The Book of R by Tilman M. Davies [519.50285 DAV] *and additional sources*

| **Keystroke Shortcuts** | | |
|---|---|---|
| | | Notes: |
| CTRL-R | Automatically sends lines from built-in editor to the console | - Can send the line upon which the cursor sits<br>- Executes everything that is highlighted |
| CTRL-S | To save a script from the built-in editor | Ensure the editor is selected |
| CTRL-O | To open a previously saved script | |
| CTRL-L | Clears console | |
| [Beginning of command] + (CTRL-<UP>) | To find a more specific command | E.g. "*qp*" of "*qplot*" |
| [Beginning of function] + TAB | Auto-completes function | |
| [Function] + ( + TAB | Show function's arguments | |
| CTRL + SHIFT-S | *Source*: Runs all the code in the editor in the background | To run all the code and have all the output **shown**, use "*Source with Echo*" button |
| CTRL + 1 | Go to source pane (make it active) | |
| CTRL + 2 | Go to console pan | |
| [Highlight function] + F1 | Alternative way to search for help | Package for function needs to be loaded first |
| CTRL + SHIFT-M | Shortcut for pipe operator ( %>% ) | |
| ALT + ( - ) | Shortcut for assignment operator ( <- ) | |
| CTRL + SHIFT-C | **Comment** lines of code ( # ) | To uncomment, do it again |
| CTRL + SHIFT + ( / ) | Shortcut to **reflow** comment: makes a single line of commented code easier to read | |
| CTRL + SHIFT-Z | Redo: opposite of undo | |
| CTRL-ALT + [Click where you want] | Creates a 2nd or more cursor | To return to a single cursor, single click inside the source |
| CTRL-P | Jumps between parentheses | |
| CTRL + SHIFT + ENTER | Runs the entire file | |
| CTRL + SHIFT-F<br>CTRL + ( . ) | Finds text across all of the file in the project directory | |
| [Beginning of function] + TAB | Quick way to create a function using a pre-defined structure | |
| *Magic wand* button | Extract function | E.g. New function: *rescale(center(x))*, where *rescale* and *center* are functions already defined |
| Home: Fn + (←) arrow | Jump cursor to beginning of line | |
| End: Fn + (→) arrow | Jump cursor to end of line | |

| Tool Commands | | |
|---|---|---|
| getwd() | Checks the location of the working directory | |
| setwd(“…”) | Change the default working directory | E.g. “/folder1/folder2/folder3/” |
| library(“…”) | To load any package and gain access to its functions and data sets | E.g. “MASS”<br>- Only provides access for the running R session<br>- A new session requires you to reload the package |
| require(“…”) | Loads packages into R session that is not yet installed | - Can only load 1 package at a time (same goes for *library*)<br>- Use *result <- require(“…”)* to see if packages is successfully attached<br>- *result= FALSE* signifies failure |
| install.packages(“…”) | Package installation | - Requires internet connection<br>- Need only to install a package once |
| update.packages() | Check for updates to your collection of installed packages | |
| help / ?…<br>help.search / ??”…” | To seek information on precisely how to use a given function and specify its arguments, to clarify its role etc. | |
| ls() | Lists all objects, variables, and user-defined functions currently present in the active workspace/environment | |
| search() | Search list – Shows packages already loaded in the environment (including the 7 packages already loaded in R by default) | |
| save.image<br>load | Both functions contain a *file* argument to which you pass the folder location and name of the target *.RData* file | |
| **saveRDS**(<file name>, file= “<filename>.rds”)<br><br><new file name> <- **readRDS**(“<file name>.rds”) | Allows for *an* object to be saved and retrieved for use | - Useful in a loop<br>- Assigning the same name will overwrite the old version<br>- Preferred to *save()* and *load()*<br>- Note that <*file name*> includes path to the file from the working directory if the file is not from there<br>- E.g. *“../data/raw_hawker.rds”* goes one directory up (by ../) and into the folder *data* for the file |
| save(…)<br>load(…) | Allows for *multiple* objects to be saved into a file at a go | |
| browser() | Enter debugging mode | - A call to *browser* can be included in the body of a function<br>- When reached, this causes a pause in execution of the current expression and allows access to the R interpreter |
| options(…)<br>options(error= NULL) | Allows the user to set and examine a variety of global *options* which affect the | - Invoking *options()* with no arguments returns a list with the current values of the options |

