Getting Started with R & RStudio

Trent University Library & Archives' Maps, Data, and Government Information Centre

2023-09-25

Getting Started with R and RStudio

Have you always wanted to learn R programming but need help figuring out where to start? Or need a refresher on R? This introductory workshop will teach you the fundamentals of R programming to complete tasks in R and RStudio. R is a simple programming environment for statistics and graphics. RStudio is a user-friendly environment for working with R. Both are popular within industry and academics.

Maps, Data, and Government Information Centre (MaDGIC)

MaDGIC designed and presented this workshop. MaDGIC is an essential resource for all students and faculty who need access to research materials, including cartographic resources, geospatial and statistical data, and government information.

We would be delighted to assist you with your research and teaching needs, and to participate in collaborative research projects. Contact us at **madgichelp@trentu.ca** or visit us in **Bata Library room 415**. For more details on our services, please visit our website.

Installing R and RStudio

To install R, download R from the CRAN website then run the downloaded .exe file.

After you install R, install RStudio by going to the RStudio download page. Scroll to the bottom of this page, and select the appropriate installer for your PC.

Workshop script

The R script and other workshop materials are available from MaDGIC's Github.

Commenting

Within the provided R script, we provided many comments to aid your learning experience.

One or more # defined a comment. Anything following one or more # on a single line is considered a comment and not executed.

For example:

```r
30 + 4 # This is a comment
```

[1] 34

Only the addition is returned, the text preceding # is ignored by R.

```r
# 5 + 6
```

Similarly, this addition is not calculated as it is preceded by a #.

Use comments to annotate R script to improve understanding, make notes, prevent execution of code, and/or include resources. Application of four or five #s creates headers and sub-headers, respectively. You can quickly comment and uncomment entire lines with Ctrl + Shift + C on Windows and Cmd + Shift + C on Mac.

Using R as a calculator

We will begin by using R as a calculator as an introduction to R, RStudio, and the R programming language. Basic calculator operations in R work the same as a standard calculator.

- Addition:

```r
5 + 4
```

- Subtraction:

```r
10 - 9
```

- Multiplication:

```r
3 * 2
```

- Division:

```r
20/4
```

- Exponent:

```r
2^3
```

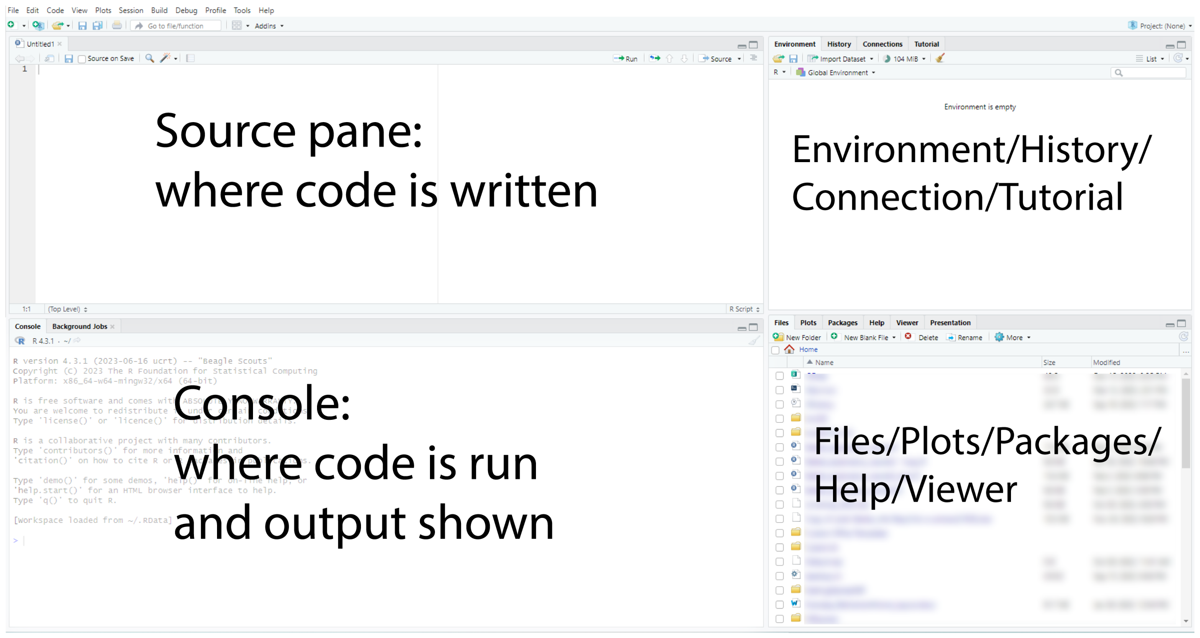
Using arithmetic operations, let's practice how to run code in R.

Running R code

R is a programming language which requires text commands known as code. These commands are entered directly into the console or initiated from the source pane (Fig. 1).

If the source pane is not visible, select in RStudio to create a new R script file.

In the console , type '123 + 789' and press enter.



```
R version 4.3.1 (2023-06-16 ucrt) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

>
```

Figure 1: The default layout of RStudio panes

```
123 + 789
```

[1] 912

Notice that the console runs the command and returns the output. Running code directly in the console is great for quick calculations. However, the console does not save R code for later use.

Instead, working from the source pane is recommended. The source pane allows one to save and edit the script for later reference.

We can run code from the source pane by highlighting one or more lines and:

- pressing Run
- Ctrl + Enter on Windows, Cmd + Enter on Mac or
- Alt + Enter

Let's run code from the source pane. Put your cursor on or highlight line 15 of the provided script and press Run.

```
3 * 5
```

[1] 15

Repeat by using the keyboard shortcuts Ctrl + Enter or Alt + Enter. Notice that all will return the same result. How you run code is your preference; do what works best for you.

More complex calcualtions: BEDMAS

When calculating more complex calculations R will apply the order of operations, commonly known as BEDMAS. Applying BEDMAS changes how R calculates the provided command.

For example, running a calculation without brackets provides:

```r
2 + 100 / 2 - 25
```

[1] 27

However, if we run the same calculation with brackets:

```r
(2 + 100) / (2 - 25)
```

[1] -4.434783

The use of brackets changed the calculation owing to BEDMAS. If in doubt, **use brackets to ensure R interprets the calculation as intended**.

Challenge 1

Run the following equation in the source pane or console: $5^2 + (8 * 5) + 3$

```r
(5^2) + (8*5) + 3
```

[1] 68

Storing data as objects

A fundamental concept in R programming is objects. Objects allow us to store a value (i.e. 5), function, plot, output, model, and more under a name. Object names can easily be applied to access the corresponding value or other associated data.

Objects are created by:

1. Defining a name
2. Specify the **assignment operator**: **<-**
3. Assigning a value

To explore objects, we will use daily weather data for Peterborough, ON from Daymet. Daymet provides long-term, continuous, 1 x 1 km gridded estimates of daily weather and climatology variables by interpolating and extrapolating ground-based observations through statistical modeling techniques.

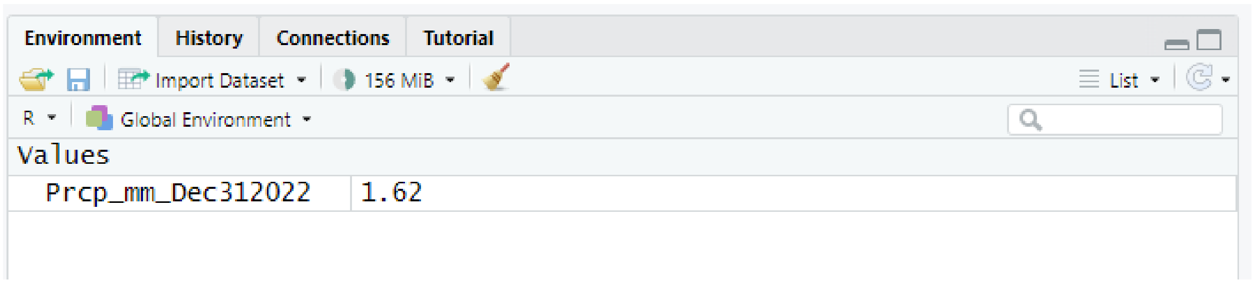
Create an object that indicates how much precipitation fell on December 31, 2022, in the Peterborough, ON area:

```r
Prcp_mm_Dec312022 <- 13.2
```

Enter object names to the left of the assignment operator (<-). These names can contain letters, numbers, underscores, and periods, but must begin with a letter. Try to use clear and short object names.

Once an object is created, its name and corresponding value appears in the environment pane (Fig. 2).

Now that the object is defined, running the object's name returns the assigned value:



```
Values
  Prcp_mm_Dec312022    1.62
```

Figure 2: Environment pane with the object PrcP_mm_Dec312022

```
Prcp_mm_Dec312022
```

[1] 13.2

Object names are **case and spelling-sensitive**! If the name is wrong, R will return an error or the wrong object. For example, try to call the object *Prcp_mm_Dec312022*:

When the case is wrong…

```
prcp_mm_dec312022
```

```
## Error in eval(expr, envir, enclos): object 'prcp_mm_dec312022' not found
```

… or the object name is misspelled..

```
Prc_mm_D312022
```

```
## Error in eval(expr, envir, enclos): object 'Prc_mm_D312022' not found
```

.. an error is returned indicating that object does not exist. Be careful when entering object names. RStudio will prompt you with the correct spelling of objects loaded in the environment.

