000 pointers to a 136 B string is $8 \cdot 1000 + 136 = 8.13$ kB.

Missing values

Note that each type of atomic vector has its own missing value:

```
NA            # logical
```

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

```
NA_integer_   # integer
```

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

```
NA_real_      # double
```

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

```
NA_character_ # character
```

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

Normally you don't need to know about these different types because you can always use NA and it will be converted to the correct type using the implicit coercion rules described next. However, there are some functions that are strict about their inputs, so it's useful to have this knowledge sitting in your back pocket so you can be specific when needed.

Using atomic vectors

Now that you understand the different types of atomic vector, it's useful to review some of the important tools for working with them. These include:

1. How to convert from one type to another, and when that happens automatically;
2. How to tell if an object is a specific type of vector;
3. What happens when you work with vectors of different lengths;
4. How to name the elements of a vector;
5. How to pull out elements of interest.

Coercion

There are two ways to convert, or coerce, one type of vector to another:

1. Explicit coercion happens when you call a function like as.logical(), as.integer(), as.double(), or as.character(). Whenever you find yourself using explicit coercion, you should always check whether you can make the fix upstream, so that the vector never had the wrong type in the first place.
2. Implicit coercion happens when you use a vector in a specific context that expects a certain type of vector. For example, when you use a logical vector with a numeric summary function, or when you use a double vector where an integer vector is expected.

Because explicit coercion is used relatively rarely, and is largely easy to understand, I'll focus on implicit coercion here.

You've already seen the most important type of implicit coercion: using a logical vector in a numeric context. In this case TRUE is converted to 1 and FALSE converted to 0. That means the sum of a logical vector is the number of trues, and the mean of a logical vector is the proportion of trues:

```
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y)  # how many are greater than 10?
```

```
## [1] 53
```

```
mean(y) # what proportion are greater than 10?
```

```
## [1] 0.53
```

You may see some code (typically older) that relies on implicit coercion in the opposite direction, from integer to logical:

```
if (length(x)) {
  # do something
}
```

In this case, 0 is converted to FALSE and everything else is converted to TRUE. I think this makes it harder to understand your code, and I don't recommend it. Instead be explicit: length(x) > 0.

It's also important to understand what happens when you try and create a vector containing multiple types with c(): the most complex type always wins.

```
typeof(c(TRUE, 1L))
```

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

```
typeof(c(1L, 1.5))
```

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

```
typeof(c(1.5, "a"))
```

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

An atomic vector can not have a mix of different types because the type is a property of the complete vector, not the individual elements. If you need to mix multiple types in the same vector, you should use a list, which you'll learn about shortly.

Test functions

Sometimes you want to do different things based on the type of vector. One option is to use `typeof()`. Another is to use a test function which returns a TRUE or FALSE. Base `R` provides many functions like is.vector() and is.atomic(), but they often return surprising results. Instead, it's safer to use the is_* functions provided by purrr, which are summarised in the table below.

	lgl	int	dbl	chr	list
is_logical()	X				
is_integer()		X			
is_double()			X		
is_numeric()		X	X		
is_character()				X	
is_atomic()	X	X	X	X	
is_list()					X
is_vector()	X	X	X	X	X

Each predicate also comes with a "scalar" version, like is_scalar_atomic(), which checks that the length is 1. This is useful, for example, if you want to check that an argument to your function is a single logical value.

Scalars and recycling rules

As well as implicitly coercing the types of vectors to be compatible, R will also implicitly coerce the length of vectors. This is called vector recycling, because the shorter vector is repeated, or recycled, to the same length as the longer vector.

This is generally most useful when you are mixing vectors and "scalars". I put scalars in quotes because R doesn't actually have scalars: instead, a single number is a vector of length 1. Because there are no scalars, most built-in functions are vectorised, meaning that they will operate on a vector of numbers. That's why, for example, this code works:

```
sample(10) + 100
```

```
##  [1] 101 102 103 105 107 106 109 110 108 104
```

```
runif(10) > 0.5
```

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

In R, basic mathematical operations work with vectors. That means that you should never need to perform explicit iteration when performing simple mathematical computations.

It's intuitive what should happen if you add two vectors of the same length, or a vector and a "scalar", but what happens if you add two vectors of different lengths?

```
1:10 + 1:2
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

Here R, will expand the shortest vector to the same length as the longest, so called recycling. This is silent except when the length of the longer is not an integer multiple of the length of the shorter:

```
1:10 + 1:3
```

```
## Warning in 1:10 + 1:3: longer object length is not a multiple of shorter
## object length
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

While vector recycling can be used to create very succinct, clever code, it can also silently conceal problems. ‘

Naming vectors

All types of vectors can be named. You can name them during creation with c():

```
c(x = 1, y = 2, z = 4)
```

```
## x y z
## 1 2 4
```

Or after the fact with purrr::set_names():

```
purrr::set_names(1:3, c("a", "b", "c"))
```

```
## a b c
## 1 2 3
```

Named vectors are most useful for subsetting, described next.

