Chapter 2 Introduction to R

We're assuming you're either new to R or need a refresher.

We'll start with some basic R operations entered directly in the console in RStudio.

2.1 Variables

Variables are objects that store values. Every computer language, like in math, stores values by assigning them constants or results of expressions. `x <- 5` uses the R standard assignment operator `<-` though you can also use `=` . We'll use `<-` because it is more common and avoids some confusion with other syntax.

Variable names must start with a letter, have no spaces, and not use any names that are built into the R language or used in package libraries, such as reserved words like `for` or function names like `log()`

```
x <- 5
y <- 8
longitude <- -122.4
latitude <- 37.8
my_name <- "Inigo Montoya"
```

To check the value of a variable or other object, you can just enter the name in the console, or even in the code in a code chunk.

```
x
```

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

```
y
```

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

```
longitude
```

```
## [1] -122.4
```

```
latitude
```

```
## [1] 37.8
```

```
my_name
```

```
## [1] "Inigo Montoya"
```

This is counter to the way printing out values work in programming, and you will need to know how this method works as well because you will want to use your code to develop tools that accomplish things, and there are also limitations to what you can see by just naming variables.

To see the values of variables in programming mode, use the `print()` function, or to concatenate character string output, use `paste()` :

```
print(x)
```

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

```
print(y)
```

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

```
print(latitude)
```

```
## [1] 37.8
```

```
paste("The location is latitude", latitude, "longitude", longitude)
```

```
## [1] "The location is latitude 37.8 longitude -122.4"
```

```
paste("My name is", my_name, "-- Prepare to die.")
```

```
## [1] "My name is Inigo Montoya -- Prepare to die."
```

2.2 Functions

Once you have variables or other objects to work with, most of your work involves *functions* such as the well-known math functions

```
log10(100)
log(exp(5))
cos(pi)
sin(90 * pi/180)
```

Most of your work will involve functions and there are too many to name, even in the base functions, not to mention all the packages we will want to use. You will likely have already used the `install.packages()` and `library()` functions that add in an array of other functions. Later we'll also learn how to write our own functions, a capability that is easy to accomplish and also gives you a sense of what developing your own package might be like.

Arithmetic operators There are of course all the normal arithmetic operators (that are actually functions) like + - * /. You're probably familiar with these from using equations in Excel if not in some other programming language you may have learned. These operators look a bit different from how they'd look when creating a nicely formatted equation. $\frac{NIR-R}{NIR+R}$ instead looks like `(NIR-R)/(NIR+R)`. Similarly `*` must be used to multiply; there's no assumed multiplication that we expect in a math equation like $x(2+y)$ which would need to be written `x*(2+y)`.

In contrast to those four well-known operators, the symbol used to exponentiate – raise to a power – varies among programming languages. R uses ** so the the Pythagorean theorem $c^2 = a^2 + b^2$ would be written `c**2 = a**2 + b**2` except for the fact that it wouldn't make sense to R. We'll need to talk about expressions and statements.

2.3 Expressions and Statements

The concepts of expressions and statements are very important to understand when using any programming environment.

An *expression* in R (or any programming language) has a *value* just like a variable has a value. An expression will commonly combine variables and functions to be *evaluated* to derive the value of the expression. Here are some examples of expressions:

```
5
x
x*2
sin(x)
sqrt(a**2 + b**2)
(-b+sqrt(b**2-4*a*c))/2*a
paste("My name is", aname)
```

Note that some of those expressions used previously assigned variables – x, a, b, c, aname.

An expression can be entered in the console to display its current value.

```
cos(pi)
```

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

```
print(cos(pi))
```

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

A *statement* in R does something. It represents a directive we're assigning to the computer, or maybe the environment we're running on the computer (like RStudio, which then runs R). A simple `print()` *statement* seems a lot like what we just did when we entered an expression in the console, but recognize that it *does something*:

```
print("Hello, World")
```

```
## [1] "Hello, World"
```

Which is the same as just typing "Hello, World", but that's just because the job of the console is to display what we are looking for [where we are the ones *doing something*], or if our statement includes something to display.

Statements in R are usually put on one line, but you can use a semicolon to have multiple statements on one line, if desired:

```
x <- 5; print(x); print(x**2)
```

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

```
## [1] 25
```

Many (perhaps most) statements don't actually display anything. For instance:

```
x <- 5
```

doesn't display anything, but it does assign the value 5 to the variable x, so it *does something*. It's an *assignment statement* and uses that special assignment operator `<-` . Most languages just use `=` which the designers of R didn't want to use, to avoid confusing it with the equal sign meaning "is equal to".

An assignment statement assigns an expression to a variable.

2.4 Data Types

Variables, constants and other data elements in R have data types. Common types are numeric and character.

```
x <- 5
class(x)
```

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

```
class(4.5)
```

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

```
class("Fred")
```

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

2.4.1 Integers

By default, R creates double-precision floating-point numeric variables To create integer variables: - append an L to a constant, e.g. `5L` is an integer 5 - convert with `as.integer` We're going to be looking at various `as.` functions in R, more on that later, but we should look at `as.integer()` now. Most other languages use `int()` for this, and what it does is converts *any number* into an integer, *truncating* it to an integer, not rounding it.

```
as.integer(5)
```

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

```
as.integer(4.5)
```

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

To round a number, there's a `round()` function or you can easily use `as.integer` adding 0.5:

```
x <- 4.8
y <- 4.2
as.integer(x + 0.5)
```

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

```
round(x)
```

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

```
as.integer(y + 0.5)
```