Major data types

R interprets objects based on their data type or structure (see Data Structures). The major data types are:

- Numeric (num): numbers with or without decimal values (numeric values are also known as double (dlb)).

```
MeanTemp_C_Mar012023 <- -3.9
MeanTemp_C_Mar012023
```

[1] -3.9

- Integer (int): number values without decimal values. Integers are specified by placing an **L** at the end of the number. Integers use less memory than numeric values.

```r
n_Ptbo_weatherstations <- 13L
n_Ptbo_weatherstations
```

[1] 13

- Character (chr): text values are defined **in quotations**, " " or ' '. Any text within a single pair of quotations is a single character value.

```r
Data_Source <- "Daymet"
Data_Source
```

[1] "Daymet"

- Logical (logi): true, false, or missing values. Logical values are defined in **all capital letters without quotations**. True or false may be defined with TRUE or T, FALSE or F.

```r
Precip_Mar112022 <- TRUE
Precip_Mar112022
```

[1] TRUE

```r
Precip_Mar112022 <- T
Precip_Mar112022
```

[1] TRUE

Remember, objects represent the values assigned. Therefore objects can be used in the place of raw values in calculations. For example, we can define how much snow fell on each day during the 2020 holiday season, then add them together:

```r
Snow_mm_Dec242020 <- 17.26
Snow_mm_Dec252020 <- 6.84
Snow_mm_Dec262020 <- 1.96

Snow_mm_Dec242020 + Snow_mm_Dec252020 + Snow_mm_Dec262020
```

[1] 26.06

Similarly, you can assign a calculation to a new object for later use:

```r
TotalHolidaySnowfall_mm_2020 <- Snow_mm_Dec242020 + Snow_mm_Dec252020 + Snow_mm_Dec262020
TotalHolidaySnowfall_mm_2020
```

[1] 26.06

Challenge 2

Create the following three objects:

- an object named "x" with a value of 5

- an object named "b" with a value of 8
- an object named "c" with a value of 3

then use the objects to calculate (x^2) + (b * x) + c

```
x <- 5
b <- 8
c <- 3

(x^2) + (b * x) + c
```

[1] 68

Functions

Functions allow users to execute simple to complex commands quickly. Thousands of functions are available in R for its users.

For example, the *abs()* function returns the absolute value of a number:

```
abs(-3)
```

[1] 3

abs() will only work for numeric values.

Meanwhile, *toupper()* capitalizes all letters in the character value(s) provided:

```
toupper("Hello world")
```

[1] "HELLO WORLD"

Once an object is defined, it may be managed, analyzed, or applied in another functionality (i.e. in an app). These processes are all accomplished through functions.

Recall two of our objects:

```
MeanTemp_C_Mar012023
```

[1] -3.9

```
Data_Source
```

[1] "Daymet"

As MeanTemp_C_Mar012023 is a numeric value, apply it to *abs()*:

```
abs(MeanTemp_C_Mar012023)
```

[1] 3.9

Similarly, we can alter Data_Source with *toupper*:

```r
toupper(Data_Source)
```

[1] "DAYMET"

Some functions return information about the object provided. For example, *class()* determines the data type of the object provided.

```r
class(MeanTemp_C_Mar012023)
```

[1] "numeric"

```r
class(Data_Source)
```

[1] "character"

We will continue to explore more functions through this workshop. If you are interested in seeing all functions available in R (excluding additional packages), please refer to the R base package.

Getting help with functions

R has thousands of functions available out of the box, each with a specific purpose, inputs, and arguments. An input is some value(s) that must be provided by the user. Arguments are settings within the functions that may or may not require user input.

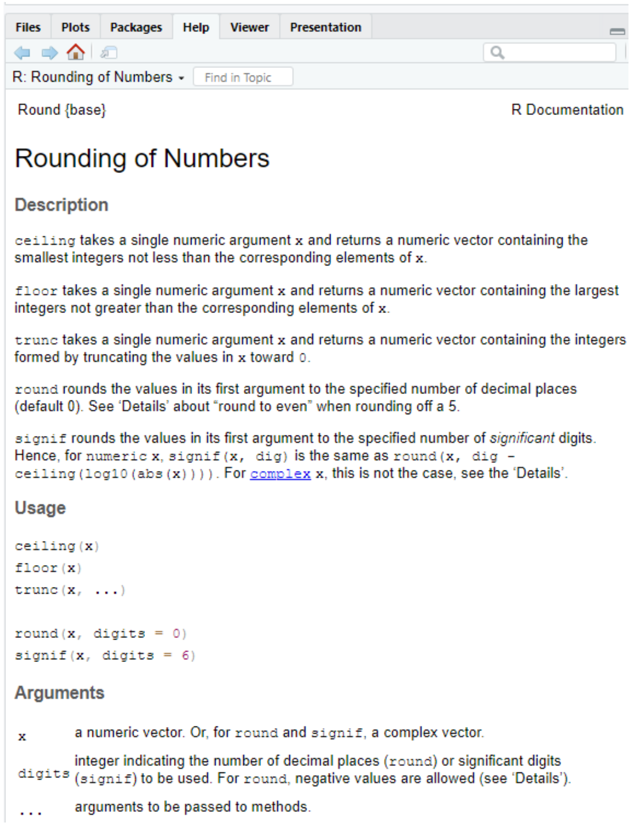
Each function has its own required/optional inputs and arguments. Learning to use each function properly is crucial to success in R. To ensure users understand functions, all R functions have a help page accessed by running ?function().

When starting out, it can be overwhelming determining which function you function should apply. If you find yourself in this situation, do not hesitate to contact MaDGIC at madgichelp@trentu.ca. Otherwise, Google is your friend.

Basic functions

For example, if you needed to round a number to two decimal places, you could use *round()*. But how does this function work? To answer that, open the help page:

```r
?round()
```



Round {base} R Documentation

Rounding of Numbers

Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward `0`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). See 'Details' about "round to even" when rounding off a 5.

`signif` rounds the values in its first argument to the specified number of *significant* digits. Hence, for `numeric x`, `signif(x, dig)` is the same as `round(x, dig - ceiling(log10(abs(x))))`. For `complex` `x`, this is not the case, see the 'Details'.

Usage

```
ceiling(x)
floor(x)
trunc(x, ...)

round(x, digits = 0)
signif(x, digits = 6)
```

Arguments

`x`	a numeric vector. Or, for `round` and `signif`, a complex vector.
`digits`	integer indicating the number of decimal places (`round`) or significant digits (`signif`) to be used. For `round`, negative values are allowed (see 'Details').
`...`	arguments to be passed to methods.

The help page will open on the Help tab in RStudio's bottom right pane. If the pane does not open automatically, please navigate to the help tab.

The help page provides essential of information about *round()*. First, **Description** outlines the specified and similar functions that you may wish to apply and what each does. Here, we see that R has five dedicated functions for rounding. These include rounding up, down, to integer, some decimal places, or significant digits. As we want to round to two decimal places, we will use *round()*.

The second section, **Usage**, indicates the argument positions and any default values, if applicable. Specifically, *round()* is presented as: **round(x, digits = 0)**, indicating *round()* has two arguments x and digits. Digits has a default value of zero, while x has no default value.

The third second, **Arguments**, describes each of the arguments specified in Usage. In the case of *round()*, *x* is a numeric value or vector (see Data Structures) and *digits* is an integer indicating the number of decimal places to round to.

When learning a new function, take a look at the **Examples** section at the end of the help page. Working through an example of how the function was intended to be used can be very insightful.

Using this information, let's try rounding MeanAnnualTemp_degC_2022 to two decimal places using *round()*.

```r
MeanAnnualTemp_degC_2022 <- 7.25156
round(MeanAnnualTemp_degC_2022)
```

```
[1] 7
```

Running *round()* on our object, the value is rounded to zero decimal places as zero is the default digits argument value. To achieve two decimal places, we must specify *digits = 2* in *round()*:

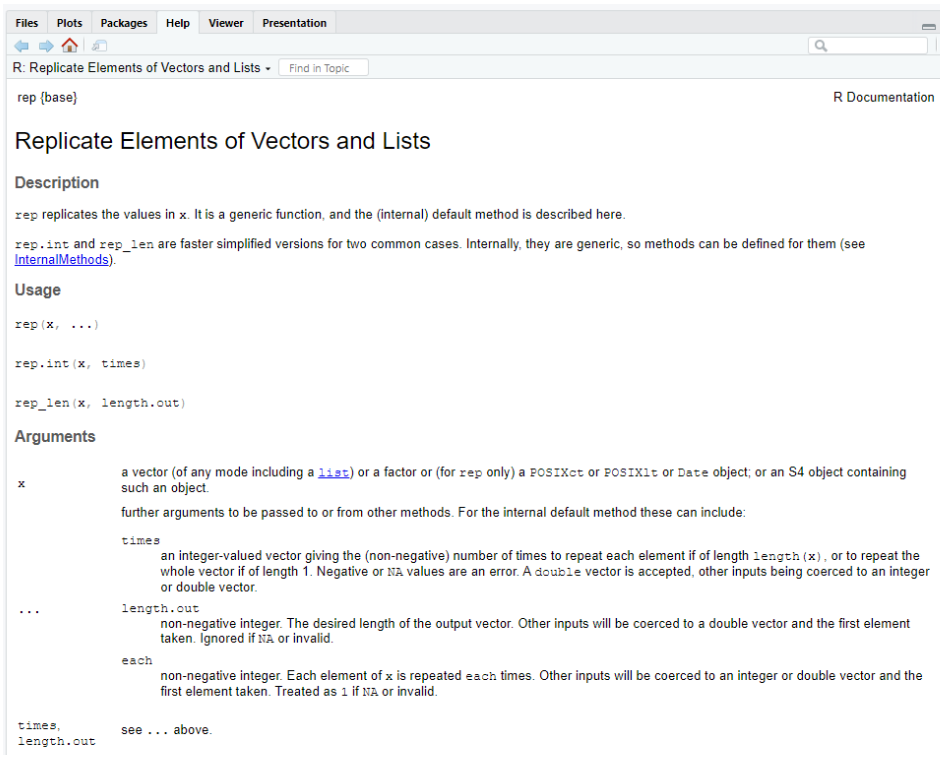
```r
round(MeanAnnualTemp_degC_2022, digits = 2)
```

```
[1] 7.25
```

Increasing complexity: multiple and optional arguments

round() is a simple function with only a single input and argument. Let's try a function with multiple arguments, *rep()*, the replicate elements function.

```r
?rep()
```



rep {base} R Documentation

Replicate Elements of Vectors and Lists

Description

`rep` replicates the values in `x`. It is a generic function, and the (internal) default method is described here.