Subsetting

[is the subsetting function, and is called like x[a]. There are four types of things that you can subset a vector with:

1) A numeric vector containing only integers. The integers must either be all positive, all negative, or zero.

Subsetting with positive integers keeps the elements at those positions:

```
x <- c("one", "two", "three", "four", "five")
x[c(3, 2, 5)]
```

```
## [1] "three" "two"   "five"
```

By repeating a position, you can actually make a longer output than input:

```
x[c(1, 1, 5, 5, 5, 2)]
```

```
## [1] "one"  "one"  "five" "five" "five" "two"
```

Negative values drop the elements at the specified positions:

```
x[c(-1, -3, -5)]
```

```
## [1] "two"  "four"
```

It's an error to mix positive and negative values:

```
x[c(1, -1)]
```

The error message mentions subsetting with zero, which returns no values:

```
x[0]
```

```
## character(0)
```

This is not useful very often, but it can be helpful if you want to create unusual data structures to test your functions with.

2) Subsetting with a logical vector keeps all values corresponding to a TRUE value. This is most often useful in conjunction with the comparison functions.

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# All non-missing values of x
x[!is.na(x)]
```

```
## [1] 10  3  5  8  1
```

3) If you have a named vector, you can subset it with a character vector:

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
```

```
## xyz def
##   5   2
```

Like with positive integers, you can also use a character vector to duplicate individual entries.

4) The simplest type of subsetting is nothing, x[], which returns the complete x. This is not useful for subsetting vectors, but it is useful when subsetting matrices (and other high dimensional structures) because it lets you select all the rows or all the columns, by leaving that index blank. For example, if x is 2d, x[1,] selects the first row and all the columns, and x[, -1] selects all rows and all columns except the first.

To learn more about the applications of subsetting, reading the "Subsetting" chapter of Advanced R: http://adv-r.had.co.nz/Subsetting.html#applications.

There is an important variation of [called [[. [[only ever extracts a single element, and always drops names. It's a good idea to use it whenever you want to make it clear that you're extracting a single item, as in a for loop. The distinction between [and [[is most important for lists, as we'll see shortly.

Recursive vectors (lists)

Lists are a step up in complexity from atomic vectors, because lists can contain other lists. This makes them suitable for representing hierarchical or tree-like structures. You create a list with list():

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

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

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

A very useful tool for working with lists is str() because it focusses on the structure, not the contents.

```
str(x)
```

```
## List of 3
##  $ : num 1
##  $ : num 2
##  $ : num 3
```

```
x_named <- list(a = 1, b = 2, c = 3)
str(x_named)
```

```
## List of 3
##  $ a: num 1
##  $ b: num 2
##  $ c: num 3
```

Unlike atomic vectors, list() can contain a mix of objects:

```
y <- list("a", 1L, 1.5, TRUE)
str(y)
```

```
## List of 4
##  $ : chr "a"
##  $ : int 1
##  $ : num 1.5
##  $ : logi TRUE
```

Lists can even contain other lists!

```
z <- list(list(1, 2), list(3, 4))
str(z)
```

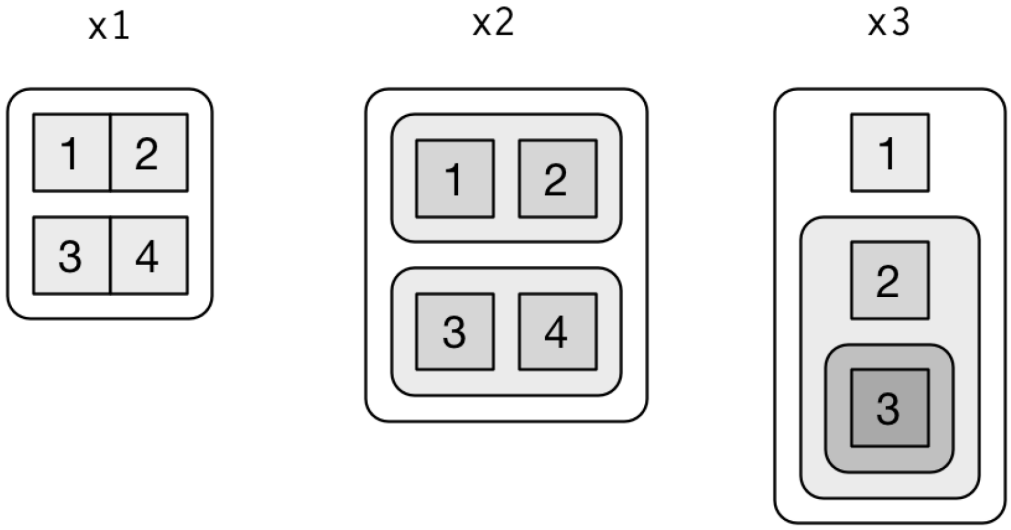
```
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ :List of 2
##   ..$ : num 3
##   ..$ : num 4
```

Visualising lists

To explain more complicated list manipulation functions, it's helpful to have a visual representation of lists. For example, take these three lists:

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

I'll draw them as follows:



There are three principles:

Lists have rounded corners. Atomic vectors have square corners.