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

```
round(y)
```

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

Integer divison:

```
5 %/% 2
```

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

Integer remainder from division (the modulus, using a `%%` to represent the modulo):

```
5 %% 2
```

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

Surprisingly, the values returned by integer division or the remainder are not stored as integers. R seems to prefer floating point…

2.5 Rectangular data

A common data format used in most types of research is *rectangular* data such as in a spreadsheet, with rows and columns, where rows might be *observations* and columns might be *variables*. We'll read this type of data in from spreadsheets or even more commonly from comma-separated-variable (CSV) text files that spreadsheet programs like Excel commonly read in just like their native format.

```
sierraFeb
```

```
## # A tibble: 82 x 7
##    STATION_NAME    COUNTY ELEVATION LATITUDE LONGITUDE PRECIPITATION TEMPERATURE
##    <chr>           <chr>      <dbl>    <dbl>     <dbl>         <dbl>       <dbl>
##  1 GROVELAND 2, C~ Tuolu~     853.      37.8     -120.         176.          6.1
##  2 CANYON DAM, CA~ Plumas    1390.      40.2     -121.         164.          1.4
##  3 KERN RIVER PH ~ Kern       824.      35.8     -118.          67.1         8.9
##  4 DONNER MEMORIA~ Nevada    1810.      39.3     -120.         167.         -0.9
##  5 BOWMAN DAM, CA~ Nevada    1641.      39.5     -121.         277.          2.9
##  6 BRUSH CREEK RA~ Butte     1085.      39.7     -121.         296.         NA  
##  7 GRANT GROVE, C~ Tulare    2012.      36.7     -119.         186.          1.7
##  8 LEE VINING, CA~ Mono      2072.      38.0     -119.          71.9         0.4
##  9 OROVILLE MUNIC~ Butte       57.9     39.5     -122.         138.         10.3
## 10 LEMON COVE, CA~ Tulare     156.      36.4     -119.          62.7        11.3
## # ... with 72 more rows
```

2.6 Data Structures in R

We looked briefly at numeric and character string (we'll abbreviate simply as "string" from here on). We'll also look at factors and dates/times later on.

2.6.1 Vectors

A vector is an ordered collection of numbers, strings, vectors, data frames, etc. What we mostly refer to as vectors are formally called *atomic vectors* which requires that they be *homogeneous* sets of whatever type we're referring to, such as a vector of numbers, or a vector of strings, or a vector of dates/times.

You can create a simple vector with the `c()` function:

```
lats <- c(37.5,47.4,29.4,33.4)
lats
```

```
## [1] 37.5 47.4 29.4 33.4
```

```
states = c("VA", "WA", "TX", "AZ")
states
```

```
## [1] "VA" "WA" "TX" "AZ"
```

```
zips = c(23173, 98801, 78006, 85001)
zips
```

```
## [1] 23173 98801 78006 85001
```

The class of a vector is the type of data it holds

```
temp <- c(10.7, 9.7, 7.7, 9.2, 7.3, 6.7)
class(temp)
```

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

Vectors can only have one data class, and if mixed with character types, numeric elements will become character:

```
mixed <- c(1, "fred", 7)
class(mixed)
```

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

```
mixed[3]   # gets a subset, example of coercion
```

```
## [1] "7"
```

2.6.1.1 NA

Data science requires dealing with missing data by storing some sort of null value, called various things: - null - nodata - NA “not available” or “not applicable”

```
as.numeric(c("1","Fred","5")) # note NA introduced by coercion
```

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

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

Ignoring NA in statistical summaries is commonly used. Where normally the summary statistic can only return NA…

```
mean(as.numeric(c("1", "Fred", "5")))
```

```
## Warning in mean(as.numeric(c("1", "Fred", "5"))): NAs introduced by coercion
```

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

… with `na.rm=T` you can still get the result for all actual data:

```
mean(as.numeric(c("1", "Fred", "5")), na.rm=T)
```

```
## Warning in mean(as.numeric(c("1", "Fred", "5")), na.rm = T): NAs introduced by
## coercion
```

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

Don’t confuse with `nan` (“not a number”) which is used for things like imaginary numbers (explore the help for more on this)

```
is.na(NA)
```

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

```
is.nan(NA)
```

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

```
is.na(as.numeric(''))
```

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

```
is.nan(as.numeric(''))
```

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

```
i <- sqrt(-1)
```

```
## Warning in sqrt(-1): NaNs produced
```

```
is.na(i) # interestingly nan is also na
```

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

```
is.nan(i)
```

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

2.6.1.2 Sequences

An easy way to make a vector from a sequence of values. The following 3 examples are equivalent:

```
seq(1,10)
c(1:10)
c(1,2,3,4,5,6,7,8,9,10)
```

The seq() function has special uses like using a step parameter:

```
seq(2,10,2)
```

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

2.6.1.3 Vectorization and vector arithmetic

Arithmetic on vectors operates element-wise

```
elev <- c(52,394,510,564,725,848,1042,1225,1486,1775,1899,2551)
elevft <- elev / 0.3048
elevft
```

```
##  [1]  170.6037 1292.6509 1673.2283 1850.3937 2378.6089 2782.1522 3418.6352
##  [8] 4019.0289 4875.3281 5823.4908 6230.3150 8369.4226
```

Another example, with 2 vectors:

```
temp03 <- c(13.1,11.4,9.4,10.9,8.9,8.4,6.7,7.6,2.8,1.6,1.2,-2.1)
temp02 <- c(10.7,9.7,7.7,9.2,7.3,6.7,4.0,5.0,0.9,-1.1,-0.8,-4.4)
tempdiff <- temp03 - temp02
tempdiff
```

```
##  [1] 2.4 1.7 1.7 1.7 1.6 1.7 2.7 2.6 1.9 2.7 2.0 2.3
```