| | | |
|---|---|---|
| | way in which R computes and displays its results<br>Restores R’s default behaviour | - Note that not all options listed below are set initially |
| q() | Quickest way to exit the software using a prompt | |

| **Functions** | | |
|---|---|---|
| sqrt(x =…) | Finds the square root of any non-negative number | |
| log( x =… , base =…) | Log transformation where default base is natural log e | - Both x and the base must be positive<br>- The log of any number *x* when the base is equal to *x* is 1<br>- The log of *x* = 1 is always 0, regardless of base |
| exp(x =…) | Exponential function | |
| rnorm(n, mean= …, sd= …) | Generates *n* number observations from a normal random variable of *mean* … and standard *deviation* … | |
| runif(n, min= x, max= y) | Generates random deviates of a uniform distribution | - Argument *n* is the number of observations<br>- *min*, *max* is the lower and upper limits of the distribution<br>- Note that when *min* or *max* is not specified, they assume the default values of *0* and *1* respectively<br>- Also, *runif* will not generate either of the extreme values unless *max=min* or *max-min* is small compared to *min* |
| sample(x : y, size= z, replace= T/F) | Random number generator - carries out sampling from a given vector *x : y* | - Function will give z numbers given by *size*<br>- For sampling with replacement, use *replace= TRUE* |
| mean(x= …, trim= …, na.rm= T/F) | Finds the average of values in a vector | - *trim* chops off a fraction of observations from each end of x before computation<br>- Default for *trim= 0* but takes values:<br>*0 < trim < 0.5*<br>- *na.rm* decides whether to remove NA values in the vector automatically or not |
| set.seed(…) | Helps to reset random number generator with an integer as the argument | - Helpful when working and coordinating with people, the random numbers will be generated with the same sequence (especially when there is a bug with a certain seed) |
| table(…) | Factorises a vector with discrete values and counts the number of occurrences of each value into a contingency table | |
| tail(…, x) | Takes the last *x* elements of an object, e.g. vector | |
| summary(…) | Gives a quick overview of contents in an object | |
| subset(…, subset= <some condition>) | Returns subset of vectors, matrices or data frames which meet conditions | |
| order(…, decreasing= T/F) | Returns a vector of positions which tell the rank of each element based on value | |
| print(…) | **Prints** to console **argument**, does not join output | |

| | | |
|---|---|---|
| paste(…) | **Converts** vectors to **character** and then **concatenates** them | |
| cat(…) | **Output** objects after **concatenating** representations | |
| args(…) | Gives the argument of a function | |
| unlist(…) | Turns a list into a single vector | |
| tolower(…) | Converts a character vector to lowercase letters | |
| str(…) | Prints structure of object and returns *NULL* | |
| nchar(…) | Returns the length of string | |
| identical(…, …) | Compare and test for 2 objects being *exactly* equal | |
| unique(x) | Returns a vector, data frame or array like *x* but with duplicate elements/rows removed | |
| arrayInd(ind, .dim) | Gives the *TRUE* indices of a logical object, allowing for array indices<br>Finds the exact **coordinates** of an element in a matrix (by row and column) | - *ind* is a integer-valued index vector obtained from *which(x)*<br>- *.dim* is an integer vector (i.e. the dimensions of the matrix) |
| mat[…, drop= F] | Tells R not to remove dimension names when subsetting a matrix | |
| diff(my_vec) | Finds the difference between consecutive elements of a vector | |