Children are drawn inside their parent, and have a slightly darker background to make it easier to see the hierarchy.

The orientation of the children (i.e. rows or columns) isn't important, so I'll pick a row or column orientation to either save space or illustrate an important property in the example.

Subsetting

There are three ways to subset a list, which I'll illustrate with a list named a:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

1) [extracts a sub-list. The result will always be a list.

```
str(a[1:2])
```

```
## List of 2
##  $ a: int [1:3] 1 2 3
##  $ b: chr "a string"
```

```
str(a[4])
```

```
## List of 1
##  $ d:List of 2
##   ..$ : num -1
##   ..$ : num -5
```

Like with vectors, you can subset with a logical, integer, or character vector.

2) [[extracts a single component from a list. It removes a level of hierarchy from the list.

```
str(a[[1]])
```

```
##  int [1:3] 1 2 3
```

```r
str(a[[4]])
```

```
## List of 2
##  $ : num -1
##  $ : num -5
```

3) $ is a shorthand for extracting named elements of a list. It works similarly to [[except that you don't need to use quotes.

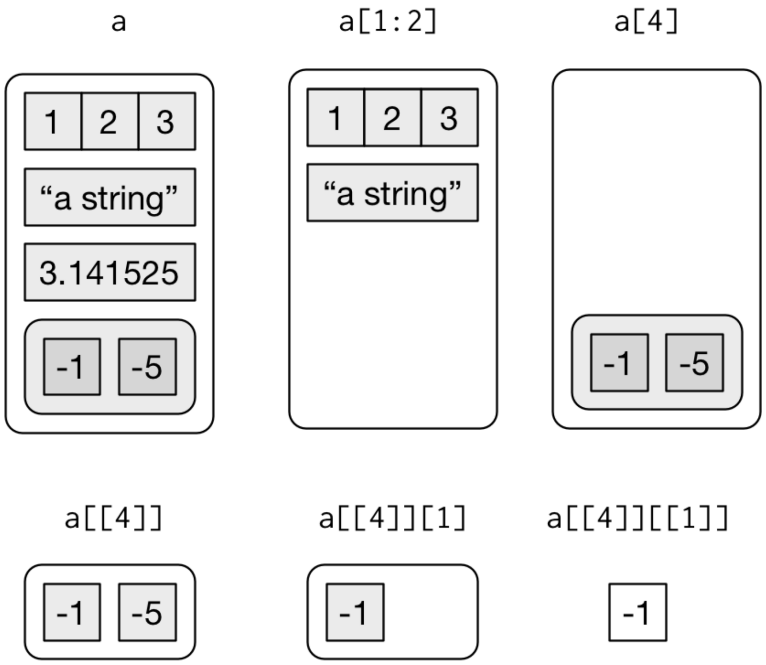
```r
a$a
```

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

```r
a[["a"]]
```

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

The distinction between [and [[is really important for lists, because [[drills down into the list while [returns a new, smaller list.



Attributes

Any vector can contain arbitrary additional metadata through its attributes. You can think of attributes as named list of vectors that can be attached to any object. You can get and set individual attribute values with attr() or see them all at once with attributes().

```r
x <- 1:10
attr(x, "greeting")
```

```
## NULL
```

```r
attr(x, "greeting") <- "Hi!"
attr(x, "farewell") <- "Bye!"
attributes(x)
```

```
## $greeting
## [1] "Hi!"
##
## $farewell
## [1] "Bye!"
```

There are three very important attributes that are used to implement fundamental parts of R:

1. Names are used to name the elements of a vector;
2. dimensions (dims, for short) make a vector behave like a matrix or array;
3. class is used to implement the S3 object oriented system.

You've seen names above, and we won't cover dimensions because we don't use matrices here. Generic functions are key to object oriented programming in `R`, because they make functions behave differently for different classes of input. A detailed discussion of object oriented programming is beyond the scope of this book, but you can read more about it in Advanced R at http://adv-r.had.co.nz/OO-essentials.html#s3.