2.6.1.4 Plotting vectors

Vectors of Feb temperature, elevation and latitude at stations in the Sierra:

```
temp <- c(10.7, 9.7, 7.7, 9.2, 7.3, 6.7, 4.0, 5.0, 0.9, -1.1, -0.8, -4.4)
elev <- c(52, 394, 510, 564, 725, 848, 1042, 1225, 1486, 1775, 1899, 2551)
lat <- c(39.52, 38.91, 37.97, 38.70, 39.09, 39.25, 39.94, 37.75, 40.35, 39.33, 39.17, 38.2
```

Plot individually

```
plot(temp)
```

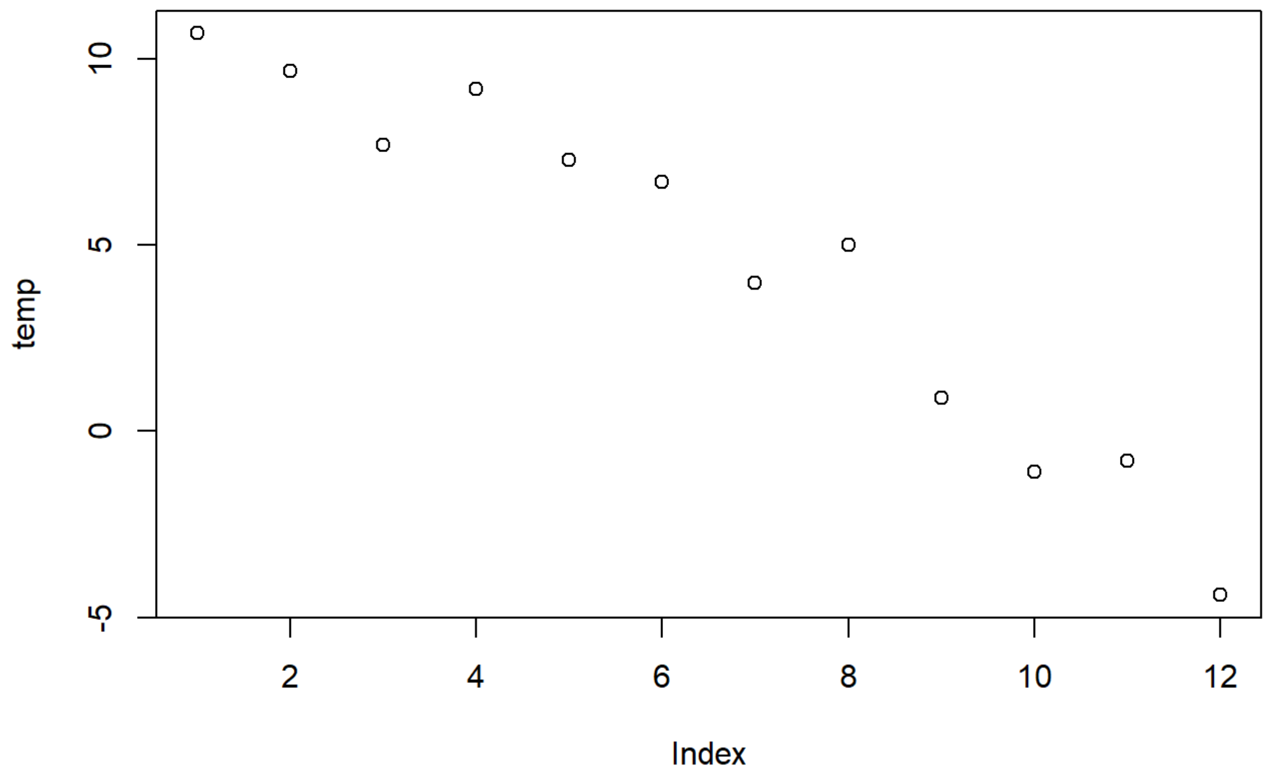

Figure 2.1: Temperature

```r
plot(elev)
```

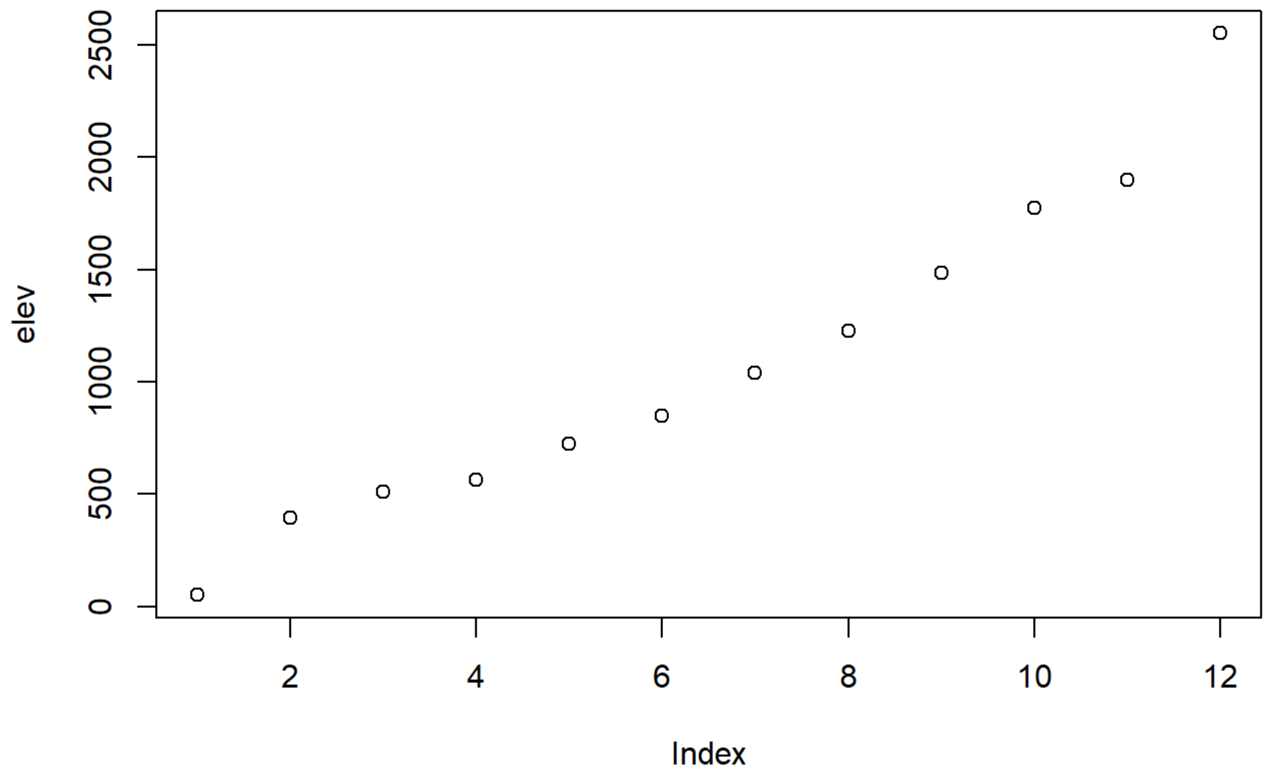

Figure 2.2: Elevation

```r
plot(lat)
```

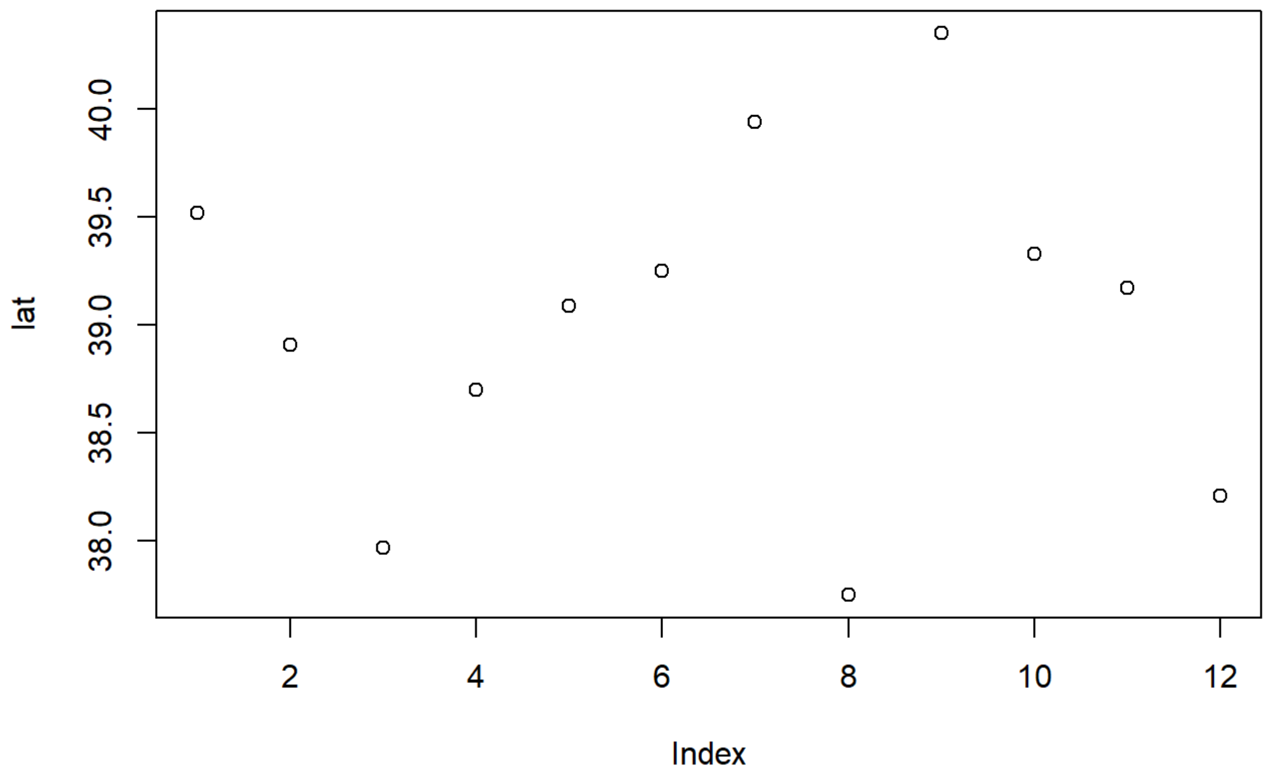

Figure 2.3: Latitude

Then plot as a scatterplot

```
plot(elev,temp)
```

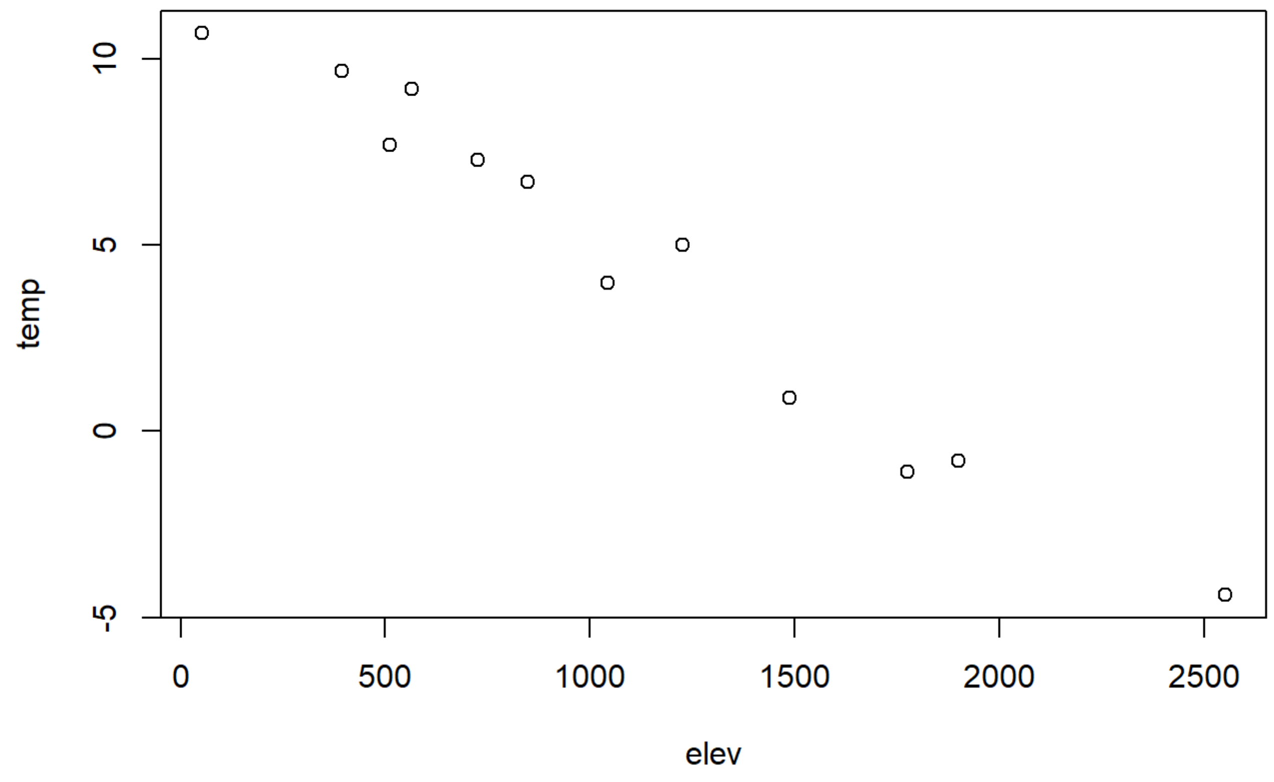

Figure 2.4: Temperature~Elevation

2.6.1.5 Named indices

Vector indices can be named.

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

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

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

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

```
str(codes)
```

```
##  Named num [1:3] 380 124 818
##  - attr(*, "names")= chr [1:3] "italy" "canada" "egypt"
```

Why? I guess so you can refer to observations by name instead of index. The following are equivalent:

```
codes[1]
```

```
## italy
##   380
```

```
codes["italy"]
```

```
## italy
##   380
```

2.6.2 Lists

Lists can be heterogeneous, with multiple class types. Lists are actually used a lot in R, but we won’t see them for a while.

2.6.3 Matrices

Vectors are commonly used as a column in a matrix (or as we’ll see, a data frame), like a variable

```
temp <- c(10.7, 9.7, 7.7, 9.2, 7.3, 6.7, 4.0, 5.0, 0.9, -1.1, -0.8, -4.4)
elev <- c(52, 394, 510, 564, 725, 848, 1042, 1225, 1486, 1775, 1899, 2551)
lat <- c(39.52, 38.91, 37.97, 38.70, 39.09, 39.25, 39.94, 37.75, 40.35, 39.33, 39.17, 38.2
```

Building a matrix from vectors as columns

```
sierradata <- cbind(temp, elev, lat)
class(sierradata)
```

```
## [1] "matrix" "array"
```

2.6.3.1 Dimensions for arrays and matrices

Note: a matrix is just a 2D array. Arrays have 1, 3, or more dimensions.

```
dim(sierradata)
```

```
## [1] 12  3
```

```
a <- 1:12
dim(a) <- c(3, 4)   # matrix
class(a)
```

```
## [1] "matrix" "array"
```

```
dim(a) <- c(2,3,2)  # 3D array
class(a)
```

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

```
dim(a) <- 12        # 1D array
class(a)
```

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

```
b <- matrix(1:12, ncol=1)  # 1 column matrix is allowed
```

2.6.4 Data frames

A data frame is a database with fields (as vectors) with records (rows), so is very important for data analysis and GIS. They're kind of like a spreadsheet with rules (first row is field names, fields all one type). So even though they're more complex than a list, we use them so frequently they become quite familiar [whereas I continue to find lists confusing, especially when discovering them as what a particular function returns.]

```
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
```

Creating a data frame out of a matrix

```
mydata <- as.data.frame(sierradata)
plot(data = mydata, x = elev, y = temp)
```

```
## Warning in plot.window(...): "data" is not a graphical parameter
## Warning in plot.xy(xy, type, ...): "data" is not a graphical parameter
```

```
## Warning in axis(side = side, at = at, labels = labels, ...): "data" is not a
## graphical parameter
```

```
## Warning in axis(side = side, at = at, labels = labels, ...): "data" is not a
## graphical parameter
```

```
## Warning in box(...): "data" is not a graphical parameter
```

```
## Warning in title(...): "data" is not a graphical parameter
```

