Introduction to R Tutorial

Kevin Shook and R. Chlumsky

June 5, 2022

Introduction to R Tutorial

This tutorial is intended to provide some background training on using **R**

- Getting help in R
- Basic R Syntax
- Data types
- Flow control
- Creating plots (base R)
- Advanced topics
- Challenge exercises

How this document works

This document was generated using **R Markdown**. This is a great tool for code transparency and data analysis, because the code blocks, code outputs, and your comments are "knit" into a single document!

This document has a number of headings, text blocks (like this one), and code chunks that are run as the document is compiled. This allows you to see the script and output in one document (plus the code gets run and checked for errors as the document is made, which helps to reduce typos in the document code sections).

If you want to run one line in the chunk you can place your cursor in one line and press CTRL+ENTER to run that line (CMD+ENTER for Mac). Here's an example with just one line

```
# this is a comment in R. It is preceded by an octothorpe (or pound symbol,
# or hashtag, depending on your generation).

print("This is a line of code with text in it, preceeded by ## in the output printout.")
```

```
## [1] "This is a line of code with text in it, preceeded by ## in the output printout."
```

Note: when you see the pound sign (#, aka octothorpe or "hash") next to a line of code, what follows to the right is a text comment that does not affect the code. Commenting is an important part of remembering what complex lines of code are meant to do!

Where are you working?

One of the most common issues that new R users encounter is understanding the working directory. Your working directory is the folder location where your R session is operating from, and is where all files are assumed to be (and written to) unless otherwise specified. You can check your working directory with the command `getwd()`. You can also change the working directory with `setwd()`. Alternatively, if you are working in RStudio, your Files window can be used to view Files and navigate the working directory (same Pane where you can find Plots/Packages/Help/Viewer). If you click on *Files –> More*, you can see options to set or go to your working directory. Take a moment to check this before proceeding with this document.

Getting Help in R

One of the first things to know is where to go for help. All functions that you come across in **R** have documentation and examples to help you understand them. They can be accessed with either the `help()` function (e.g. `help(rep)`), or by using a question mark in front of the function name (e.g. `?rep`).

```
# get help on command, like the variance function 'rep()'
help(rep)
?rep
```

If you don't quite know the name of a function, two "?" question marks followed by a guess will search for all the related topics.

To search for all the help on calculating the variance, try this

```
?? variance
```

Finally, if you place your cursor within the function name in your chunk and press **F1** on your keyboard, the documentation will again come up.

If you are using **RStudio** , the help information should appear on the right side of your screen in the Help menu. If you don't see the Files/Plots/Help menu, then at the top menu, click on *View –> Panes –> Show All Panes*, and your layout should be restored.

For more resources, check out the CRAN website (https://cran.r-project.org/) for information, documentation, etc.

It's hard to Google "R". To search for help on **R** topics use https://rseek.org/.

Basic R Syntax

Basic Arithmetic

Let's try some basic arithmetic. The `*` symbol is used for multiplication, `/` is used for division, and `^` is for exponentiation (raising to powers).

```
1 + 1
```

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

```
3 - 1
```

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

```
2 * 2
```

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

```
4 / 3
```

```
## [1] 1.333333
```

```
3 ^ 2
```

```
## [1] 9
```

Variables

Like all programming languages, **R** allows you to store values in named variables. **R** is not strict in its definition of data types, so you can assign and re-assign the data types to variables at your whim.

The symbol (<-) *assigns* the value from one end of the arrow to the object at the point of the arrow. This method is directional, but you can also use the equal sign (=) as a left-only equivalent. Note that using <- is considered to be better style in **R** (the reasons are complicated), so you should be using it and that the <- symbol is only used to assign values to variables, while the = sign is also used to specify parameter values to functions. In *Rstudio*, you can insert <- by pressing [Alt][-].

Variable names have to begin with a letter, and can't include spaces or other sybols, other than numbers, periods (.) and underscores (_). There are also some reserved names that you can't use: `if else repeat while function for in next break TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_` as they are used by **R** itself.

R is sensitive to case. So the variables `my_variable` and `My_variable` are completely different.

```
my_variable = 1
My_variable <- 5
my_variable
```

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

```
My_variable
```

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

Coming up with good names for variables can be difficult. This document discusses some of the naming conventions that are in use: https://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf

Functions

R uses functions, which are pre-written pieces of code, for everything. Functions are called using round brackets (parentheses). Inside the brackets you specify the values (often called arguments) used by the function. In this example, the `rnorm` function is called, followed by the `mean` and `sd` (standard deviation) functions.

```
a <- rnorm(n = 20, mean = 0, sd = 2) # sample a normal distribution of 20 values with mean = 0, sd = 2
mean(a)                              # calculates the mean of values in vector a
```

```
## [1] -0.3197487
```

```
sd(a)                                # calculates standard deviation of a
```

```
## [1] 1.800592
```

You can create your own **R** functions, which is very useful as it makes your work even more reproducible. The functions can be stored in an **R** package

Checking data types

Here are some commands to check what type of data you are dealing with, and help you convert between different types.

```
# what class of variable is it?
b <- 6
class(b)
```

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

```
# what 'typeof' value is b?
typeof(b) # 'double' is the type of number (double precision)
```

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

```
b <- "hello, world" # change the value assigned to b to a character value
b
```

```
## [1] "hello, world"
```

```
typeof(b) # this has now changed from numeric/double
```

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

Convert between data types

```
as.numeric("501") + 1  # convert quoted string to numeric and add 1
```

```
## [1] 502
```

```
round(5.6)             # round to integer
```

```
## [1] 6
```

```
floor(5.6)             # round down to integer
```

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

```
as.character(502)      # convert numeric to character
```

```
## [1] "502"
```

Now that we've started re-assigning values to variables, it's useful to recall their new values. On the upper-right portion of your screen in RStudio, the **Environment** tab shows the values of all user-defined variables and functions.

Working with Text

R can read, write, and modify strings of text. A character string is any sequence of any characters. Character strings are denoted by being enclosed in either single `''` or double `""` quotes.

```
a <- "This is a character string"
b <- 'This is another one'
a
```

```
## [1] "This is a character string"
```

```
b
```

```
## [1] "This is another one"
```

You can't do arithmetic with character strings, but they are still very useful. Many types of data, from names of places to names of months are characters. **R** has many functions for working with character data. You can sort character strings, slice them into pieces, or combine them together.

You can combine strings together using the `paste` function, which concatenates them together. Note that by default `paste` will add a space between character strings. If you want to override this or use another separator, you have to specify it.

```
day_num <- "05"
month_num <- "06"
year_num <- "2022"
date <- paste(day_num, month_num, year_num, sep = "/")
date
```

```
## [1] "05/06/2022"
```

Dates

R is excellent at working with dates (and times). There is a built-in `date` data type. You can convert a character date to a `date` type using the function `as.Date()`. `date` values can be used for arithmetic.

If you don't tell `as.Date` the format, it will guess. However the guess may be wrong. You can specify the format using the symbols shown below, which represent the day, month and year, as well as the "/" symbol which separates them. If you want to see all of the available formatting symbols, use the command `?strptime`.

```
day1 <- as.Date("1900-01-01")
day2 <- as.Date("05/06/2022", format = "%d/%m/%Y")
```

Vectors, data frames, lists, arrays and matrices

A single valued variable, such as the ones above (also known as a scalar), is called an *atom* . Multiple atoms come together to form a set of values, called a **vector**. Multiple vectors form a **matrix** or **data frame**.