Here's what a typical generic function looks like:

```
as.Date
```

```
## function (x, ...)
## UseMethod("as.Date")
## <bytecode: 0x0000000019979208>
## <environment: namespace:base>
```

The call to "UseMethod" means that this is a generic function, and it will call a specific method, a function, based on the class of the first argument. (All methods are functions; not all functions are methods). You can list all the methods for a generic with methods():

```
methods("as.Date")
```

```
## [1] as.Date.character as.Date.default   as.Date.factor    as.Date.IDate*
## [5] as.Date.numeric   as.Date.POSIXct   as.Date.POSIXlt
## see '?methods' for accessing help and source code
```

For example, if x is a character vector, as.Date() will call as.Date.character(); if it's a factor, it'll call as.Date.factor().

You can see the specific implementation of a method with getS3method():

```
getS3method("as.Date", "default")
```

```
## function (x, ...)
## {
##     if (inherits(x, "Date"))
##         x
##     else if (is.logical(x) && all(is.na(x)))
##         .Date(as.numeric(x))
##     else stop(gettextf("do not know how to convert '%s' to class %s",
##         deparse(substitute(x)), dQuote("Date")), domain = NA)
## }
## <bytecode: 0x0000000018ab8260>
## <environment: namespace:base>
```

```
getS3method("as.Date", "numeric")
```

```
## function (x, origin, ...)
## {
##     if (missing(origin))
```

```
##         stop("'origin' must be supplied")
##     as.Date(origin, ...) + x
## }
## <bytecode: 0x0000000018abb620>
## <environment: namespace:base>
```

The most important S3 generic is print(): it controls how the object is printed when you type its name at the console. Other important generics are the subsetting functions [, [[, and $.

Augmented vectors

Atomic vectors and lists are the building blocks for other important vector types like factors and dates. These are called *augmented vectors*, because they are vectors with additional attributes, including class. Because augmented vectors have a class, they behave differently to the atomic vector on which they are built. In this book, we make use of four important augmented vectors:

1. Factors
2. Dates
3. Date-times
4. Tibbles

These are described below.

Factors

Factors are designed to represent categorical data that can take a fixed set of possible values. Factors are built on top of integers, and have a levels attribute:

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)
```

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

```
attributes(x)
```

```
## $levels
## [1] "ab" "cd" "ef"
##
## $class
## [1] "factor"
```

Dates and date-times

Dates in R are numeric vectors that represent the number of days since 1 January 1970.

```
x <- as.Date("1971-01-01")
unclass(x)
```

```
## [1] 365
```

```
typeof(x)
```

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

```
attributes(x)
```

```
## $class
## [1] "Date"
```

Date-times are numeric vectors with class POSIXct that represent the number of seconds since 1 January 1970. (In case you were wondering, "POSIXct" stands for "Portable Operating System Interface", calendar time.)

```
x <- lubridate::ymd_hm("1970-01-01 01:00")
unclass(x)
```

```
## [1] 3600
## attr(,"tzone")
## [1] "UTC"
```

```
typeof(x)
```

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

```
attributes(x)
```

```
## $class
## [1] "POSIXct" "POSIXt"
##
## $tzone
## [1] "UTC"
```

The tzone attribute is optional. It controls how the time is printed, not what absolute time it refers to.

```
attr(x, "tzone") <- "US/Pacific"
x
```

```
## [1] "1969-12-31 17:00:00 PST"
```

```
attr(x, "tzone") <- "US/Eastern"
x
```

```
## [1] "1969-12-31 20:00:00 EST"
```

There is another type of date-times called POSIXlt. These are built on top of named lists:

```
y <- as.POSIXlt(x)
typeof(y)
```

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

```
attributes(y)
```

```
## $names
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"
##  [8] "yday"   "isdst"  "zone"   "gmtoff"
##
## $class
## [1] "POSIXlt" "POSIXt"
##
## $tzone
## [1] "US/Eastern" "EST"        "EDT"
```

POSIXlts are rare inside the tidyverse. They do crop up in base R, because they are needed to extract specific components of a date, like the year or month. Since lubridate provides helpers for you to do this

instead, you don't need them. POSIXct's are always easier to work with, so if you find you have a POSIXlt, you should always convert it to a regular data time lubridate::as_date_time().

Tibbles

Tibbles are augmented lists: they have class "tbl_df" + "tbl" + "data.frame", and names (column) and row.names attributes:

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)
```

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

```
attributes(tb)
```

```
## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3 4 5
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

The difference between a tibble and a list is that all the elements of a data frame must be vectors with the same length. All functions that work with tibbles enforce this constraint.

Traditional data.frames have a very similar structure:

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)
```

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

```
attributes(df)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
```

The main difference is the class. The class of tibble includes "data.frame" which means tibbles inherit the regular data frame behaviour by default.