`rep.int` and `rep_len` are faster simplified versions for two common cases. Internally, they are generic, so methods can be defined for them (see InternalMethods).

Usage

```
rep(x, ...)

rep.int(x, times)

rep_len(x, length.out)
```

Arguments

`x`	a vector (of any mode including a `list`) or a factor or (for `rep` only) a `POSIXct` or `POSIXlt` or `Date` object; or an S4 object containing such an object.
`...`	further arguments to be passed to or from other methods. For the internal default method these can include: `times` an integer-valued vector giving the (non-negative) number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1. Negative or `NA` values are an error. A `double` vector is accepted, other inputs being coerced to an integer or double vector. `length.out` non-negative integer. The desired length of the output vector. Other inputs will be coerced to a double vector and the first element taken. Ignored if `NA` or invalid. `each` non-negative integer. Each element of `x` is repeated `each` times. Other inputs will be coerced to an integer or double vector and the first element taken. Treated as `1` if `NA` or invalid.
`times`, `length.out`	see `...` above.

Review *rep()*'s Usage section. Notice, we have an argument of x and "…".

Reading the arguments section, we see that x is a input object and "…" represents further optional arguments to be passed without default values. These include:

- *times*: the number of times the input value will be repeated
- *length*: the number of total elements in the output
- *each*: the number of times each input element is output

Let's try using this function:

```r
replicate_this <- 1:3
replicate_this
```

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

Using a colon, :, generates a sequence between the two numbers specified on either side of the colon. The sequence will increase or decrease by one accordingly.

```r
rep(replicate_this)
```

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

Notice, without specifying any arguments *rep()* returns the same as the input. *rep()* has no default argument values and thus does not know how to replicate the provided values.

Let's add some arguments to *rep()*:

```r
rep(replicate_this, times = 3)
```

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

Adding *times* repeats the entire object the specified number of times.

```r
rep(replicate_this, each = 3)
```

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

Adding *each*, tells *rep()* to replicate each number in the object three times, instead of the entire object.

```r
rep(replicate_this, each = 3, times = 3)
```

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

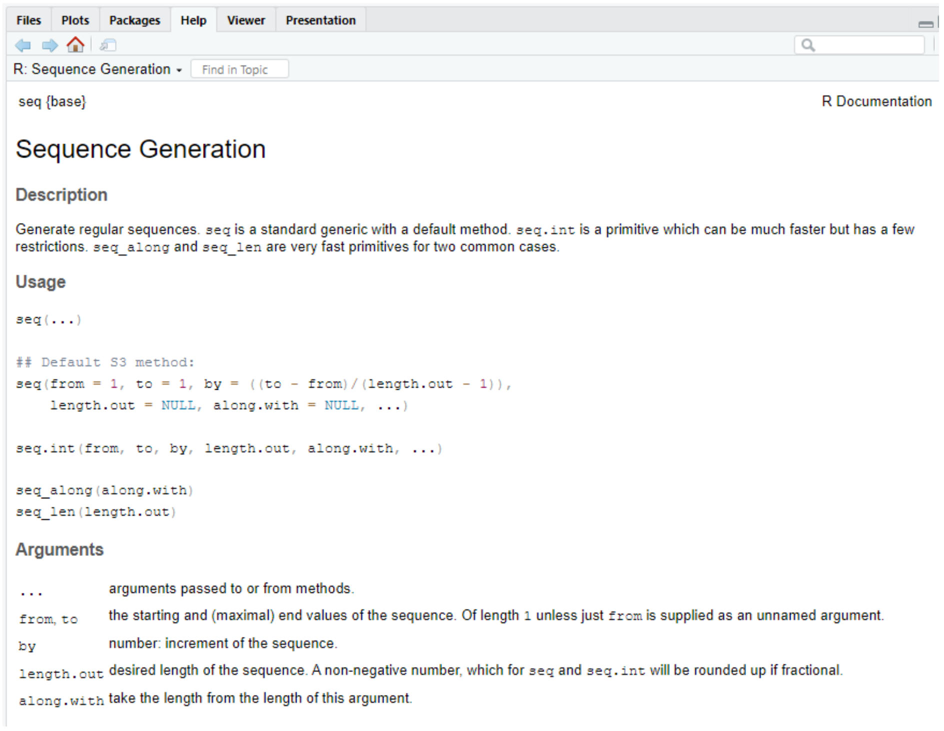
Using *each* and *times* repeats each number in the object three times, then all repeated values three times.

Including, excluding, and/or changing function arguments can significantly impact the output. Be sure to take some time to understand a function and its arguments to ensure it is applied correctly.

Challenge 3

Let's look at one more function: *seq()*, sequence generation.

```r
?seq()
```



seq {base} R Documentation

Sequence Generation

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Arguments

`...`	arguments passed to or from methods.
`from, to`	the starting and (maximal) end values of the sequence. Of length `1` unless just `from` is supplied as an unnamed argument.
`by`	number: increment of the sequence.
`length.out`	desired length of the sequence. A non-negative number, which for `seq` and `seq.int` will be rounded up if fractional.
`along.with`	take the length from the length of this argument.

Notice *seq()*'s usage only indicates optional argument with "…". Just below, the "Default S3 Method", which provides the default value for all optional arguments.

Therefore, *seq()* has all optional arguments, and can be run without an input:

```
seq()
```

[1] 1

Review the help page, which of the following seq() commands would provide the output:

[1] 3 7 11 15 19 23 27

```
# A.
seq(3, 4, 7)
# B.
seq(from = 3, by = 4, length.out = 7)
# C.
seq(from = 3, to = 30, by = 4)
# D.
seq(from = 3, to = 32, by = 4)
```

Either B

```
# B.
seq(from = 3, by = 4, length.out = 7)
```

[1] 3 7 11 15 19 23 27

or C will return the desired sequence.

```
# C.
seq(from = 3, to = 30, by = 4)
```

```
[1] 3 7 11 15 19 23 27
```

Multiple ways exist to accomplish any task in R, even with a single function. When learning R, focus on understanding how you get to the answer rather than how someone else accomplished a similar task. As you work more in R and other programming languages, you will begin to see different ways to perform tasks.

Packages

R comes with a base set of functions. However, there are millions more available through packages. Packages allow you to install more functions to expand its capabilities. All R packages are free owing to R's open-sourced nature.

Think of an R package as analogous to a smartphone App— a package effectively extends what R can do, just as an App extends what a smartphone can do (Fig. 3).

Figure 3: R packages are to R as Smartphone apps are to smartphones. Both extend the functionality of the base software.

Many packages specialize in a specific task, field, or data type. What package you need depends on what you are trying to do. Some popular packages include *dplyr* for data manipulation and *xlsx* for importing and exporting Microsoft Excel work books.

Packages must be **installed once per computer** and **loaded once per session** (each time R is opened). Packages may be installed and loaded by code or the package pane. Installing and loading packages by code or the package pane are equivalent processes. The only difference is personal preference.

Let's look at how to install and load a package by code vs. through the package pane.

Install and load pacakges by code

Install packages with *install.packages()*, with the desired package(s) specified provided in quotations:

```
install.packages("dplyr")
```

Once a package is installed, load it with *library()*. The package name can be provided in *library()* with or without quotations.

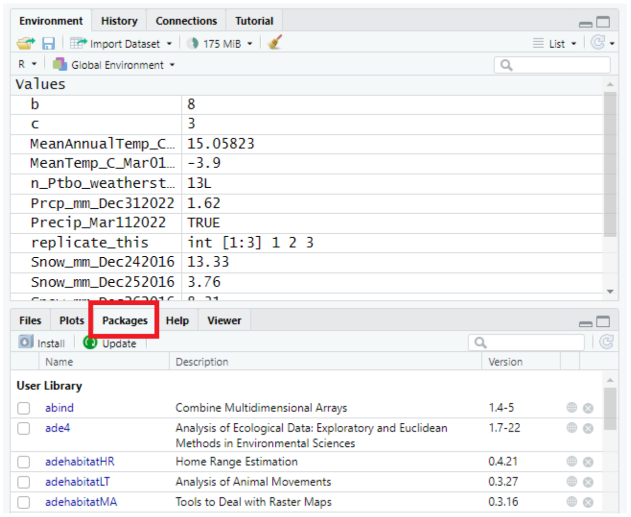
```r
library(dplyr)
```

Install and load pacakges by the package pane

Packages may also be installed and loaded using the RStudio interface, precisely the package pane (Fig. 1). This process is unique to RStudio, and does not apply to R.