`c()` is the concatenate function, taking multiple atoms separated by commas and creating a new vector. We saw above how we can use addition or multiplication on an atom. We can also perform a single operation on all values in a vector. Let's perform apply arithmetic on the new vector `a`.

```
a <- c(1,2,3,4,5)
a
```

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

Vectorized functions

Most functions in **R** are *vectorized* meaning that they are applied over a whole vector. For example the `paste` function works this way. In this example, the paste function is applied to each value in the vector `dates`. Because there is only one value of `year` the value is *recycled* for each value in `dates`. This is really powerful.

```
year <- "2020"
dates <- c("01/01", "02/01", "03/01", "04/01", "05/01")
complete_dates <- paste(dates, year, sep = "/")
complete_dates
```

```
## [1] "01/01/2020" "02/01/2020" "03/01/2020" "04/01/2020" "05/01/2020"
```

The same applies to arithmetic

```
a <- seq(1, 10)
a
```

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

```
b <- sqrt(a)       # uses built-in square root function on all values of a
b
```

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

R has built-in functions that are also helpful in generating vectors, matrices, data frames, etc.

Subsetting Vectors

It's important to be able to subset vectors, to extract specific values. To do this, we use the square brackets "[ ]". They are not to be confused with () or {}, which each have their own special uses, as discussed above. The `length` function returns the length of a vector.

```
# set up a vector
# sequence from 1 to 30, jumping by a value of 2 (instead of the default 1)
a <- seq(1, 30, by = 2)
a
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29
```

```
length(a) # get the length of vector a (number of values)
```

```
## [1] 15
```

```
# subset by location
a[1] # first value in the vector
```

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

```
a[c(1,2)] # first and second value in the vector
```

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

```
a[c(1,2,3)] # note the use of c() to specify multiple values
```

```
## [1] 1 3 5
```

```
a[1:3]
```

```
## [1] 1 3 5
```

```
a[-1]  # all values without the first one
```

```
##  [1]  3  5  7  9 11 13 15 17 19 21 23 25 27 29
```

```
a[c(-1, -2, -3)] # omits the first 3 slements
```

```
##  [1]  7  9 11 13 15 17 19 21 23 25 27 29
```

Using Booleans

Boolean operators refer to greater than (>), less than (<), or equal to (==). Other operators are greater than *or* equal to (>=), and not equal to (!=)

```
1 > 3
```

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

```
1 < 3
```

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

```
1 <= 3 # less than or equal to
```

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

```
1 != 3
```

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

```
1 == 1
```

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

Note: <= looks very similar to <- but they are nothing alike.

We can also apply these basic operations with vectors, as shown below. In **R**, this will automatically apply the function > for all elements provided.

```
c <- seq(1,10)
c > 5
```

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

You can use Booleans to perform subsetting operations (i.e. extract values by conditions).

```
# set up vectors
a <- seq(1,10,2)
b <- seq(2,10,2)
a
```

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

```
b
```

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

```
# subset vectors
a > 5 # which values are greater than 5?
```

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

```
a[ a > 5] # same as above, but returns the values of a that meet the criteria
```

```
## [1] 7 9
```

```
a[(a >= 3) & (a < 8)] # multiple conditions for subsetting, take values >=3 and <8 only
```

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

```
subset(a, a >= 3 & a < 8) # same operation with the subset function
```

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

```
a[which(a >= 3 & a < 8)] # same operation as above with the which function
```

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

```
which( a >= 3 & a < 8) # returns the index values *which* meet the criteria specified
```

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

```
any(a %in% b)  # check if any values of a exist in b (FALSE, since we have evens and odds)
```

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

Data frames

Data frames are the most useful data structure in **R** and are created by assembling vectors. Each vector becomes a column in the data frame, and can be any data type, which is what makes data frames so useful. The command `data.frame` is used to construct a data frame from individual vectors. Note that all of the vectors must have the same number of values. The function `length()` is useful to get the number of values in a vector.

In this example we create 2 vectors, one containing dates and the other containing numeric values, and put them in the same data frame.

```
date <- seq(day1, day2, by = 1)     # a sequence between the 2 dates defined earlier
value <- runif(length(date))        # random numbers of the same length as 'date'

df <- data.frame(date, value)       # assemble the data frame
summary(df)                         # summarize the values in the data frame
```

```
##       date                value
##  Min.   :1900-01-01   Min.   :0.0000164
##  1st Qu.:1930-08-10   1st Qu.:0.2494015
##  Median :1961-03-19   Median :0.4947645
##  Mean   :1961-03-19   Mean   :0.4982572
##  3rd Qu.:1991-10-27   3rd Qu.:0.7488922
##  Max.   :2022-06-05   Max.   :0.9999673
```

If you click on `df` in the **Environment** tab, you can see the values. This is *really* useful.

Subsetting data frames The same principles that work on selecting and subsetting vectors also work with data frames (and arrays and matrices). Note that the `length()` function is not useful as it just returns the number of variables(columns) in a data frame.

```
length(df)
```

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

```
dim(df)    # returns [number of rows, number of columns]
```

```
## [1] 44716     2
```

```
nrow(df)   # number of rows
```

```
## [1] 44716
```

```
ncol(df)   # number of columns
```

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

Subsetting data frames, as with vectors, uses the square brackets "[ ]", but instead of one value, you list the desired row and column, separated by a comma. Note that the row is specified first, and the column second. Calling a row of a dataframe will return a mini dataframe with just the one row

```
df[2, 2]   # second row, second columns
```

```
## [1] 0.5685295
```

```
df[2,] # leaving an entry blank will return the entire row or column specified
```

```
##         date     value
## 2 1900-01-02 0.5685295
```

You can select any number of rows and/or columns

```
df[1:3, 2]                # rows 1-3, column 2
```

```
## [1] 0.4739756 0.5685295 0.0888453
```

```
df[4:8,]                  # rows 4-8, all columns
```

```
##         date      value
## 4 1900-01-04 0.41278780
## 5 1900-01-05 0.35480978
## 6 1900-01-06 0.27166330
## 7 1900-01-07 0.12179595
## 8 1900-01-08 0.08155765
```

```
df[c(1,3,5), 1]           # rows 1,3,5, column 1
```

```
## [1] "1900-01-01" "1900-01-03" "1900-01-05"
```

You can also subset data frames by specifying the column names using "$" The function `names()` is used to list the names in a data frame and to set them.

```
names(df)
```

```
## [1] "date"  "value"
```

```
df$date[1:3]
```

```
## [1] "1900-01-01" "1900-01-02" "1900-01-03"
```

```
df2 <- df
names(df2) <- c("DATE", "VALUE")
names(df2)
```

```
## [1] "DATE"  "VALUE"
```

```
names(df2)[1] <- "Date"
names(df2)
```

```
## [1] "Date"  "VALUE"
```

Examples of working with data frames The CSHShydRology package has some built in data that we can use when we load the package

```
library(CSHShydRology)
data()
```

R uses 'lazy-loading' so the data isn't loaded until you ask for it. You can see that the data frame has 2 columns, the date and the flowrate.

```
CAN01AD002 <- CAN01AD002
summary(CAN01AD002)
```

```
##       date                 flow
##  Min.   :1926-10-01   Min.   :  14.4
##  1st Qu.:1948-10-23   1st Qu.:  70.8
##  Median :1970-11-15   Median : 136.0
##  Mean   :1970-11-15   Mean   : 278.8
##  3rd Qu.:1992-12-07   3rd Qu.: 295.8
##  Max.   :2014-12-31   Max.   :4630.0
```

Because data frames hold more than one column of data, you can select values from one column based on another.