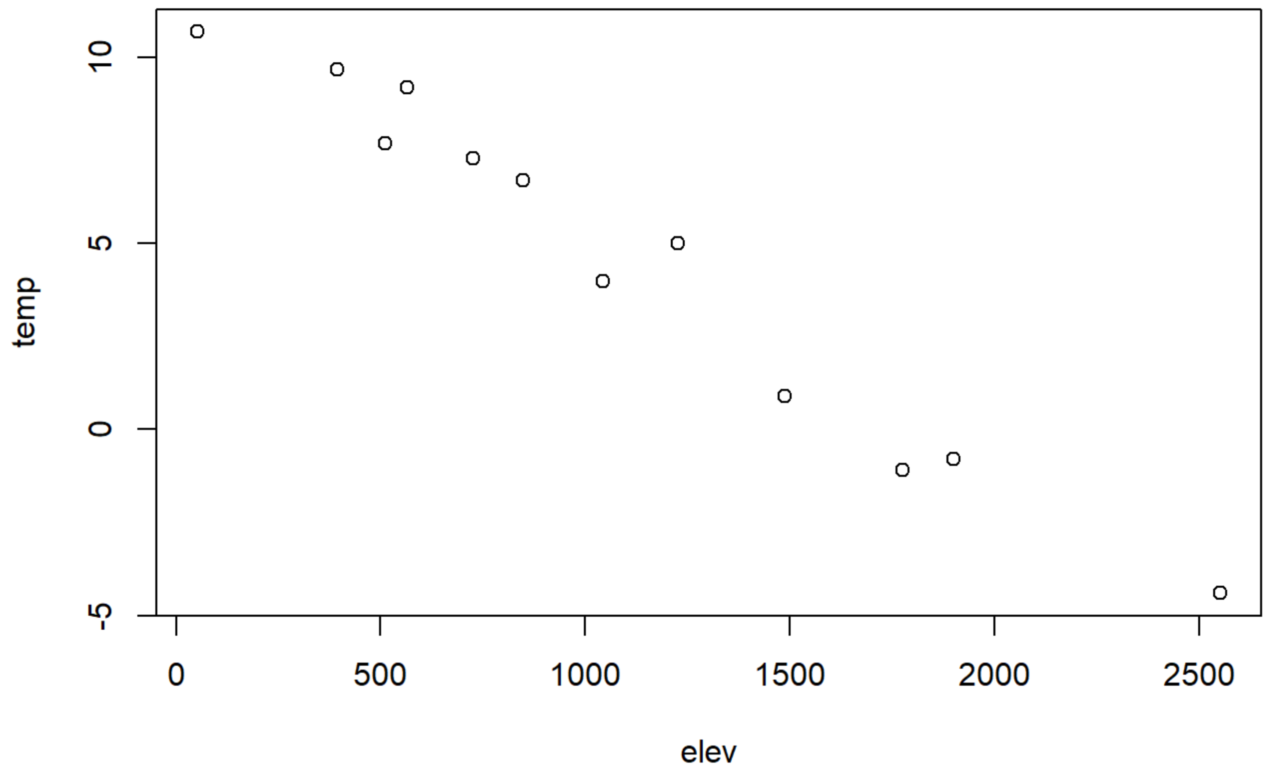

Figure 2.5: Temperature~Elevation

Read a data frame from a CSV

```
TRI87 <- read_csv("data/TRI_1987_BaySites.csv")
```

```
## Parsed with column specification:
## cols(
##   TRI_FACILITY_ID = col_character(),
##   count = col_double(),
##   FACILITY_NAME = col_character(),
##   COUNTY = col_character(),
##   air_releases = col_double(),
##   fugitive_air = col_double(),
##   stack_air = col_double(),
##   LATITUDE = col_double(),
##   LONGITUDE = col_double()
## )
```

```
TRI87
```

```
## # A tibble: 335 x 9
##    TRI_FACILITY_ID count FACILITY_NAME COUNTY air_releases fugitive_air
##    <chr>           <dbl> <chr>         <chr>         <dbl>        <dbl>
##  1 91002FRMND585BR     2 FORUM IND     SAN M~         1423         1423
##  2 92052ZPMNF2970C     1 ZEP MFG CO    SANTA~          337          337
##  3 93117TLDYN3165P     2 TELEDYNE MEC  SANTA~        12600        12600
##  4 94002GTWSG477HA     2 MORGAN ADVAN~ SAN M~        18700        18700
##  5 94002SMPRD120SE     2 SEM PRODS INC SAN M~         1500          500
##  6 94025HBLNN151CO     2 HEUBLEIN INC  SAN M~          500            0
##  7 94025RYCHM300CO    10 TE CONNECTIV~ SAN M~       144871        47562
##  8 94025SNFRD990OB     1 SANFORD META~ SAN M~         9675         9675
##  9 94026BYPCK3575H     2 BAY PACKAGIN~ SAN M~        80000        32000
## 10 94026CDRSY3475E     2 CDR SYS CORP  SAN M~       126800            0
## # ... with 325 more rows, and 3 more variables: stack_air <dbl>,
## #   LATITUDE <dbl>, LONGITUDE <dbl>
```