Install package by the package pane

1. Select the **Packages** tab in the bottom right pane of RStudio.



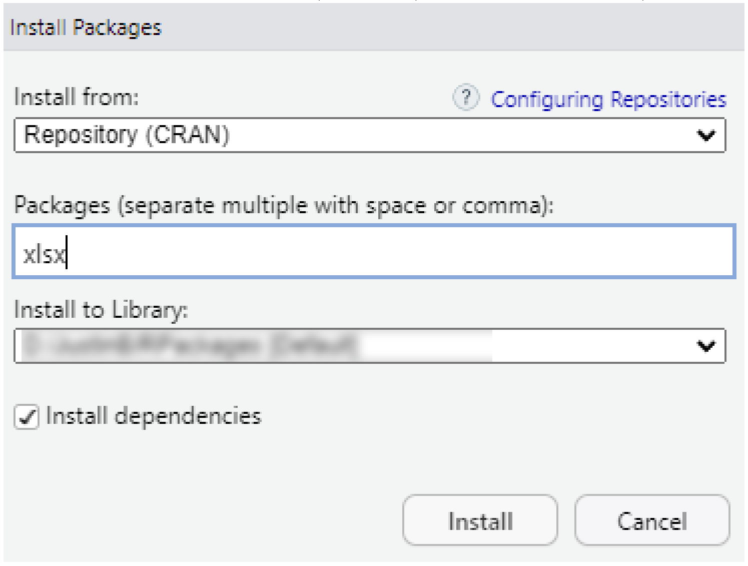
Values

b	8
c	3
MeanAnnualTemp_C…	15.05823
MeanTemp_C_Mar01…	-3.9
n_Ptbo_weatherst…	13L
Prcp_mm_Dec312022	1.62
Precip_Mar112022	TRUE
replicate_this	int [1:3] 1 2 3
Snow_mm_Dec242016	13.33
Snow_mm_Dec252016	3.76

User Library

Name	Description	Version
abind	Combine Multidimensional Arrays	1.4-5
ade4	Analysis of Ecological Data: Exploratory and Euclidean Methods in Environmental Sciences	1.7-22
adehabitatHR	Home Range Estimation	0.4.21
adehabitatLT	Analysis of Animal Movements	0.3.27
adehabitatMA	Tools to Deal with Raster Maps	0.3.16

2. Click Install
3. Enter the package's name (e.g. *xlsx*) in the "Packages (separate multiple with space or comma)" filed.



Install from: Repository (CRAN)

Packages (separate multiple with space or comma): xlsx

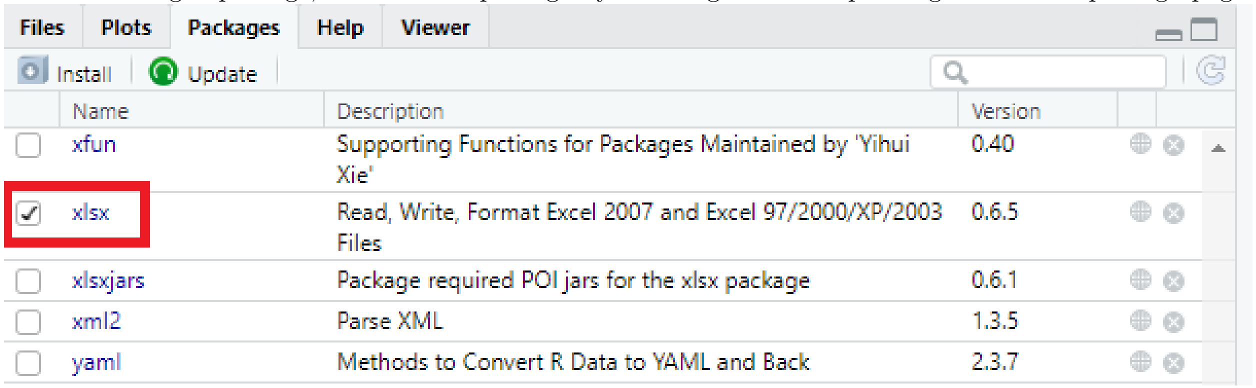
Install to Library: [redacted]

[✓] Install dependencies

4. Click Install. If no errors are returned, the package is now installed.

Load package in the package pane

After installing a package, we load the package by checking the corresponding box in the package page:



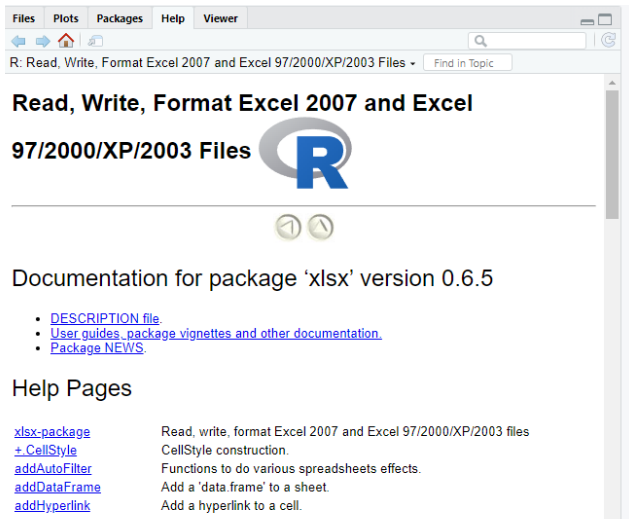
	Name	Description	Version
☐	xfun	Supporting Functions for Packages Maintained by 'Yihui Xie'	0.40
✓	xlsx	Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files	0.6.5
☐	xlsxjars	Package required POI jars for the xlsx package	0.6.1
☐	xml2	Parse XML	1.3.5
☐	yaml	Methods to Convert R Data to YAML and Back	2.3.7

As you use the package pane, RStudio will auto generate the corresponding script required to install and load packages.

Getting help with packages

Packages also have help pages which are accessed with *help()*. For example, open the help page for the *xlsx* package.

```r
help(package = "xlsx")
```



R: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files

Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files

Documentation for package 'xlsx' version 0.6.5

- DESCRIPTION file.
- User guides, package vignettes and other documentation.
- Package NEWS.

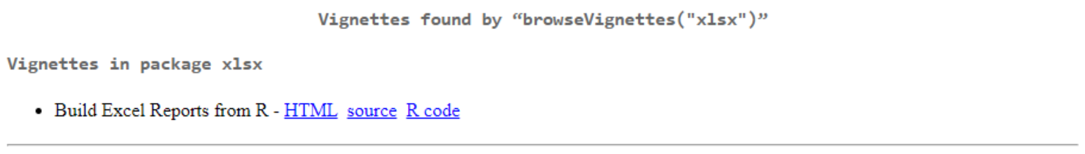
Help Pages

xlsx-package	Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files
+ CellStyle	CellStyle construction.
addAutoFilter	Functions to do various spreadsheets effects.
addDataFrame	Add a 'data.frame' to a sheet.
addHyperlink	Add a hyperlink to a cell.

All R packages have these help pages, which outline all available package specific functions with a short description and their individual help pages linked. This allows you to get a sense of what the package provides.

Alternatively, some packages offer tutorials known as vignettes. These can be accessed with *browseVingettes()*. If a vignette is available, it will open in a web browser.

```r
browseVignettes("xlsx")
```



Vignettes found by "browseVignettes("xlsx")"

Vignettes in package xlsx

- Build Excel Reports from R - HTML source R code

Vignettes are a great resource to demonstrate the intended package workflow or how to use multiple functions together.

Lastly, the most popular R packages have cheat sheets available. Cheat sheets provide a visual summary of the package and its functions (similar to the cheat sheet provided for this workshop). Please refer Posit Cheatsheets.

Challenge 4

Install and load the *beepr* package by code or package pane.

```r
install.packages("beepr")
library(beepr)
```

Ensure the package is installed and loaded by running one of the package's functions:

```r
beep()
```

If successful, you will hear a beep; nothing is returned in the console. Remember to examine *beep()*'s help page to understand its arguments.

Data Structures

So far, we have only worked with objects with a single value assigned to them. Formally, these are **scalar** values. Applying scalar values will only get us so far. Instead, we use data structures that create data sets.

There are five major data structures in R (Fig. 4; exemplified by R built-in data sets):

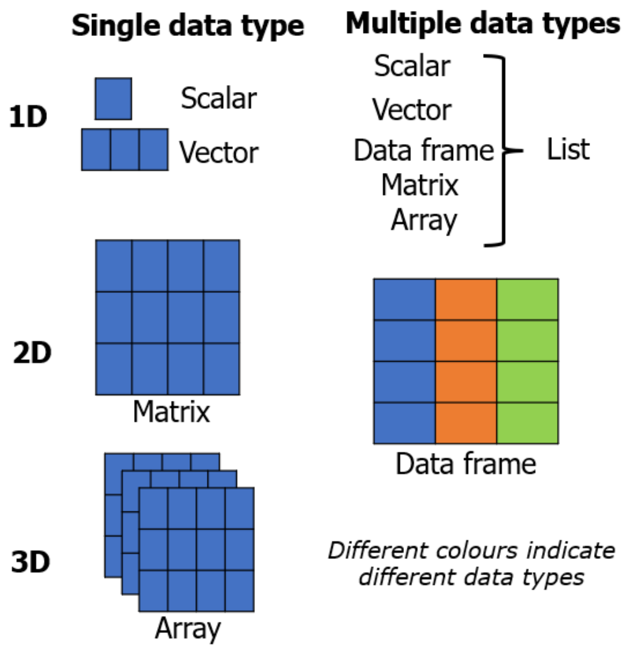


Figure 4: Overview of data structures in R. Different coloured cells represented different data types (e.g. numeric, character, logical)

- **Vector**: one-dimensional series of the **same** data type.