For example, this would calculate the mean flows between 2 dates. The data frame is subset by selecting rows where the `date` variable lies between the start and end dates. The mean flow can then be calculated using the `mean` function on the specified column.

```
start_date <- as.Date("2014-01-01", format = "%Y-%m-%d")
end_date <- as.Date("2014-12-31", format = "%Y-%m-%d")
selected <- CAN01AD002[CAN01AD002$date >= start_date & CAN01AD002$date <= end_date,]
summary(selected)
```

```
##       date                 flow
##  Min.   :2014-01-01   Min.   :  39.6
##  1st Qu.:2014-04-02   1st Qu.:  72.5
##  Median :2014-07-02   Median : 124.0
##  Mean   :2014-07-02   Mean   : 268.6
##  3rd Qu.:2014-10-01   3rd Qu.: 231.0
##  Max.   :2014-12-31   Max.   :2340.0
```

```
mean_flows <- mean(selected$flow)
cat("mean daily flow:", mean_flows, "m3/s")
```

```
## mean daily flow: 268.6088 m3/s
```

What if you wanted the maximum or minimum yearly values? One way would be to add a column for the year of each date. You can do this using the `format()` function. Because `format()` returns a character string, it's often more useful to convert it to a number, using the function `as.numeric()`.

```
CAN01AD002$year <- as.numeric(format(CAN01AD002$date, format = "%Y"))
summary(CAN01AD002)
```

```
##       date                 flow             year
##  Min.   :1926-10-01   Min.   :  14.4   Min.   :1926
##  1st Qu.:1948-10-23   1st Qu.:  70.8   1st Qu.:1948
##  Median :1970-11-15   Median : 136.0   Median :1970
##  Mean   :1970-11-15   Mean   : 278.8   Mean   :1970
##  3rd Qu.:1992-12-07   3rd Qu.: 295.8   3rd Qu.:1992
##  Max.   :2014-12-31   Max.   :4630.0   Max.   :2014
```

Having added the years, you can get the summary statistic for each year using the function `aggregate()`. The tilde (~) is used in **R** when you want to express a relationship between two variables. In this case we are saying that we want to aggregate the flows with respect to the years.

```
yearly_max <- aggregate(flow~year, data = CAN01AD002, FUN = "max")
yearly_max
```

```
##    year flow
## 1  1926  923
## 2  1927 1880
## 3  1928 2550
```

```
## 4  1929 2210
## 5  1930 2730
## 6  1931 1370
## 7  1932 1940
## 8  1933 3310
## 9  1934 2670
## 10 1935 2040
## 11 1936 2280
## 12 1937 1720
## 13 1938 1950
## 14 1939 3260
## 15 1940 2440
## 16 1941 3000
## 17 1942 3230
## 18 1943 2470
## 19 1944 2040
## 20 1945 2210
## 21 1946 1940
## 22 1947 3140
## 23 1948 1650
## 24 1949 1370
## 25 1950 1850
## 26 1951 2030
## 27 1952 1950
## 28 1953 1980
## 29 1954 2700
## 30 1955 2380
## 31 1956 1350
## 32 1957 1390
## 33 1958 3280
## 34 1959 1460
## 35 1960 2570
## 36 1961 3680
## 37 1962 1290
## 38 1963 2110
## 39 1964 1260
## 40 1965  691
## 41 1966 1620
## 42 1967 1860
## 43 1968 2520
## 44 1969 3570
## 45 1970 2660
## 46 1971 2520
## 47 1972 2480
## 48 1973 3680
## 49 1974 3680
## 50 1975 2230
## 51 1976 2660
## 52 1977 2760
## 53 1978 2710
## 54 1979 4130
## 55 1980 1550
## 56 1981 2270
## 57 1982 2830
```

```
## 58 1983 3790
## 59 1984 3030
## 60 1985 2190
## 61 1986 1740
## 62 1987 3090
## 63 1988 1240
## 64 1989 1390
## 65 1990 2350
## 66 1991 2410
## 67 1992 2860
## 68 1993 2320
## 69 1994 3030
## 70 1995 1460
## 71 1996 2970
## 72 1997 2860
## 73 1998 2460
## 74 1999 1370
## 75 2000 2210
## 76 2001 2650
## 77 2002 2420
## 78 2003 1740
## 79 2004 2220
## 80 2005 3130
## 81 2006 2080
## 82 2007 2410
## 83 2008 4630
## 84 2009 2660
## 85 2010 2530
## 86 2011 3200
## 87 2012 2520
## 88 2013 1930
## 89 2014 2340
```

Sorting

The best way to sort data (vectors and data frames) in **R** is using the `order` function. It sorts values, but returns their *order*, not the sorted values. It may seem strange, but is actually very powerful.

In this example, we will sort the annual flows from largest to smallest. What you are seeing is the *location* (i.e. its row) of each value in its column, in order from largest to smallest.

```
order(yearly_max$flow, decreasing = TRUE)  # increasing order is the default
```

```
##  [1] 83 54 58 36 48 49 44  8 33 14 17 86 22 80 62 59 69 16 71 67 72 57 52  5 53
## [26] 29  9 45 51 84 76 35  3 85 43 46 87 47 18 73 15 77 66 82 30 65 89 68 11 56
## [51] 50 79  4 20 75 60 38 81 10 19 26 28 13 27  7 21 88  2 42 25 61 78 12 23 41
## [76] 55 34 70 32 64  6 24 74 31 37 39 63  1 40
```

If you want to see the values arranged by largest to smallest, the command would be

```
yearly_max$flow[order(yearly_max$flow, decreasing = TRUE)]
```

```
##  [1] 4630 4130 3790 3680 3680 3680 3570 3310 3280 3260 3230 3200 3140 3130 3090
## [16] 3030 3030 3000 2970 2860 2860 2830 2760 2730 2710 2700 2670 2660 2660 2660
## [31] 2650 2570 2550 2530 2520 2520 2520 2480 2470 2460 2440 2420 2410 2410 2380
## [46] 2350 2340 2320 2280 2270 2230 2220 2210 2210 2210 2190 2110 2080 2040 2040
## [61] 2030 1980 1950 1950 1940 1940 1930 1880 1860 1850 1740 1740 1720 1650 1620
```

```
## [76] 1550 1460 1460 1390 1390 1370 1370 1370 1350 1290 1260 1240  923  691
```

If you want to see the years arranged by largest to smallest max flows, the command is

```
yearly_max$year[order(yearly_max$flow, decreasing = TRUE)]
```

```
##  [1] 2008 1979 1983 1961 1973 1974 1969 1933 1958 1939 1942 2011 1947 2005 1987
## [16] 1984 1994 1941 1996 1992 1997 1982 1977 1930 1978 1954 1934 1970 1976 2009
## [31] 2001 1960 1928 2010 1968 1971 2012 1972 1943 1998 1940 2002 1991 2007 1955
## [46] 1990 2014 1993 1936 1981 1975 2004 1929 1945 2000 1985 1963 2006 1935 1944
## [61] 1951 1953 1938 1952 1932 1946 2013 1927 1967 1950 1986 2003 1937 1948 1966
## [76] 1980 1959 1995 1957 1989 1931 1949 1999 1956 1962 1964 1988 1926 1965
```