| **Vector** | | |
|---|---|---|
| c(…, …, …) | Creates a vector with desired entries in parentheses separated by commas | Vector entries can be calculations, or previously stored items (including vectors themselves) |
| seq(from =…, to = …, by =…) / seq(from =…, to =…, length.out =…) | Returns a corresponding sequence using the arguments as a numeric vector | - Note that sequence will always start at the *from* number but will not always include the *to* number, depending on what you are asking R to increase (or decrease) *by*<br>- A *length.out* value can be specified instead of a *by* value to produce a vector with that many numbers, evenly spaced between the *from* and *to* values<br>- For decreasing sequences, the use of *by* must be negative<br>- The use of *length.out* to create decreasing sequences is to ensure that the *from* value is greater than the *to* value<br>- Used in plots (tick marks and gridlines) and indexing |
| rep(x =…, times =…, each =…) | Value/ vector repetition | - Argument *x* can be a single value or a vector of values<br>- Argument *times* provides the number of times to repeat *x*<br>- *times* can also be a vector to specify the number of times to repeat each element<br>E.g.<br>*x<- x <- c(1, 1.5, 2.0)*<br>*rep(x, times= c(1, 2, 3))*<br>gives *1.0 1.5 1.5 2.0 2.0 2.0*<br>- Note that vector length must match the length of vector *x*<br>- Argument *each* provides the number of times to repeat each element of *x*<br>- *times* and *each* can also be used separately<br>- If neither *times* nor *each* is specified, R's default is to treat the values of them as 1 |
| sort(x =…, decreasing = FALSE/TRUE) | Vector sorting in increasing/ decreasing order | Argument *x* is a vector |
| length(x =…) | Determines how many entries exist in a vector given the argument *x* | Note that if including entries that depend on the evaluation of other functions (e.g. *rep* and *sep*), *length* tells you the number of entries *after* those inner functions have been executed |
| sum(foo) | Finds the sum of elements in *foo* vector | |
| prod(foo) | Finds the product of elements in *foo* vector | |

| | | |
|---|---|---|
| Colon ( : ) / *x* : *y* | Creates a sequence/ vector with numerical values separated by intervals of 1 | - For example, *3 : 27* is “from 3 to 27 by 1”<br>- *x* and *y* can be a previously stored value or a (strictly parenthesized) calculation<br>- *x* and *y* are inclusive |
| Square brackets [ ] / myvec[x] | Vector subsetting/ extraction: Where *x* being an index allows retrieving of specific elements from a vector | - By using ***negative*** versions of the indexes supplied in the square brackets, individual elements can be deleted ***<br>- Using the [ ] operator does not change the original vector *unless* you explicitly overwrite the vector with the subsetted version<br>- *x* can also be vectors of indexes / using [*x* : *y*] to extract or delete more than one element<br>- The vector *x* can include repetition of indexes to form a new vector<br>- Note that it is not possible to mix positive and negative indexes in a single index vector |
| bar[x] <- y | Overwrites certain elements in an existing vector | - *x* and *y* can be vectors / vectors created using colons<br>- Overwriting multiple elements but with a pattern can be done:<br>*foo <- c(**2**, 4, **6**, 8, **10**, 12, **14**, 16)*<br>*foo[c(1, 3, 5, 7)] <- c(0, 1)*<br>*foo = c(**0**, 4, **1**, 8, **0**, 12, **1**, 16)*<br>- Note that the length of the vector of replacements must evenly divide the number of elements being overwritten |
| Minus ( - ) | Can be used to minus a vector of values from a vector | - It is fine as long as the length of the longer vector can be evenly divided by the length of the shorter vector<br>- R will attempt to replicate the shorter vector as many times as needed to match the length of the longer vector before completing the specified operation<br>- For example, doing something different to alternating entries of the vector:<br>*c(1, 2, 3, 4) * c(1, -1)*<br>*= c(1, -2, 3, -4)*<br>- While substracting a shorter vector from a longer one:<br>*c(0, 1) – c(1, 2, 3, 4)*<br>*= c(-1, -1, -3, -3)* |

| | | |
|---|---|---|
| | | - When you just want to add the same value to all the entries in vector:<br>*c(1, 2, 3, 4) + 1*<br>*= c(2, 3, 4, 5)*<br>- When vector lengths are not evenly divisible, a warning message will occur in R |