Sort, Index, & Max/Min

```
TRI87 <- read_csv("data/TRI_1987_BaySites.csv")
```

```
## Parsed with column specification:
## cols(
##   TRI_FACILITY_ID = col_character(),
##   count = col_double(),
##   FACILITY_NAME = col_character(),
##   COUNTY = col_character(),
##   air_releases = col_double(),
##   fugitive_air = col_double(),
##   stack_air = col_double(),
##   LATITUDE = col_double(),
##   LONGITUDE = col_double()
## )
```

```
head(sort(TRI87$air_releases))
```

```
## [1]  2  5  5  7  9 10
```

```
index <- order(TRI87$air_releases)
head(TRI87$FACILITY_NAME[index])   # displays facilities in order of their air releases
```

```
## [1] "AIR PRODUCTS MANUFACTURING CORP"
## [2] "UNITED FIBERS"
## [3] "CLOROX MANUFACTURING CO"
## [4] "ICI AMERICAS INC WESTERN RESEARCH CENTER"
## [5] "UNION CARBIDE CORP"
## [6] "SCOTTS-SIERRA HORTICULTURAL PRODS CO INC"
```

```
i_max <- which.max(TRI87$air_releases)
TRI87$FACILITY_NAME[i_max]   # was NUMMI at the time
```

```
## [1] "TESLA INC"
```

2.6.5 Factors

Factors are vectors with predefined values - Normally used for categorical data. - Built on an *integer* vector - Levels are the set of predefined values.

```
fruit <- factor(c("apple", "banana", "orange", "banana"))
fruit   # note that levels will be in alphabetical order
```

```
## [1] apple  banana orange banana
## Levels: apple banana orange
```

```
class(fruit)
```

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

```
typeof(fruit)
```

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

An equivalent conversion:

```
fruitint <- c(1, 2, 3, 2) # equivalent conversion
fruit <- factor(fruitint, labels = c("apple", "banana", "orange"))
str(fruit)
```

```
##  Factor w/ 3 levels "apple","banana",..: 1 2 3 2
```

2.6.5.1 Categorical Data and Factors

While character data might be seen as categorical (e.g. “urban”, “agricultural”, “forest” land covers), to be used as categorical variables they must be made into factors.

```
grain_order <- c("clay", "silt", "sand")
grain_char <- sample(grain_order, 36, replace = TRUE)
grain_fact <- factor(grain_char, levels = grain_order)
grain_char
```

```
##  [1] "silt" "clay" "silt" "clay" "clay" "clay" "sand" "clay" "clay" "sand"
## [11] "clay" "sand" "sand" "clay" "clay" "silt" "clay" "clay" "clay" "clay"
## [21] "silt" "silt" "clay" "clay" "clay" "clay" "sand" "silt" "silt" "silt"
## [31] "clay" "silt" "silt" "sand" "clay" "sand"
```

```
grain_fact
```

```
##  [1] silt clay silt clay clay clay sand clay clay sand clay sand sand clay clay
## [16] silt clay clay clay clay silt silt clay clay clay clay sand silt silt silt
## [31] clay silt silt sand clay sand
## Levels: clay silt sand
```

To make a categorical variable a factor:

```
fruit <- c("apples", "oranges", "bananas", "oranges")
farm <- c("organic", "conventional", "organic", "organic")
ag <- as.data.frame(cbind(fruit, farm))
ag$fruit <- factor(ag$fruit)
ag$fruit
```

```
## [1] apples  oranges bananas oranges
## Levels: apples bananas oranges
```

Factor example

```
sierraFeb$COUNTY <- factor(sierraFeb$COUNTY)
str(sierraFeb$COUNTY)
```

```
##  Factor w/ 21 levels "Amador","Butte",..: 20 14 7 12 12 2 19 11 2 19 ...
```

2.7 Programming and Logic

Given the exploratory nature of the R language, we sometimes forget that it provides significant capabilities as a programming language where we can solve more complex problems by coding procedures and using logic to control the process and handle a range of possible scenarios.

Programming languages are used for a wide range of purposes, from developing operating systems built from low-level code to high-level *scripting* used to run existing functions in libraries. R and Python are commonly used for scripting, and you may be familiar with using arcpy to script ArcGIS geoprocessing tools. But whether low- or high-level, some common operational structures are used in all computer programming languages:

- Conditional operations: *If* a condition is true, do this, and maybe otherwise do something *else*.

  ```
  if x!=0 {print(1/x)} else {print("Can't divide by 0")}
  ```

- Loops

  ```
  for(i in 1:10) print(paste(i, 1/i))
  ```

- Functions (defining your own then using it in your main script)

```
turnright <- function(ang){(ang + 90) %% 360}
turnright(c(260, 270, 280))
```

```
## [1] 350   0  10
```

Free-standing scripts

As we move forward, we'll be wanting to develop complete, free-standing scripts that have all of the needed libraries and data. Your scripts should stand on their own. One example of this that may seem insignificant is using print() statements instead of just naming the object or variable in the console. While that is common in exploratory work, we need to learn to create free-standing scripts.

However, "free standing" still allows for loading libraries of functions we'll be using. We're still talking about high-level (*scripting*), not low-level programming, so we can depend on those libraries that any user can access by installing those packages. If we develop our own packages, we just need to provide the user the ability to install those packages.

2.7.1 Subsetting with logic

We'll use a package that includes data from Irizarry, Rafael (2020) *Introduction to Data Science* section 2.13.1.