Factors

R has an unusual data type, **factor**, which is useful for dealing with categorical data. Factors are only found is vectors, or data frame columns. Although factors look like characters, they are actually integers. Factors are useful for grouping other data values. In this example, we have 4 hydrometric stations whose status us either "A" (active) or "D" (discontinued). We use the `factor` function to assign factor levels to each status. The `summary` command shows the number of values for each level.

```
station <- c("05HF021", "05HG001", "05HG002", "05HG003")
status <- c("D", "A", "A", "D" )
status <- factor(status, levels = c("D", "A"))
station_df <- data.frame(station, status)
summary(station_df)
```

```
##    station          status
##  Length:4           D:2
##  Class :character   A:2
##  Mode  :character
```

You can select the values by treating them as characters

```
station_df[station_df$status == "D",]
```

```
##   station status
## 1 05HF021      D
## 4 05HG003      D
```

You can also sort them using the `order` function. Because we defined the levels of the factors to be "D" before "A", we can sort the values in this order. This is very useful when you want to override the default alphabetical sorting order on graphs.

```
station_df[order(station_df$status),]
```

```
##   station status
## 1 05HF021      D
## 4 05HG003      D
## 2 05HG001      A
## 3 05HG002      A
```

Lists

Lists are one of the most powerful ways of storing data in **R** because they can contain *any* type of data - including other lists!

A list is created with the command `list()`. You can specify the names of the list elements when it is created. In this example, the list contains characters, a numeric value and a data frame.

```
gauge_info <- list(number = "01AD002", name = "SAINT JOHN RIVER AT FORT KENT", prov = "NB",
                   drainage_area = 14700, annual_max_daily_Q = yearly_max)
names(gauge_info)
```

```
## [1] "number"             "name"               "prov"
## [4] "drainage_area"      "annual_max_daily_Q"
```

To add or extract values from a list, you can select the variable by its name (if it has one), or by its number in the list, using `[[]]` (double square brackets).

```
gauge_info[[1]]
```

```
## [1] "01AD002"
```

```
gauge_info[[3]]
```

```
## [1] "NB"
```

```
gauge_info$name
```

```
## [1] "SAINT JOHN RIVER AT FORT KENT"
```

Matrices and arrays

Because data frames are so useful and powerful, **R** uses arrays and matrices less frequently. However, they can still be useful and some functions will require or return arrays/matrices, so you need to know about them.

Things you need to know:

1. arrays and matrices can only contain 1 type of data
2. an array can have any number of dimensions, a matrix is only two-dimensional
3. some functions like `t()` only work on matrices, not arrays
4. you can convert 2d arrays and matrices to data frames and data frames with a single type to arrays and matrices.

The `matrix()` function creates a matrix. You can give all the matrix elements an initial value. Note that the values are assigned one column at a time.

```
# d <- matrix(base value for matrix, number of rows = 3, number of columns = 4)
a <- seq(1, 12)
d <- matrix(a, nrow = 3, ncol = 4)
d
```

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

```
typeof(d)
```

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

```
dim(d)
```

```
## [1] 3 4
```

R has all of the usual matrix manipulation and math functions. One of the most useful is `t()` which transposes the rows and columns.

```
t(d)
```

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

Flow control

It's often useful for **R** to control the order and number of statements which are executed. You do this with the `if` statement and loops. These statements can help you when you want to process a lot of data.

Branches (if statements)

The `if` statement consists of a test statement, which returns a Boolean value (`TRUE` or `FALSE`). If the result is `TRUE` then the following statment is executed. If it is `FALSE` then nothing is done, unless there is an `else` statement. In that case, the statement following the `else` would be executed. If you want the `if` to execute more than one line after the test, enclose the lines in braces `{}`.

```
a <- 5
if (a < 2) {
  print("a is less than 2!")
} else {
  print("a is not less than 2!")
}
```

```
## [1] "a is not less than 2!"
```

Here the printed statement indicates that "a is not less than 2", since we defined `a` with a value of 5. This same code would print the other statement if we instead gave `a` a value less than 2.

ifelse

Often you will want to apply an `if` to each element of a vector or a data frame column. The simplest way to do this is with the `ifelse` command, which has the syntax `ifelse(test, TRUE value, FALSE value)`. The test is applied to each element and either the TRUE value or the FALSE value is returned for each.

You can do very useful things with `ifelse` on dataframes. For example, you could classify the flows based on their value. In this example, the flows are tested to see if they are within the lowest decile and a comment "low flow" is added if they are.

```
decile <- quantile(CAN01AD002$flow, 0.1)
decile
```

```
##  10%
## 43.9
```

```
CAN01AD002$comment <- ifelse(CAN01AD002$flow < decile, "low flow", "")
head(CAN01AD002[which(CAN01AD002$comment == "low flow"),])
```

```
##           date flow year  comment
## 120 1927-01-28 42.5 1927 low flow
## 121 1927-01-29 39.4 1927 low flow
## 122 1927-01-30 42.5 1927 low flow
## 125 1927-02-02 42.5 1927 low flow
## 126 1927-02-03 40.5 1927 low flow
## 127 1927-02-04 39.4 1927 low flow
```

Loops

Loops give programming its power by repeating a task. No more manual repetition! (or at least less of it).

There are 3 types of loops in **R**.

- the **for** loop will repeat a task *for* the items it is given, then stop on its own
- the **while** loop will repeat a task *while* a given condition is true, and will break once that condition is no longer true
- the **repeat** loop will do just that until the loop is broken with a command such as **break**; note that the **break** command can be used to stop any of the loops above

```
# for loops
for (i in 1:3) {
  print(i)
}
```

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

```
# while loop
i = 1
while (i < 4) {
  print(sprintf("%i is still less than 4",i))
  i = i + 1
}
```

```
## [1] "1 is still less than 4"
## [1] "2 is still less than 4"
## [1] "3 is still less than 4"
```

```
# repeat loop
i = 1
repeat {
  print(i)
  if (i >= 3) {
    break
  }
  i = i + 1
}
```

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

Loops in **R** are generally considered to be slow. However, recent tests have shown that they have improved, and that they are sometimes actually faster than using the `apply` functions, which are often used instead of loops: https://lorentzen.ch/index.php/2022/02/19/avoid-loops-in-r-really/. That said, it's best to avoid loops if there is a built-in (vectorized) function which will do the same thing.

This example shows which method is faster to get the square root of a million numbers by calculating the execution time for each.

```
d1 <- as.numeric(seq(1, 1e6)) # create a vector of 1 to 1 million
d2 <- vector(length = 1e6)    # create empty vector to hold results in loop

# take the sqrt of each element in the vector
Sys.sleep(1) # pause for 1 second to let CPU settle
system.time(for (i in 1:length(d1)) {d2[i] <- sqrt(d1[i])})
```

```
##    user  system elapsed
##   0.061   0.000   0.061
```