| **Matrix** | | |
|---|---|---|
| matrix(data= c(…), nrow= …, ncol= …, byrow= FALSE/TRUE, dimnames= list(c(…), c(…)) | Creates a matrix in R | - Make sure length of vector matches exactly with the number of desired rows (*nrow*) and columns (*ncol*)<br>- Without *nrow* and *ncol*, R's default behavior is to return a single-column matrix<br>- *c(1, 2, 3, 4)* returns $\begin{matrix}1 & 3\\ 2 & 4\end{matrix}$, where *nrow*, *ncol*= 2<br>- Default for R is to fill in data column-by-column<br>- Set optional argument *byrow= TRUE* to fill in structure row-wise<br>- The argument for *dimnames* is a list of 2 **character** vectors of appropriate length (i.e. matrix dimensions) – 1st the row names and 2nd the column names |
| rbind(v1, v2…),<br>cbind(v1, v2…) | Binds together vectors of equal length as rows or columns | For example, *rbind(1:3, 4:6)* gives<br>$\begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\end{matrix}$ |
| dim(mymat)<br>dim(mymat)[x] | Provides dimensions of matrix stored in your workspace, the number of rows and number of columns | Vector subsetting can be used to extract out only the number of rows or columns where *x* is either 1, rows or 2, columns |
| nrow(mymat)<br>ncol(mymat) | Provide the number of rows and columns only respectively | |
| diag(x= A) | Returns a vector with elements along the diagonal of matrix A, starting at A[1,1] | If A is a **value**, e.g. 3, it will **create an identity matrix** of the corresponding dimension, i.e.<br>$\begin{matrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{matrix}$ |
| t(mymat) | Transpose a matrix | |
| solve(A) | Gives the inverse of matrix A, i.e. $A^{-1}$ | - Note that matrices that are not invertible are referred to as singular<br>- An invertible matrix satisfies the following equation:<br>$AA^{-1} = I_m$ |
| rowname(A) <- …<br>colnames(A) <- … | Renaming row and column names | |
| dimnames(A)<br>dimnames(A) <- … | Gets array dimension names | Assignment operator can be used to rename the members of the list |
| rowSums(A)<br>colSums(A) | Sums up the elements row and column-wise respectively | |
| Matrix subsetting, A[x, y] | Gives position of element at position *[x,y]* in the matrix | - Where *x* refers to the row and *y* refers to the column, i.e. *[row, column]*<br>- To extract an entire row or column, specify the desired row or column number and leave the other value blank, i.e. *[x, ]* or *[ ,y]* |

| | | |
|---|---|---|
| | | - *x* and *y* can be adjusted to return multiple rows or columns, i.e. *[2:3, ]* for the 2nd and 3rd row or *[ ,c(3, 1)]* for the 3rd and 1st column in a new matrix<br>- *A[c(3,1), 2:3]* will access the 1st and 3rd rows of matrix A, and from those rows it returns the 2nd and 3rd column elements<br>- To ***delete or omit elements*** from a matrix, use ***negative*** indices *** |
| A[x, y] <- z | Overwrites particular elements, or entire rows or columns in a matrix with new elements of a single value, a vector of the same length as the number of elements to be replaced, or a vector whose length evenly divides the number of elements to be replaced | - For example, let matrix B be<br>$\begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{matrix}$<br>- Overwrite the 2nd row of B with sequence 101, 102, 103 with *B[2, ] <- 101:103*<br>- B is now $\begin{matrix}1 & 2 & 3\\ \mathbf{101} & \mathbf{102} & \mathbf{103}\\ 7 & 8 & 9\end{matrix}$<br>- Overwrite the 2nd column elements of the 1st and 3rd rows with 999 with *B[c(1,3), 2] <- 999*<br>- B is now $\begin{matrix}1 & \mathbf{999} & 3\\ 101 & 102 & 103\\ 7 & \mathbf{999} & 9\end{matrix}$<br>- Replace the 3rd column of B with the values in the 3rd row of B with *B[ ,3] <- B[3, ]*<br>- B is now $\begin{matrix}1 & 999 & \mathbf{7}\\ 101 & 102 & \mathbf{999}\\ 7 & 999 & \mathbf{9}\end{matrix}$<br>- Using R's vector recycling to overwrite the 1st and 3rd column elements of rows 1 and 3 (total of 4 elements) with values -4 and 4 with *B[c(1, 3), c(1, 3)] <- c(-7, 7)*<br>- B is now $\begin{matrix}\mathbf{-4} & 999 & \mathbf{-4}\\ 101 & 102 & 999\\ \mathbf{4} & 999 & \mathbf{4}\end{matrix}$<br>- Note that the vector of length 2 replaced the 4 elements in a ***column-wise*** fashion ***<br>- Overwriting the 1st and 3rd element in the 2nd column first followed by the 1st column with *B[c(1, 3), 2:1] <- c(67, -67, 78, -78)*<br>- B is now $\begin{matrix}\mathbf{78} & \mathbf{67} & -4\\ 101 & 102 & 999\\ \mathbf{-78} & \mathbf{-67} & 4\end{matrix}$<br>- Note that overwriting ***proceeds accordingly*** in the order of columns or rows specified *** |
| diag(x= B) <- … | Replace the diagonal of a square matrix | |