Identify all states with murder rates ≤ that of Italy.

```
library(dslabs)
data(murders)
murder_rate <- murders$total / murders$population * 100000
i <- murder_rate <= 0.71
murders$abb[i]
```

```
## [1] "HI" "IA" "NH" "ND" "VT"
```

which

```
library(readr)
TRI87 <- read_csv("data/TRI_1987_BaySites.csv")
```

```
## Parsed with column specification:
## cols(
##   TRI_FACILITY_ID = col_character(),
##   count = col_double(),
##   FACILITY_NAME = col_character(),
##   COUNTY = col_character(),
##   air_releases = col_double(),
##   fugitive_air = col_double(),
##   stack_air = col_double(),
##   LATITUDE = col_double(),
##   LONGITUDE = col_double()
## )
```

```
i <- which(TRI87$air_releases > 1e6)
TRI87$FACILITY_NAME[i]
```

```
## [1] "VALERO REFINING CO-CALI FORNIA BENICIA REFINERY"
## [2] "TESLA INC"
## [3] "TESORO REFINING & MARKETING CO LLC"
## [4] "HGST INC"
```

%in%

```
i <- TRI87$COUNTY %in% c("NAPA","SONOMA")
TRI87$FACILITY_NAME[i]
```

```
## [1] "SAWYER OF NAPA"
## [2] "BERINGER VINEYARDS"
## [3] "CAL-WOOD DOOR INC"
## [4] "SOLA OPTICAL USA INC"
## [5] "KEYSIGHT TECHNOLOGIES INC"
## [6] "SANTA ROSA STAINLESS STEEL"
## [7] "OPTICAL COATING LABORATORY INC"
## [8] "MGM BRAKES"
## [9] "SEBASTIANI VINEYARDS INC, SONOMA CASK CELLARS"
```

2.7.2 Apply functions

There are many apply functions in R, and they largely obviate the need for looping. For instance:

- `apply` derives values at margins of rows and columns, e.g. to sum across rows or down columns

```
# matrix apply – the same would apply to data frames
matrix12 <- 1:12
dim(matrix12) <- c(3,4)
rowsums <- apply(matrix12, 1, sum)
colsums <- apply(matrix12, 2, sum)
sum(rowsums)
```

```
## [1] 78
```

```
sum(colsums)
```

```
## [1] 78
```

```
zero <- sum(rowsums) - sum(colsums)
matrix12
```

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

Apply functions satisfy one of the needs that spreadsheets are used for. Consider how of ten you use sum, mean or similar functions in Excel.

`sapply`

sapply applies functions to either:

- all elements of a vector – unary functions only

```
sapply(1:12, sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278 3.316625 3.464102
```

- or all variables of a data frame (not a matrix), where it works much like a column-based apply (since variables are columns) but more easily interpreted without the need of specifying columns with 2:

```
sapply(cars,mean)  # same as apply(cars,2,mean)
```

```
## speed  dist
## 15.40 42.98
```

```
temp02 <- c(10.7,9.7,7.7,9.2,7.3,6.7,4.0,5.0,0.9,-1.1,-0.8,-4.4)
temp03 <- c(13.1,11.4,9.4,10.9,8.9,8.4,6.7,7.6,2.8,1.6,1.2,-2.1)
sapply(as.data.frame(cbind(temp02,temp03)),mean) # has to be a data frame
```

```
##   temp02   temp03
## 4.575000 6.658333
```

While various `apply` functions are in base R, the purrr package takes these further. See: purrr cheat sheet

2.8 Exercises

1. Assign variables for your name, city, state and zip code, and use `paste()` to combine them, and assign them to the variable `me` . What is the class of `me` ?
2. Knowing that trigonometric functions require angles (including azimuth directions) to be provided in radians, and that degrees can be converted into radians by dividing by 180 and multiplying that by pi, derive the sine of 30 degrees with an R expression. (Base R knows what pi is, so you can just use `pi`)
3. If two sides of a right triangle on a map can be represented as dX and dY and the direct line path between them c, and the coordinates of 2 points on a map might be given as $(x1, y1)$ and $(x2, y2)$, with $dX = x2 - x1$ and $dY = y2 - y1$, use the Pythagorean theorem to derive the distance between them and assign that expression to c.
4. You can create a vector uniform random numbers from 0 to 1 using `runif(n=30)` where n=30 says to make 30 of them. Use the `round()` function to round each of the values, and provide what you created and explain what happened.
5. Create two vectors of 10 numbers each with the c() function, then assigning to x and y. Then plot(x,y), and provide the three lines of code you used to do the assignment and plot.
6. Change your code from #5 so that one value is NA (entered simply as `NA` , no quotation marks), and derive the mean value for x. Then add `,na.rm=T` to the parameters for `mean()` . Also do this for y. Describe your results and explain what happens.

7. Create two sequences, `a` and `b`, with `a` all odd numbers from 1 to 99, `b` all even numbers from 2 to 100. Then derive c through vector division of `b/a`. Plot a and c together as a scatterplot.

8. Build the sierradata data frame from the data at the top of the **Matrices** section, also given here:

```
temp <- c(10.7, 9.7, 7.7, 9.2, 7.3, 6.7, 4.0, 5.0, 0.9, -1.1, -0.8, -4.4)
elev <- c(52, 394, 510, 564, 725, 848, 1042, 1225, 1486, 1775, 1899, 2551)
lat <- c(39.52, 38.91, 37.97, 38.70, 39.09, 39.25, 39.94, 37.75, 40.35, 39.33, 39.17, 38.2
```

Create a data frame from it using the same steps, and plot temp against latitude.

9. From the `sierradata` matrix built with `cbind()`, derive colmeans using the `mean` parameter on the columns `2` for `apply()`.

10. Do the same thing with the sierra data data frame with `sapply()`.