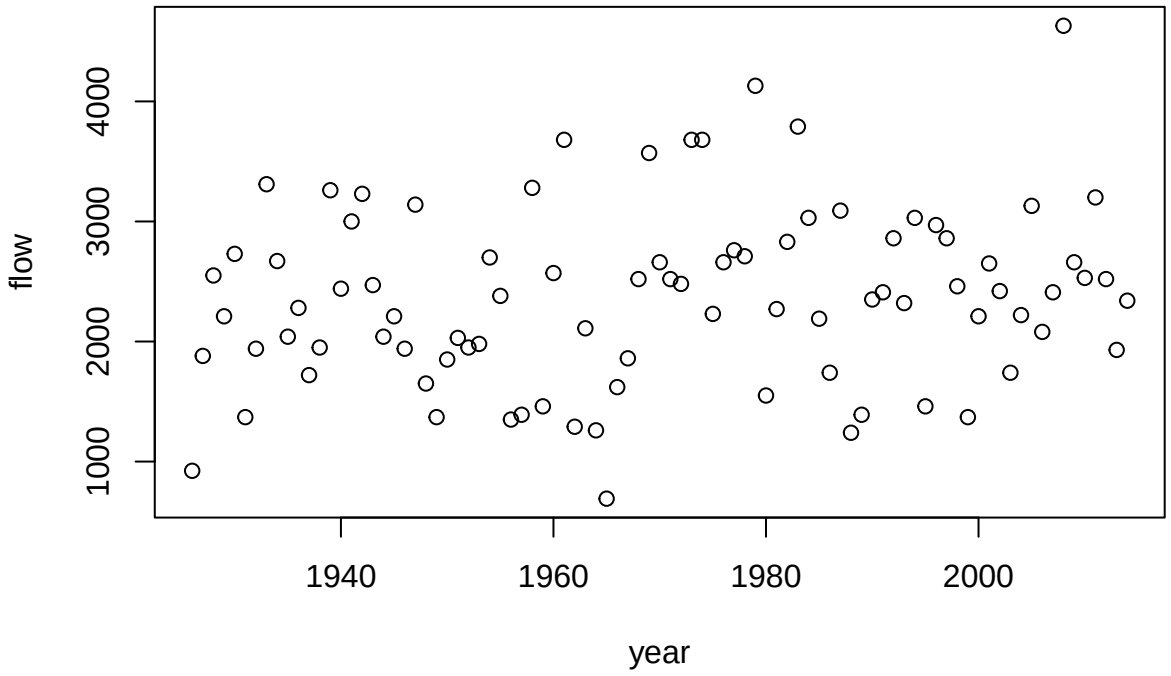
```
Sys.sleep(1) # pause for 1 second to let CPU settle
system.time(d3 <- sqrt(d1))
```

```
##    user  system elapsed
##   0.006   0.000   0.006
```

Simple plotting

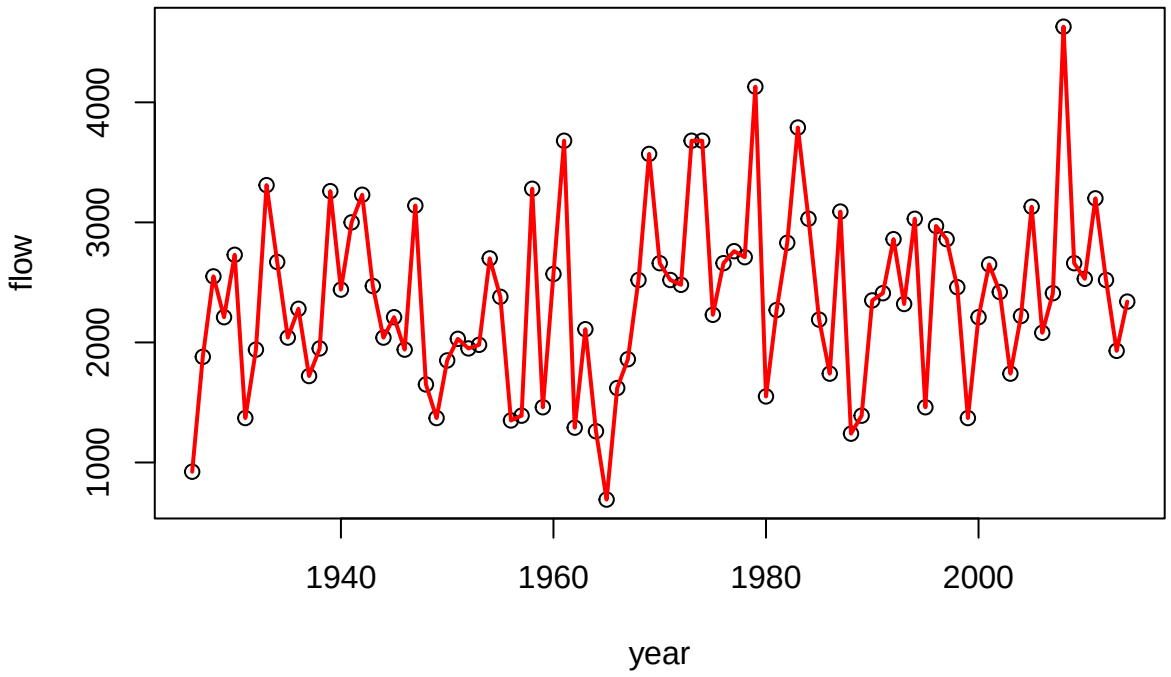
What fun is data if you can't visualize it? Let's plot the `yearly_max` data frame that we created above. Because we are going to use the first column as the x values and the second column as the y values, we can use the `plot` function defaults.

```
plot(yearly_max)
```



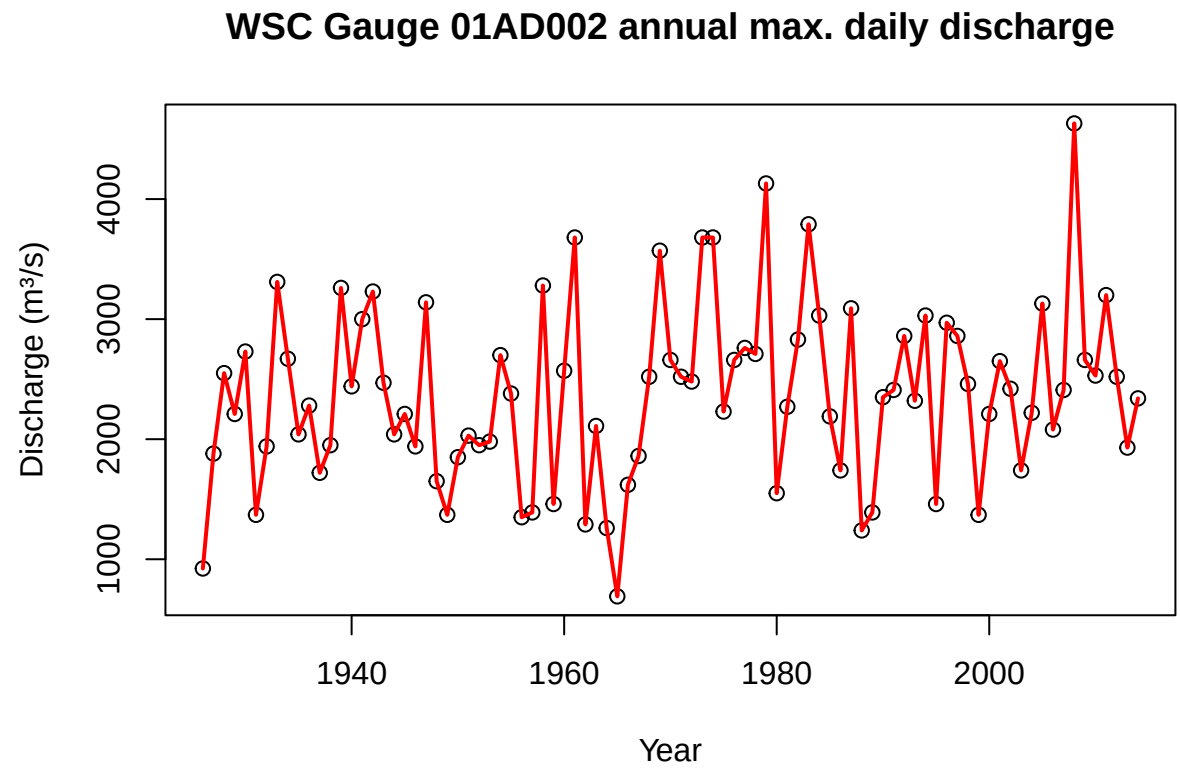
This plot is ok, but we can do better. We can add lines connecting the points and set its colour and width.

```
plot(yearly_max)
lines(yearly_max, col = 'red', lwd = 2)
```