| | | |
|---|---|---|
| upper.tri(B, diag= T/F)<br>lower.tri(B, diag= T/F) | Returns a matrix of logicals the same size of a given matrix with entries _TRUE_ in the lower or upper triangle | _diag_= _F_ is the default to exclude the diagonal |
| Scalar (*) | Multiplication of any matrix by a scalar value _a_, will result in a matrix in which every individual element is multiplied by _a_ | E.g. _2*A_ where A is a matrix |
| Addition/Subtraction (+ / -) | Add or subtract any 2 equally sized matrices with the standard + and - symbols | |
| Matrix multiplication (%*%) | Multiply 2 compatible matrices A and B of size _m_ by _n_ and _p_ by _q_ where n = p, the resulting matrix A·B will have the size _m_ by _q_ | E.g. _A%*%B_<br>- The elements of the product are computed in a column-wise fashion<br>- Note that this property is not commutative, i.e. A·B ≠ B·A |

| **Multidimensional Arrays** | | |
|---|---|---|
| array(data= …, dim= c(…)) | Creates a data structure in R where the individual elements are specified in the *data* argument as a vector and the size is specified in the *dim* argument as another vector | - Note that *array* ***fills the entries*** of each layer with the elements in data in a strict ***column-wise*** fashion, starting with the first layer<br>- E.g. *AR <- array(data=1:12, dim= c(2, 2, 3))*<br>Where AR is<br>*, , 1*<br>1 3<br>2 4<br>*, , 2*<br>5 7<br>6 8<br>*, , 3*<br>9 11<br>10 12<br>- For a 3-dimensional array, the argument for *dim* is: *c(rows, columns, layers)*<br>-<br>- As you increase the dimension further, the dim vector must be extended accordingly<br>- For example, a 4-dimensional array can be thought of as ***blocks*** of 3-dimensional arrays<br>- Its *dim* vector can be something like *c(2, 2, 3,* ***3****)* which will result in 3 copies of AR where each of these copies is split into its 2 layers<br>- R will print it as such:<br>*, , 1, 1*<br>…<br>*, , 2, 1*<br>…<br>*, , 1, 2*<br>…<br>*, , 2, 2*<br>…<br>*, , 1, 3*<br>…<br>*, , 2, 3*<br>…<br>- The rows are indexed by the 1st digit, the columns by the 2nd digit, the layers by the 3rd digit, and the blocks by the 4th digit |

| | | |
|---|---|---|
| | | i.e. ***c(rows, columns, layers, blocks)*** ***<br>- Note that the *array* function can be used to create 1-D (vectors) and 2-D (matrices) arrays, but vectors in particular may be treated differently by some functions if created with *array* instead of *c* |
| AR[a, b, c, d] | Subsets, extractions and replacements where *AR* is an array in R | - Where *[a, b, c, d]* is *[rows, columns, layers, blocks]* and any of them can be omitted as long as the commas are left<br>- An extraction that results in multiple vectors will be presented as columns in the returned matrix<br>- For example, the returned object as the first rows of each of the two matrix layers in a 4-D array. R will return each of these vectors as a *column* of a single returned matrix<br>- *BR[, , 2, ]* will return all the values in the second layer of the array *BR*<br>- Broadly speaking, if you have an extraction that results in multiple *d*-dimensional arrays, the result will be an array of the next-highest dimension, *d + 1*<br>- Deleting and overwriting elements in high-dimensional arrays follow the same rules as stand-alone vectors and matrices: use negative indices or the assignment operator |