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
euro # euro represents conversion rates between various Euro currencies
```

```
##         ATS         BEF         DEM         ESP         FIM         FRF
##   13.760300   40.339900    1.955830  166.386000    5.945730    6.559570
##         IEP         ITL         LUF         NLG         PTE
##    0.787564 1936.270000   40.339900    2.203710  200.482000
```

- **Matrix**: A two-dimensional (rows and columns)) structure with elements of the *same* data type

```
co2
# co2 represents atmospheric concentrations of CO2 are expressed in parts per million
# (ppm) and reported in the preliminary 1997 SIO manometric mole fraction scale.
```

```
##         Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct
## 1959 315.42 316.39 316.27 318.01 316.73 318.42 317.78 319.45 318.58 319.58
## 1960 316.31 314.65 316.81 315.74 317.54 316.63 318.40 317.25 318.92 317.61
## 1961 316.50 313.68 317.42 314.00 318.38 314.83 319.53 316.11 319.70 316.05
## 1962 317.56 313.18 318.87 313.68 319.31 315.16 320.42 315.27 321.22 315.83
## 1963 318.13 314.66 319.87 314.84 320.42 315.94 320.85 316.53 322.08 316.91
## 1964 318.00 315.43 319.43 316.03 319.61 316.85 320.45 317.53 321.31 318.20
##         Nov    Dec
## 1959 319.41 320.27
## 1960 320.07 318.54
## 1961 320.74 316.54
## 1962 321.40 316.71
## 1963 322.06 317.53
## 1964 321.73 318.55
```

- **Array**: A multi-dimensional (rows, columns, depth) structure with elements of the **same** data type, composed of matrices

```
Titanic
# Titanic details the fate of passengers on the fatal maiden voyage of the ocean
# liner 'Titanic', summarized according to economic status (class), sex, age and survival
```

```
## , , Age = Child, Survived = No
## 
##       Sex
## Class  Male Female
##   1st     0      0
##   2nd     0      0
##   3rd    35     17
##   Crew    0      0
## 
## , , Age = Adult, Survived = No
## 
##       Sex
## Class  Male Female
##   1st   118      4
##   2nd   154     13
##   3rd   387     89
##   Crew  670      3
## 
## , , Age = Child, Survived = Yes
## 
##       Sex
## Class  Male Female
##   1st     5      1
##   2nd    11     13
##   3rd    13     14
##   Crew    0      0
## 
## , , Age = Adult, Survived = Yes
## 
##       Sex
## Class  Male Female
##   1st    57    140
##   2nd    14     80
##   3rd    75     76
##   Crew  192     20
```

```
iris3
# iris3 contains the measurements in centimeters of the variables sepal length
# and width and petal length and width, respectively, for 50 flowers from each
# of 3 species of iris
```

```
## , , Setosa
##
##      Sepal L. Sepal W. Petal L. Petal W.
## [1,]      5.1      3.5      1.4      0.2
## [2,]      4.9      3.0      1.4      0.2
## [3,]      4.7      3.2      1.3      0.2
## [4,]      4.6      3.1      1.5      0.2
## [5,]      5.0      3.6      1.4      0.2
## [6,]      5.4      3.9      1.7      0.4
##
## , , Versicolor
##
##      Sepal L. Sepal W. Petal L. Petal W.
## [1,]      7.0      3.2      4.7      1.4
## [2,]      6.4      3.2      4.5      1.5
## [3,]      6.9      3.1      4.9      1.5
## [4,]      5.5      2.3      4.0      1.3
## [5,]      6.5      2.8      4.6      1.5
## [6,]      5.7      2.8      4.5      1.3
##
## , , Virginica
##
##      Sepal L. Sepal W. Petal L. Petal W.
## [1,]      6.3      3.3      6.0      2.5
## [2,]      5.8      2.7      5.1      1.9
## [3,]      7.1      3.0      5.9      2.1
## [4,]      6.3      2.9      5.6      1.8
## [5,]      6.5      3.0      5.8      2.2
## [6,]      7.6      3.0      6.6      2.1
```

- **Data frame**: A two dimensional (rows and columns) structure with elements of the **same and/or different** data types

```
iris
# iris contains the measurements in centimeters of the variables sepal length
# and width and petal length and width, respectively, for 50 flowers from each
# of 3 species of 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
```

```
warpbreaks # The Number of Breaks in Yarn during Weaving
```

```
##   breaks wool tension
## 1     26    A       L
## 2     30    A       L
## 3     54    A       L
## 4     25    A       L
## 5     70    A       L
## 6     52    A       L
```

- **Lists**: a data structure having components of the **same or different data structures**

```
ability.cov
# Six tests were given to 112 individuals.
# The covariance matrix is given in ability.cov.
```

```
## $cov
##         general picture  blocks   maze reading   vocab
## general  24.641   5.991  33.520  6.023  20.755  29.701
## picture   5.991   6.700  18.137  1.782   4.936   7.204
## blocks   33.520  18.137 149.831 19.424  31.430  50.753
## maze      6.023   1.782  19.424 12.711   4.757   9.075
## reading  20.755   4.936  31.430  4.757  52.604  66.762
## vocab    29.701   7.204  50.753  9.075  66.762 135.292
##
## $center
## [1] 0 0 0 0 0 0
##
## $n.obs
## [1] 112
```

Notice the list consists of a matrix (cov), vector (center), and scalar(n.obs).

Each data structure has its advantages and disadvantages, depending on the context. Users most commonly apply vectors and data frames as they are what most functions require. Meanwhile, background statistical calculations involve matrices and arrays. Lists are handy to keep similar data together or when functions return more than one object.

This workshop will focus on vectors and data frames. To learn more about data structures in R, please refer to Data structures by Hadley Wickman and Learning about data structures in R from R-bloggers.

Vectors

Vectors are the building blocks of all non-scalar data structures. Use combine, *c()*, to create vectors. Each vector element must be the same data type and separated by a comma.

Character vector:

```
Months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec")
Months
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

Numeric vector:

```
MaxMonthtemp_C_Ptbo_2022 <- c(3.62, 5.07, 17.67, 23.1, 29.26, 29.55,
                              29.14, 30.64, 25.2, 22.7, NA, 14.58)
MaxMonthtemp_C_Ptbo_2022
```

```
##  [1]  3.62  5.07 17.67 23.10 29.26 29.55 29.14 30.64 25.20 22.70    NA 14.58
```

Logical vector:

```
Snow_Present_Ptbo_2021 <- c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE,
                            FALSE, FALSE, FALSE, TRUE, TRUE)
Snow_Present_Ptbo_2021
```

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

Unlike scalar objects, vectors and other data structures can get large and complicated to see in the console. If you need to inspect the data, you can use a function that returns only part of the data structure.

For example, *head()* returns only the first six values:

```
head(MaxMonthtemp_C_Ptbo_2022, n = 6)
```

```
## [1]  3.62  5.07 17.67 23.10 29.26 29.55
```

tail() returns the last size values:

```
tail(MaxMonthtemp_C_Ptbo_2022, n = 6)
```

```
## [1] 29.14 30.64 25.20 22.70    NA 14.58
```

length() indicates how many elements are in the vector:

```
length(MaxMonthtemp_C_Ptbo_2022)
```

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

We can alter individual vector values with arithmetic (i.e. add, subtract, divide) or vector element functions. For example:

```
MaxMonthtemp_C_Ptbo_2022 + 20
```

```
##  [1] 23.62 25.07 37.67 43.10 49.26 49.55 49.14 50.64 45.20 42.70    NA 34.58
```

log() computes logarithms of each element, by default natural logarithms.

```
log(MaxMonthtemp_C_Ptbo_2022)
```

```
##  [1] 1.286474 1.623341 2.871868 3.139833 3.376221 3.386084 3.372112 3.422306
##  [9] 3.226844 3.122365       NA 2.679651
```

tolower() translates each character element to lower case

```
tolower(Months)
```

```
##  [1] "jan" "feb" "mar" "apr" "may" "jun" "jul" "aug" "sep" "oct" "nov" "dec"
```

Name vector elements

Vectors are simple, with just a list of elements divided by commas. These provide little context for the data by itself. To add context, we can assign a name to each vector element.

Above, we created a vector that represented the monthly maximum temperatures in Peterborough, ON, during 2022. Calling this vector alone . . .

```
MaxMonthtemp_C_Ptbo_2022
```

```
##  [1]  3.62  5.07 17.67 23.10 29.26 29.55 29.14 30.64 25.20 22.70    NA 14.58
```

. . . only returns the numbers in the order they were defined. We have to remember what each value corresponds to or assume it follows chronological order. Instead, it is better to assign a name to each element with *names()*.

```
names(MaxMonthtemp_C_Ptbo_2022) <- Months
```

R assigns names to vector elements in order. Therefore, the first element of the vector is given the first name, the second element receives the second name, and so forth. Before assigning names, ensure the order is correct.

After running *names()*, re-run MaxMonthtemp_C_Ptbo_2022to view.

```
MaxMonthtemp_C_Ptbo_2022
```

```
##   Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec 
##  3.62  5.07 17.67 23.10 29.26 29.55 29.14 30.64 25.20 22.70    NA 14.58
```

Each vector element is now assigned the corresponding month.