What about the axis labels? We can override the defaults and add more descriptive label and a title.

```
plot(yearly_max, xlab = 'Year',ylab = 'Discharge (m³/s)',
     main = 'WSC Gauge 01AD002 annual max. daily discharge')
lines(yearly_max, col = 'red', lwd = 2)
```



Want to output your plot to a file? Try this out.

```
# start a png file of the given name, all subsequent plotting is directed to the file
png("../figures/01AD002_annual_max_daily.png")

plot(yearly_max, xlab='Year',ylab='Discharge (m³/s)',
     main='WSC Gauge 01AD002 annual max. daily discharge')
lines(yearly_max, col='red', lwd=2)

dev.off() # release focus from the png file, return to RStudio
```

File paths

R uses POSIX (unix) standard file paths, meaning that it uses the `/` symbol to delimit directories and file names. This works well on Linux and MacOS, but if you are using Windows you have two alternatives:
1. Use the **R** standard paths, e.g. `c:/projects/current_project/figures/`, or
2. Use double `\\` symbols, e.g. `c:\\projects\\current_project\\figures\\`

The example created the figure file in the `/figures` folder. The `..` refers to the parent directory of the directory you are currently working in. Later, we will show a better way of referring to your local directories.

Note that if you ever need to change the active directory you are in, you can use the **setwd** function. This may be needed if you want to work in a specific directory to conveniently load data during your session.

More Advanced Applications in R

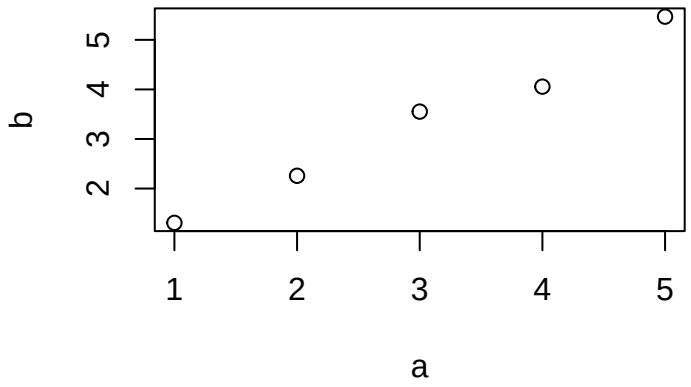
Let's take a look at some of the more useful packages and functions in R, as it relates to common tasks and usages of data.

S3 and S4 objects

R was derived from a closed-source program called "S-Plus". One of the features that **R** inherited is that functions can create their own types of objects, which have their own ways of being formatted for printing and plotting. Creating these objects (of types S3 and S4) is a *very* advanced topic. For now, you just need to know that they exist!

When you do a linear regression of one variable against another (using the `lm` function), the result is an S3 object. So when you `print` and `plot` the regression, you get customized outputs. Note that because the `plot` function returns 4 different plots, you will be prompted to hit [Return] to see each one (if you type the command at the console) or to click on the figures (if you run the chunk).

```
set.seed(100)       # set random seed so will always get same result (useful for teaching)
a <- c(1, 2, 3, 4, 5)
b <- a + runif(5)   # adds randomness
plot(a, b)
```




```
fit <- lm(b~a)
print(fit)
```

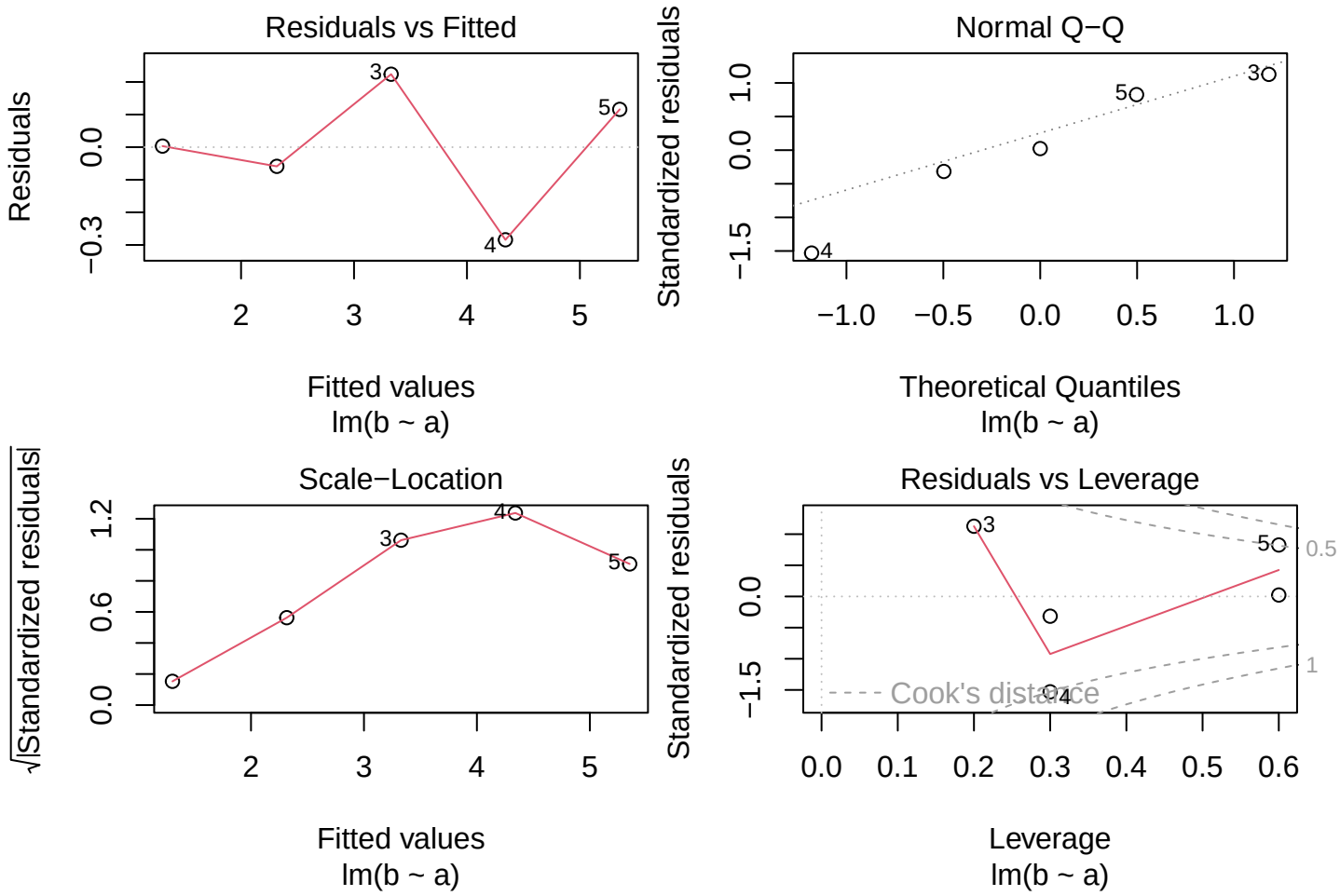
```
##
## Call:
## lm(formula = b ~ a)
##
## Coefficients:
## (Intercept)            a
##      0.2925       1.0120
```

```
summary(fit)        # shows more information including R²
```

```
##
## Call:
## lm(formula = b ~ a)
##
## Residuals:
##         1         2         3         4         5
##  0.003283 -0.058838  0.223784 -0.284183  0.115955
##
## Coefficients:
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.29246    0.23276   1.256 0.297883
## a            1.01203    0.07018  14.420 0.000723 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2219 on 3 degrees of freedom
## Multiple R-squared:  0.9858, Adjusted R-squared:  0.981
## F-statistic: 207.9 on 1 and 3 DF,  p-value: 0.0007229
```

```
plot(fit)
```



If you look in the **Environment** tab, you can see that the variable `fit`, which holds the result of the linear model fit, is a list.

To extract the slope and intercept of the fitted linear model, you can use the `coefficients` function. To extract R^2, you need to use the `summary` function shown above.

```
# extract slope and intercept
intercept <- coefficients(fit)[1]
slope <- coefficients(fit)[2]
r2 <- summary(fit)$r.squared
# print values
cat("intercept = ", intercept, "slope = ", slope, "R² =", r2)
```

```
## intercept =  0.2924556 slope =  1.012028 R² = 0.9857785
```

tidyverse

The **tidyverse** is a collection of **R** packages developed by Hadley Wickam, among others. They have changed the way **R** is used. Although they take some time to learn, the packages make **R** much more powerful.

ggplot2

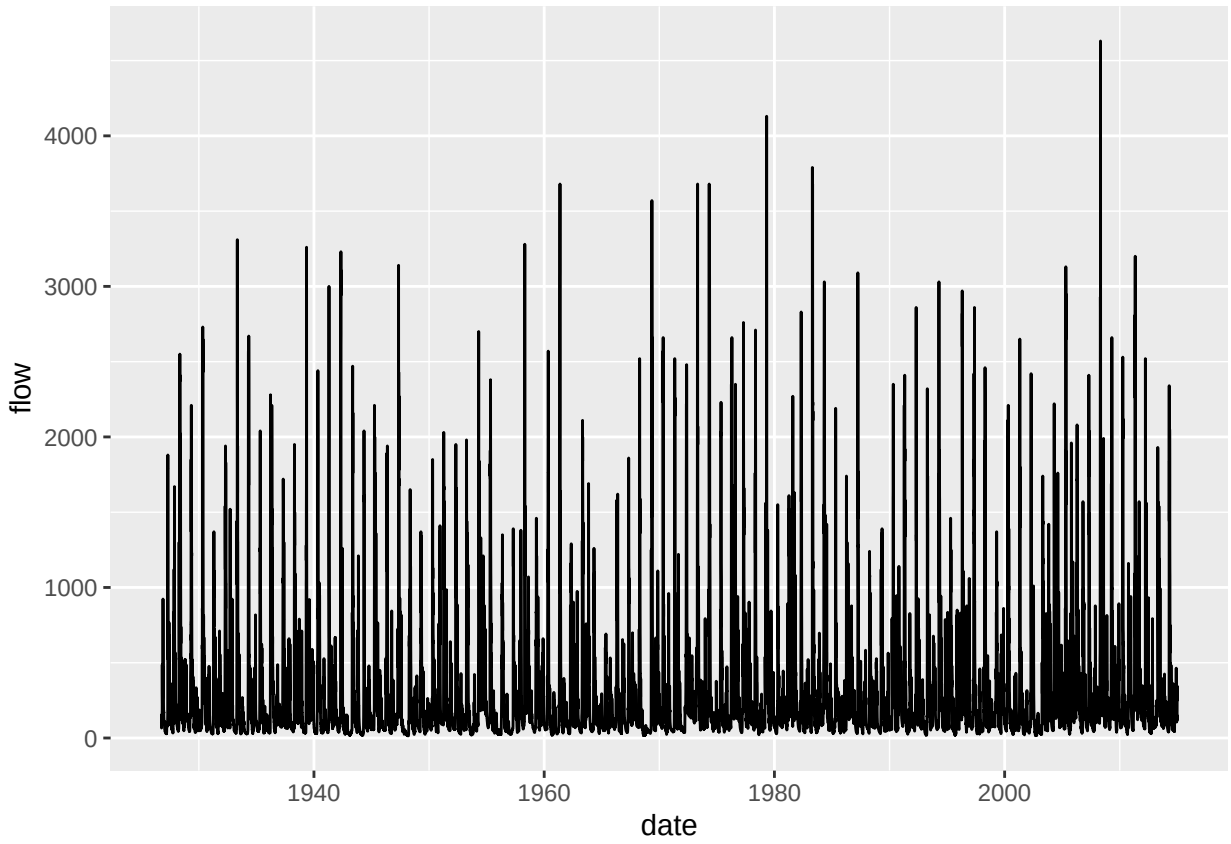
ggplot2, which is part of the **tidyverse**, gets its name because it incorporates a "Grammar of Graphing". It generates plots in a fundamentally different way than the base plot. With base plots, the plot is a static image that needs to be redrawn if changes are required. In contrast, the `ggplot` function returns an object, which contains all of the data and plot formatting used to generate the plot. The `ggplot` object can then be stored, recalled and plotted, or further modified. Although **ggplot2** works very differently from the base plot, and takes a while to learn, it is much more powerful, and allows you to create more attractive plots very easily.

Fortunately, there is a lot of help available to learn **ggplot2**, including a dedicated web site (https://ggplot2.tidyverse.org/reference/) and even a free book: https://ggplot2-book.org/.

The basic components of a **ggplot2** plot are:

- the `ggplot` call;
- supplying data, usually as a data frame or similar format;
- an aesthetic, i.e. mapping of x, y, and any groupings to the plot; and
- the format of the plot, i.e. points, lines, etc.