Descriptive statisics

Calculating descriptive statistics (i.e. mean, median, standard deviation, variance, minimum, maximum) is essential for data analysis. All base descriptive statistics functions require a numeric vector input.

Suppose you wanted to know the maximum monthly temperatures' standard deviation, *sd()*.

```
sd(MaxMonthtemp_C_Ptbo_2022)
```

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

Hmm... not applicable (NA) is returned, meaning R could not calculate the standard deviation. Let's examine the original vector to determine why:

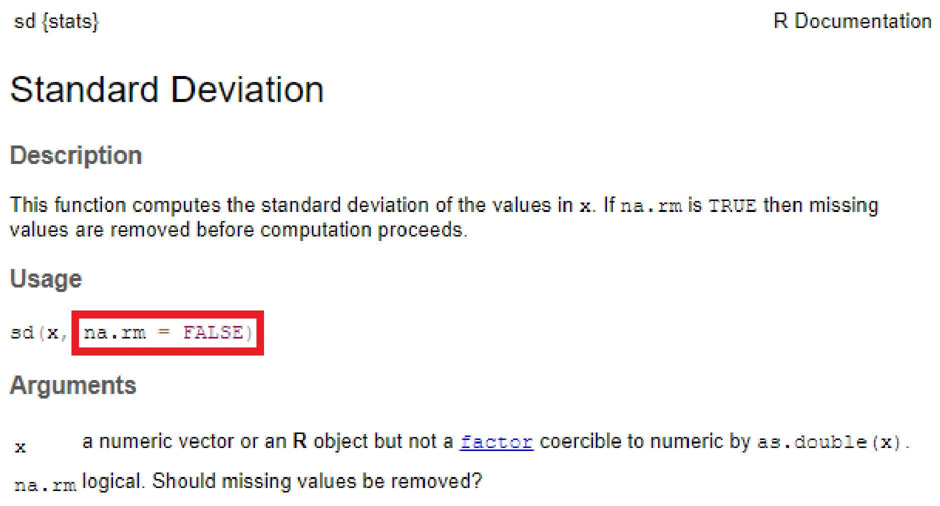
```
MaxMonthtemp_C_Ptbo_2022
```

```
##   Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
##  3.62  5.07 17.67 23.10 29.26 29.55 29.14 30.64 25.20 22.70    NA 14.58
```

Notice that November is NA. NA means the value is missing or not available. NAs are common in data. As such, there are built-in methods to address them.

Review *sd()*'s help page:

```
?sd()
```


sd {stats} R Documentation

Standard Deviation

Description

This function computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

`x`	a numeric vector or an R object but not a `factor` coercible to numeric by `as.double(x)`.
`na.rm`	logical. Should missing values be removed?

The argument na.rm (remove NAs) is set to FALSE by default, meaning *sd()* will not dismiss NAs. If we specify *na.rm = TRUE* ...

```
sd(MaxMonthtemp_C_Ptbo_2022, na.rm = TRUE)
```

```
## [1] 9.662162
```

... a value is returned. All descriptive statistics functions (i.e. *mean()*, *median()*, *var()*) have the na.rm argument. As such, it is a handy argument to remember!

Alternatively, we could replace NA values with the correct value. For more on replacing NAs, please read Replace NA with Zero in R by R-bloggers. Replacing NA values is also covered in MaDGIC's Working with Data in R workshop.

Data frames

Data frames are essential in R. They are the base and most common data structure as they reflect tabular data (i.e. spreadsheets) often applied for data analysis and visualization.

Generally, rows represent observations, and columns represent variables or attributes. We create data frames from vectors of equal length, each representing a single variable.

For example:

```r
Ptbo_Temps <- data.frame(Year = rep(2021, 21),
                         Month = rep("May", 21),
                         Day = 1:21,
                         MeanTemp_C = c(7.585, 6.33, 10.53, 12.32, 11.93, 11.9,
                                        12.935, 13.395, 16.325, 18.035, 19.89,
                                        21.22, 20.085, 15.875, 13.365, 18.945,
                                        18.3, 11.89, 6.32, 9.23, 10.39),
                         MinTemp_C = c(1.98, 1.94, 3.2, 4.65, 3.62, 3.04, 4.14,
                                       4.29, 6.94, 9.19, 12.15, 13.18, 15.73,
                                       9.14, 5.62, 10.72, 12.31, 7.05, 1.88,
                                       0.86, 1.65),
                         MaxTemp_C = c(13.19, 10.72, 17.86, 19.99, 20.24, 20.76,
                                       21.73, 22.5, 25.71, 26.88, 27.63, 29.26,
                                       24.44, 22.61, 21.11, 27.17, 24.29, 16.73,
                                       10.76, 17.6, 19.13),
                         Source = rep("Daymet", 21))
```

Similar to vectors, viewing an entire data frame in the console can be difficult. Therefore, we inspect smaller sections to check the data frame quickly.

head() returns a data frame's first six rows:

```r
head(Ptbo_Temps, n = 6)
```

```
##   Year Month Day MeanTemp_C MinTemp_C MaxTemp_C Source
## 1 2021   May   1      7.585      1.98     13.19 Daymet
## 2 2021   May   2      6.330      1.94     10.72 Daymet
## 3 2021   May   3     10.530      3.20     17.86 Daymet
## 4 2021   May   4     12.320      4.65     19.99 Daymet
## 5 2021   May   5     11.930      3.62     20.24 Daymet
## 6 2021   May   6     11.900      3.04     20.76 Daymet
```

tail() returns a data frame's last six rows:

```r
tail(Ptbo_Temps, n = 6)
```

```
##    Year Month Day MeanTemp_C MinTemp_C MaxTemp_C Source
## 16 2021   May  16     18.945     10.72     27.17 Daymet
## 17 2021   May  17     18.300     12.31     24.29 Daymet
## 18 2021   May  18     11.890      7.05     16.73 Daymet
## 19 2021   May  19      6.320      1.88     10.76 Daymet
## 20 2021   May  20      9.230      0.86     17.60 Daymet
## 21 2021   May  21     10.390      1.65     19.13 Daymet
```

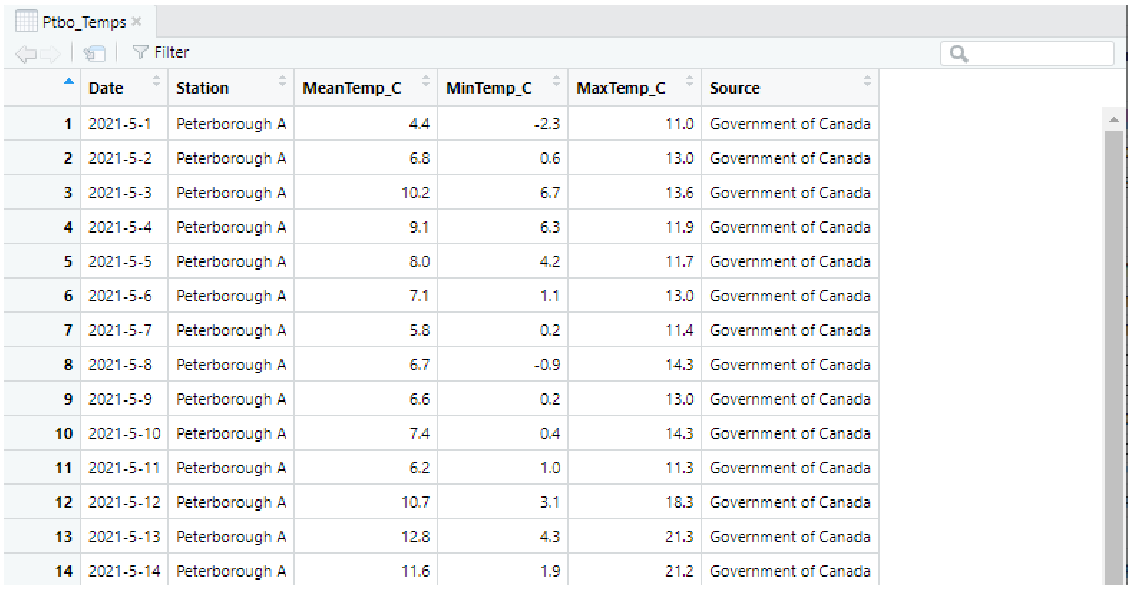
Object structure, *str()*, compactly shows the data frame's internal structure: the number of rows, columns, column names, column data types, and first few values per column.

```r
str(Ptbo_Temps)
```

```
## 'data.frame':    21 obs. of  7 variables:
##  $ Year      : num  2021 2021 2021 2021 2021 ...
##  $ Month     : chr  "May" "May" "May" "May" ...
##  $ Day       : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ MeanTemp_C: num  7.58 6.33 10.53 12.32 11.93 ...
##  $ MinTemp_C : num  1.98 1.94 3.2 4.65 3.62 3.04 4.14 4.29 6.94 9.19 ...
##  $ MaxTemp_C : num  13.2 10.7 17.9 20 20.2 ...
##  $ Source    : chr  "Daymet" "Daymet" "Daymet" "Daymet" ...
```

If you want to view a large data frame in entirety use *View()*. View() opens the data frame in a new tab.

```r
View(Ptbo_Temps)
```