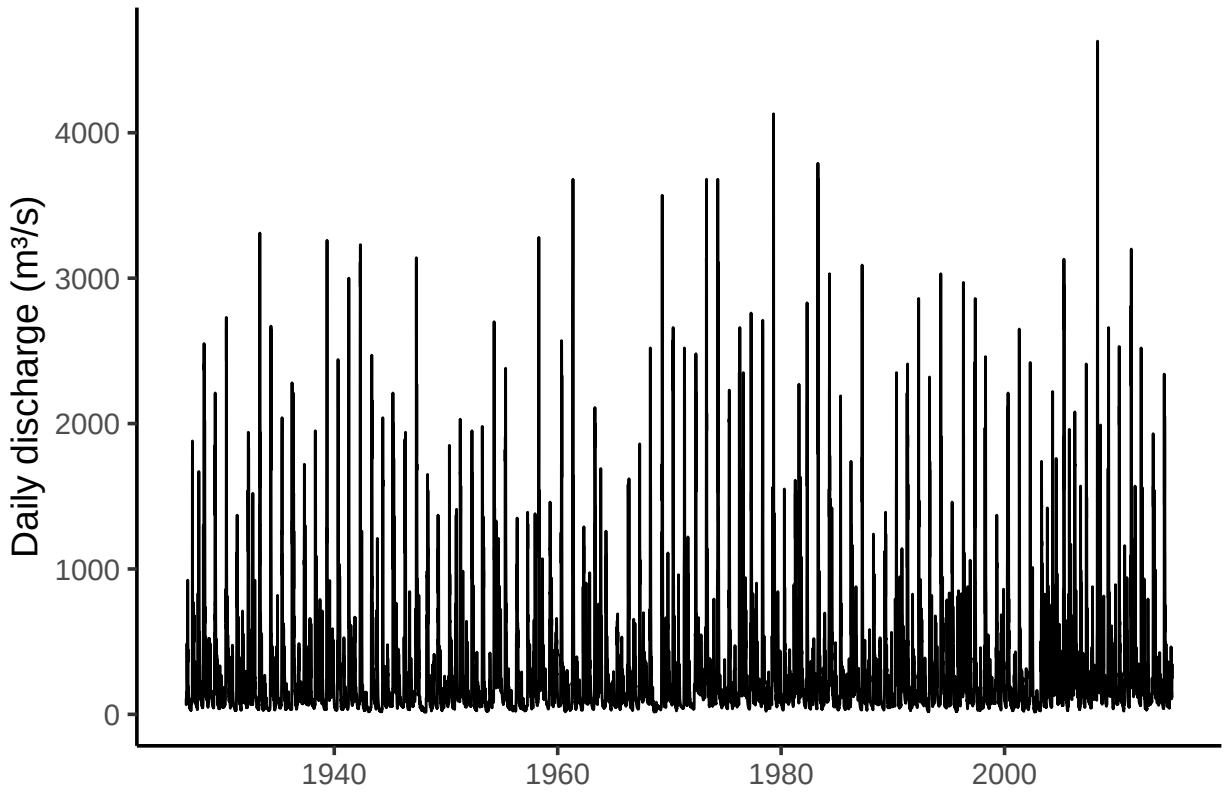
```
library(ggplot2)
p1 <- ggplot(CAN01AD002, aes(date, flow)) +
    geom_line()
p1
```



change axis labels and sizes and colour scheme

```
p2 <- p1 + theme_classic(14)+
     xlab("")+
     ylab("Daily discharge (m³/s)")

p2
```



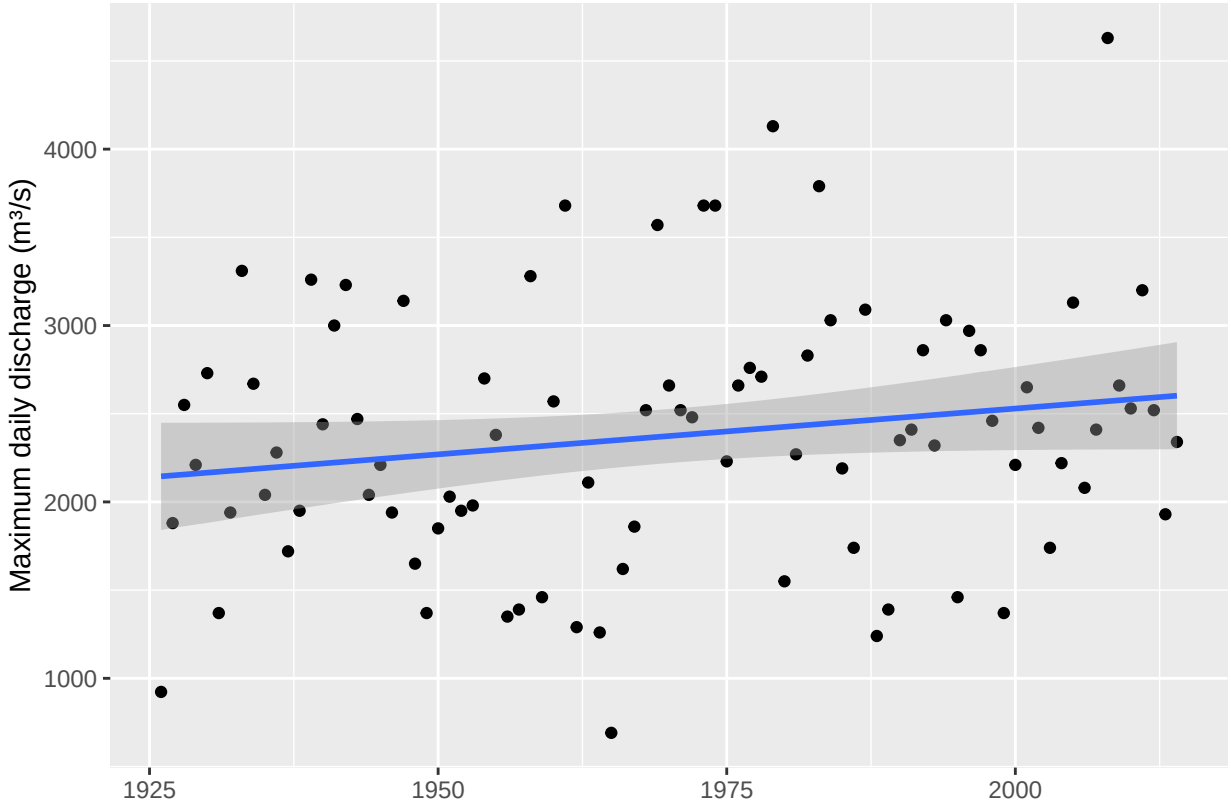
Examples of more complex plots that can be made using `ggplot2` can be found on the r-statistics.co website: http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html.

You can automatically add regression lines to plots. Saving your graph is a *lot* easier with **ggplot2** as it has a function `ggsave` which does all the work. The `width` and `height` are in inches by default, but you can use other units as well.

```
p3 <- ggplot(yearly_max, aes(year, flow)) +
  geom_point() +
  geom_smooth(method = "lm") +
  xlab("") +
  ylab("Maximum daily discharge (m³/s)")

p3
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
graph_file <- "01AD002_annual_max_Q.png"
ggsave(graph_file, width = 8, height = 5)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

Functions

If you are doing something repeatedly (like more than 3 times) it's a good idea to create a function to do it. You can create a function within your notebook, or you can store it in its own .R file, so that it can be used in other notebooks.

To create a function, you use the `function` command. You need to specify the function parameters (which can have default values). Use `return` to specify the value that the function will return.

Here's a function that converts from Fahrenheit to Celsius.

```
f2c <- function(temp_f){
  temp_c <- (temp_f - 32) / 1.8
  return(temp_c)
}
```

To use your function, just call it like any other.

```
temps_f <- seq(-40, 100, by = 20)
f2c(temps_f)
```

```
## [1] -40.000000 -28.888889 -17.777778  -6.666667   4.444444  15.555556  26.666667
## [8]  37.777778
```

You can create a file to hold this function by clicking on **File | New File | R Script**. It creates an empty file.

Now copy the function definition statement above and paste it into the new R Script.

Save the file clicking **File | Save**. Give the file a name ending in ".R". In this case call it "f2c.R".

To load in the function, you can use the command

```
source("f2c.R")
```

You will now see the function `f2c` listed in the Environment tab.

Challenge exercises

Challenge 1

Using the `ifelse` function, create a new column "hydroyear" in the CAN01AD002 data frame. If the month is 10 or greater, the hydro year is one greater than the year. Otherwise, it is the current year. Hint: you can use the `format` command to get the month from the date.

Challenge 2

Get the mean daily discharge for each hydro year. Plot the values.

Challenge 3

Classify the flows by quarter: q1 = Jan-Mar, q2 = April-June, q3 = July-September, q4 = October-December. Plot the flows facetted by each quarter using **ggplot2** with the function `facet_wrap`.