	Date	Station	MeanTemp_C	MinTemp_C	MaxTemp_C	Source
1	2021-5-1	Peterborough A	4.4	-2.3	11.0	Government of Canada
2	2021-5-2	Peterborough A	6.8	0.6	13.0	Government of Canada
3	2021-5-3	Peterborough A	10.2	6.7	13.6	Government of Canada
4	2021-5-4	Peterborough A	9.1	6.3	11.9	Government of Canada
5	2021-5-5	Peterborough A	8.0	4.2	11.7	Government of Canada
6	2021-5-6	Peterborough A	7.1	1.1	13.0	Government of Canada
7	2021-5-7	Peterborough A	5.8	0.2	11.4	Government of Canada
8	2021-5-8	Peterborough A	6.7	-0.9	14.3	Government of Canada
9	2021-5-9	Peterborough A	6.6	0.2	13.0	Government of Canada
10	2021-5-10	Peterborough A	7.4	0.4	14.3	Government of Canada
11	2021-5-11	Peterborough A	6.2	1.0	11.3	Government of Canada
12	2021-5-12	Peterborough A	10.7	3.1	18.3	Government of Canada
13	2021-5-13	Peterborough A	12.8	4.3	21.3	Government of Canada
14	2021-5-14	Peterborough A	11.6	1.9	21.2	Government of Canada

Descriptive statitics in data frames

Data frames consist of different data types, so we cannot calculate the descriptive statistics as seen with vectors. For example, if we wanted to calculate the average value, *mean()*, across the data frame or per column, one may attempt:

```r
mean(Ptbo_Temps, na.rm = TRUE)
```

```
## Warning in mean.default(Ptbo_Temps, na.rm = TRUE): argument is not numeric or
## logical: returning NA
```

[1] NA

Instead, isolate a column with $* * .Placing * *$ after a data frame name allows any column to be considered a vector for application in a function.

```
mean(Ptbo_Temps$MeanTemp_C, na.rm = TRUE)
```

```
## [1] 13.6569
```

Alternatively, you can view descriptive statistics for all columns with *summary()*:

```
summary(Ptbo_Temps)
```

```
##       Year         Month                Day       MeanTemp_C   
##  Min.   :2021   Length:21          Min.   : 1   Min.   : 6.32  
##  1st Qu.:2021   Class :character   1st Qu.: 6   1st Qu.:10.53  
##  Median :2021   Mode  :character   Median :11   Median :12.94  
##  Mean   :2021                      Mean   :11   Mean   :13.66  
##  3rd Qu.:2021                      3rd Qu.:16   3rd Qu.:18.04  
##  Max.   :2021                      Max.   :21   Max.   :21.22  
##    MinTemp_C        MaxTemp_C        Source         
##  Min.   : 0.860   Min.   :10.72   Length:21         
##  1st Qu.: 3.040   1st Qu.:17.86   Class :character  
##  Median : 4.650   Median :21.11   Mode  :character  
##  Mean   : 6.347   Mean   :20.97                     
##  3rd Qu.: 9.190   3rd Qu.:24.44                     
##  Max.   :15.730   Max.   :29.26                     
```

Application: IntordUctory data visualization and analysis

Data visualization The tabular structure of data frames allows us to relate variables for data visualization and analysis.

The following section will review the basics of data visualizations and analysis in R. For more details, please stay tuned for future workshops or use the provided resources.

More often in R, data is imported from a spreadsheet and not manually entered as seen before. Data imported from a spreadsheet will be interpreted as a data frame.

For more information on importing data into R, please see MaDGIC's Working with Data in R workshop or contact madgichelp@trentu.ca.

Let's work with a full Daymet data set. This example data can be downloaded from Github.

```
data_url <- "https://tinyurl.com/z87ekh2u"
daymet_data <- read.csv(url(data_url))
```

The *daymet_data* data frame represents daily climate measures for Peterborough, ON and its sister city, Ann Arbor, MI from 1980 to 2022. Specifically, the *daymet_data* contains the following variables (columns):

- City: the city for which the daily weather data was derived. Either Peterborough, ON, or its sister city, Ann Arbor, MI
- date: Full date of daily weather value
- year: Year value was recorded
- yday: Day of the year of daily weather value
- dayl_s: Duration of the daylight period in seconds per day. This calculation is based on the period of the day during which the sun is above a hypothetical flat horizon

- prcp_mm_day: Daily total precipitation in millimeters per day, the sum of all forms converted to water-equivalent. Precipitation occurrence on any given day may be ascertained.
- srad_W_m2: Incident shortwave radiation flux density in watts per square meter, taken as an average over the daylight period of the day. NOTE: Daily total radiation (MJ/m2/day) can be calculated as follows: ((srad (W/m2) * dayl (s/day)) / l,000,000)
- swe_kg_m2: Snow water equivalent in kilograms per square meter. The amount of water contained within the snowpack
- tmax_degC: Daily maximum 2-meter air temperature in degrees Celsius.
- tmin_degC: Daily Daily minimum 2-meter air temperature in degrees Celsius.
- vp_Pa: Water vapor pressure in pascals. Daily average partial pressure of water vapor.

To understand *daymet_data* better, let's inspect it, First, apply *head()* (or *tail()*) to inspect the data frame. This allows for a quick check that nothing appears wrong.

```r
head(daymet_data)
```

```
##                City       date yday year   daly_s prcp_mm_day srad_W_m2
## 1 Peterborough, ON 1980-01-01    1 1980 31441.25        0.00     66.67
## 2 Peterborough, ON 1980-01-02    2 1980 31490.50        0.00    128.97
## 3 Peterborough, ON 1980-01-03    3 1980 31543.76        0.00    136.67
## 4 Peterborough, ON 1980-01-04    4 1980 31600.99        0.00    215.89
## 5 Peterborough, ON 1980-01-05    5 1980 31662.15        0.00    202.87
## 6 Peterborough, ON 1980-01-06    6 1980 31727.19        4.27    184.30
##   swe_kg_m2 tmax_degC tmin_degC  vp_Pa
## 1     30.42      1.64     -0.89 572.38
## 2     29.92      0.34     -4.81 427.25
## 3     29.92     -8.43    -13.79 210.73
## 4     29.92     -5.77    -16.33 170.70
## 5     29.92     -3.33    -12.56 232.84
## 6     34.20      0.28    -16.21 172.44
```

Next, to ensure each column was assigned the appropriate data type. **This is vital**. If there was a single character value within a numeric field, all numeric values in a single column will be converted to characters.

```r
str(daymet_data)
```

```
## 'data.frame':    31390 obs. of  11 variables:
##  $ City       : chr  "Peterborough, ON" "Peterborough, ON" "Peterborough, ON" "Peterborough, ON" ...
##  $ date       : chr  "1980-01-01" "1980-01-02" "1980-01-03" "1980-01-04" ...
##  $ yday       : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ year       : int  1980 1980 1980 1980 1980 1980 1980 1980 1980 1980 ...
##  $ daly_s     : num  31441 31491 31544 31601 31662 ...
##  $ prcp_mm_day: num  0 0 0 0 0 4.27 0 0 2.37 2.64 ...
##  $ srad_W_m2  : num  66.7 129 136.7 215.9 202.9 ...
##  $ swe_kg_m2  : num  30.4 29.9 29.9 29.9 29.9 ...
##  $ tmax_degC  : num  1.64 0.34 -8.43 -5.77 -3.33 0.28 1.84 -4.82 -6.17 1.69 ...
##  $ tmin_degC  : num  -0.89 -4.81 -13.79 -16.33 -12.56 ...
##  $ vp_Pa      : num  572 427 211 171 233 ...
```

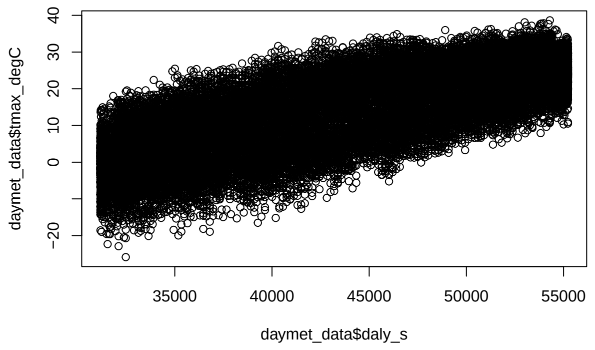
All variables are characters, integers, or numeric. These data types are acceptable for this workshop. If it were not, we would change the columns' data type.

With the data imported and checked, let's visualize the data. We can visualize data using *plot()*, base R's generic x-y plotting function. Plotting requires independently defining the x-axis (horizontal axis) and y-axis (vertical axis) variables or providing a formula.

Suppose we needed to show the relationship between daily daylight and maximum temperature:

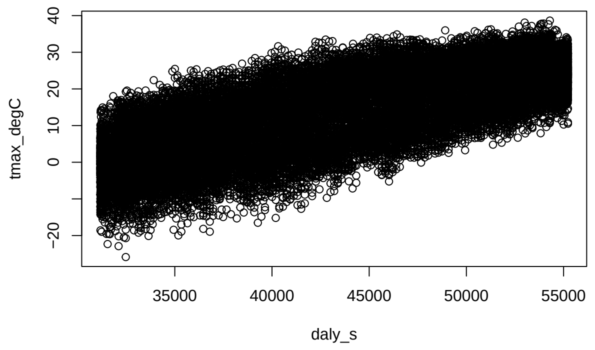
To define each axis independently, we provide a vector (Data frame column) to the x and y arguments.

```
plot(x = daymet_data$daly_s, y = daymet_data$tmax_degC)
```



Alternatively, we define a formula. Formulas in R are defined with variables on either side of the tilde operator, ~. The variable on the left-hand side of a tilde (~) is the dependent or response variable (y-axis), while the variable on the right-hand side are the independent or explanatory variables (x-axis).

```
plot(tmax_degC ~ daly_s, data = daymet_data)
```

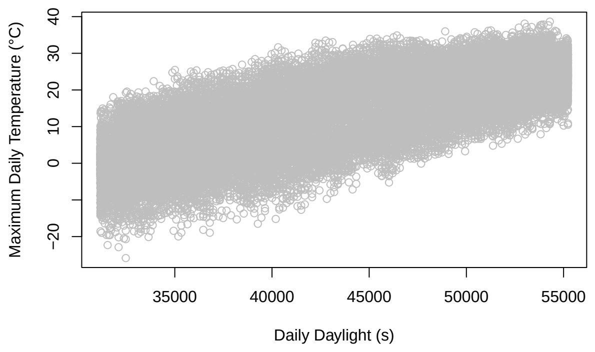


Let's improve this figure by adding appropriate axis labels and coloring the point. We accomplish this by adding arguments to *plot*.

- xlab = a title for the x-axis
- ylab = a title for the y-axis
- col = color of the points

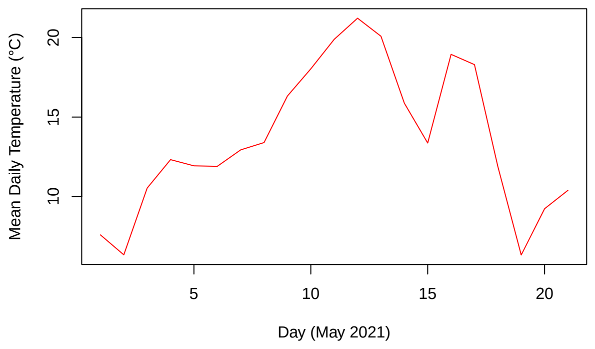
```r
plot(tmax_degC ~ daly_s, data = daymet_data,
     xlab = "Daily Daylight (s)",
     ylab = "Maximum Daily Temperature (\u00B0C)",
      col="grey")
```

Please note “u\00B0” is the unicode for the degree symbol.



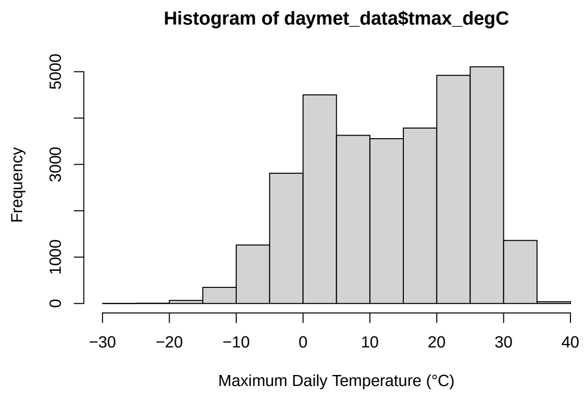
R’s generic plotting defaults to a scatter point. We can change this by altering the *type* argument. Suppose we wanted to draw a line graph indicating the change in mean temperature.

```r
plot(MeanTemp_C ~ Day, data = Ptbo_Temps,
     col="red", type = "l",
     xlab = "Day (May 2021)",
     ylab = "Mean Daily Temperature (\u00B0C)")
```



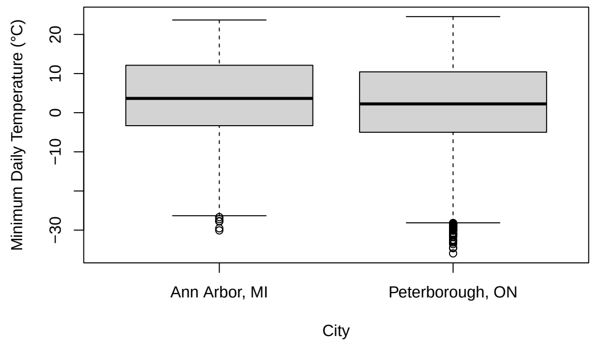
Instead of changing the type command, you can use plot specific commands. For example, to create a histogram use *hist()*:

```r
hist(daymet_data$tmax_degC, xlab = "Maximum Daily Temperature (\u00B0C)")
```



Similarly, generate Boxplots to show the locality, spread, and skewness of data with *boxplot()*:

```
boxplot(tmin_degC ~ City, data = daymet_data,
        ylab = "Minimum Daily Temperature (\u00B0C)", xlab = "City")
```



There are more plotting functions available in base R for quick visualizations. For example:

- Bar plots: *barplot()*
- Violin plots: *vioplot()*
- Dot charts: *dotchart()*
- Conditioning plots: *coplot()*
- Pair plots: *pairs()*
- QQ plots: *qqnorm()*

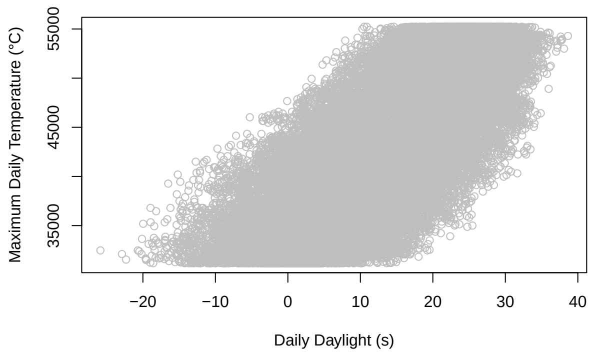
The best plot will depend on your data and the story to tell. If you have any questions or would appreciate assistance visualizing your data, please contact us at madgichelp@trentu.ca.

For more information on data visualization in R, please refer to:

- R Base Plots
- R Graph Gallery
- Data visualization by Hadley Wickham
- The R Book, Chapter 5 : Graphics

Basic Statistical Analysis At its core, R is a statistical software and commonly used to perform statistical tests. Let's explore some basic statistical tests to see how it is done.

Recall the scatter plot of daily daylight and maximum temperature.



Visually there is a linear relationship between daily daylight and maximum daily temperature. We can statistically test the presence and strength of a relationship between two variables with correlation. We can calculate correlation with *cor()* or *cor.test()* for unpaired and paired data. We are considering paired data, as the daily weather data comes from two cities on the exact dates.

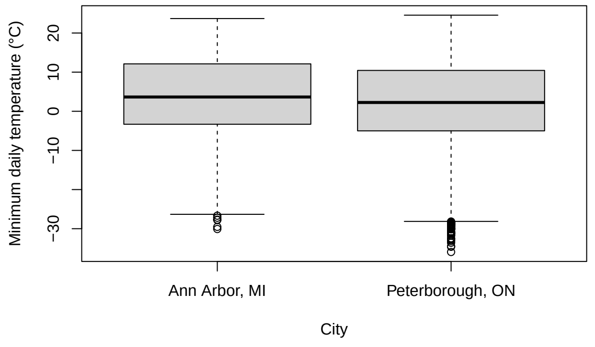
```
cor.test(daymet_data$daly_s, daymet_data$tmax_degC)
```

```
##
##  Pearson's product-moment correlation
##
## data:  daymet_data$daly_s and daymet_data$tmax_degC
## t = 223.21, df = 31388, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7789507 0.7875026
## sample estimates:
##       cor
## 0.7832637
```

This tests returns a p-value less than 2.2e^{-16}, indicating there is a significant linear relationship between daily daylight and maximum daily temperature in the considered cities. Specifically, a strong positive correlation (correlation value (r) of 0.78).

Another common statistical test is to compare a measure of two groups. Comparison of two groups is accomplished with a T-test.

Previously we examined the boxplots for each city's minimum daily temperature:



These plots look quite similar, but we can statistically compare the cities' minimum temperatures through a t-test.

T-tests are conducted with *t.test()*. You can provide a formula (Dependent variable ~ Independent variable or Response variable ~ Explanatory variable) or two vectors to compare.

```r
t.test(tmin_degC ~ City, data = daymet_data)
```

```
##
##  Welch Two Sample t-test
##
## data:  tmin_degC by City
## t = 17.364, df = 31228, p-value < 2.2e-16
## alternative hypothesis: true difference in means between group Ann Arbor, MI and group Peterborough,
## 95 percent confidence interval:
##  1.788128 2.243178
## sample estimates:
##    mean in group Ann Arbor, MI mean in group Peterborough, ON
##                       3.672312                       1.656659
```

The T-test's output indicates a t-statistic of 17.36 with 3.1228116×10^4 degrees of freedom and a p-value less than 2.2e^{-16}. Therefore, there is a significant difference between the minimum temperatures of Peterborough, ON, and its sister city, Ann Arbor, MI, during 1980-2022. Examining the T-test's sample estimates indicate Peterborough observed a 2°C lower minimum temperature compared to Ann Arbor, MI (mean Ann Arbor, MI minimum temperature: 3.67°C, mean Peterborough, ON minimum temperature: 1.66°C).

These are two simple examples of data analysis in R. For more information on statistical analysis, please refer to:

- Basic Statistics (Using R) by Michael Mahoney
- The R Book, Chapters 9 to 23

If you have any questions about R, this workshop or future workshops, please do not hesitate to contact MaDGIC at madgichelp@trentu.ca.

References:

Daymet: Daily Surface Weather Data on a 1-km Grid for North America, Version 4 R1 https://doi.org/10.3334/ORNLDAAC/2129