| Logical Values | | | | | |
|---|---|---|---|---|---|
| TRUE / FALSE | Logical values in R which can be abbreviated and used as *T* or *F* as long as no objects are created named T or F | | | | |
| Relational operators | Operator | Interpretation | | | - Typically used on numeric values but also possible on vectors, matrices and arrays<br>- R will check whether the corresponding entries in the vectors, matrices or arrays are equal/less than/etc. and will return logical results in the same data structure<br>- Vector recycling also applies to logicals<br>- For example, all the values of a vector can be checked against a single value, i.e. *foo < 3* |
| | == | Equal to | | | |
| | != | Not equal to | | | |
| | > | Greater than | | | |
| | < | Less than | | | |
| | >= | Greater than or equal to | | | |
| | <= | Less than or equal to | | | |
| any(foo) | Returns ***TRUE* if any** of the logicals in the vector, matrix or array are ***TRUE*** and returns *FALSE* otherwise | | | | Used to quickly inspect a **collection of logical values** |
| all(bar) | Returns a ***TRUE* only if all** the logicals are ***TRUE***, and returns *FALSE* otherwise | | | | |
| Logical operators | Operator | Interpretation | Results | | - Used to compare two TRUE or FALSE objects<br>- The result of using any logical operator is a logical value<br>- You can combine these operators to examine multiple conditions at once (with the use of parentheses)<br>- As with numeric arithmetic, there is an order of importance for logical operations in R<br>- Helpful to place each comparative pair in parentheses to preserve the correct order of evaluation and make the code more readable |
| | & | AND (element-wise) | TRUE & TRUE | is TRUE | |
| | | | TRUE & FALSE | is FALSE | |
| | | | FALSE & TRUE | is FALSE | |
| | | | FALSE & FALSE | is FALSE | |
| | && | AND (single comparison) | Same as & above | | |
| | \| | OR (element-wise) | TRUE \| TRUE | is TRUE | |
| | | | TRUE \| FALSE | is TRUE | |
| | | | FALSE \| TRUE | is TRUE | |
| | | | FALSE \| FALSE | is FALSE | |
| | \|\| | OR (single comparison) | Same as \| above | | |
| | ! | NOT | !TRUE | is FALSE | |

<table>
<tr><td></td><td></td><td></td><td>!FALSE</td><td>is TRUE</td><td>- ( ! ) can also be used on in front of a logical vector, e.g. <i>!logicvec</i></td></tr>
<tr><td>Element-wise comparisons</td><td colspan="2">When you have two logical vectors and you want <b>multiple</b> logicals as a result</td><td colspan="3">It's possible to compare a single pair of logicals using & or | but it's better practice to use && or || when a single TRUE/FALSE result is needed</td></tr>
<tr><td>Single comparisons</td><td colspan="2">When you compare two individual values and R returns a <b>single</b> logical value</td><td colspan="3">When comparing two vectors of equal length using && or ||, R will only compare the first pair of logicals in the two vectors, i.e. the first element of each vector</td></tr>
<tr><td>myvec[c(T, F, …)]</td><td colspan="2">Logical subsetting and extraction where logical <i>flag</i> vectors can be supplied, an element is extracted if the corresponding entry in the flag vector is TRUE or T</td><td colspan="3">- Logical flag vectors should be the same length as the vector that's being accessed (though recycling does occur for shorter flag vectors)<br>- When you want to extract elements based on whether they satisfy a certain condition (or several conditions), apply the condition to the vector to get the logicals instead of the cumbersome method of entering <i>T</i>s and <i>F</i>s yourself<br>- For example, to get the negative elements from<br><i>myvec <- c(5, -3, 6, -9, 2, 7, -1)</i><br><i>myvec < 0</i> gives<br><i>FALSE TRUE FALSE TRUE FALSE FALSE TRUE</i><br>Hence <i>myvec[myvec <b>< 0</b>]</i> gives<br><i>-3 -9 -1</i><br>- More complicated extractions can be done using relational and logical extractions, e.g. <i>mymvec[(myvec >0) <b>&</b> (myvec <1000)]</i><br>- Extraction of elements from matrices and arrays is the same<br>- For instance, <i>A[1, 2:3]</i><br>is the same as <i>A[c(T, F, F), c(F, T, T)]</i></td></tr>
<tr><td>myvec[<i>condition</i>] <- …</td><td colspan="2">Overwrite specific elements that satisfy a condition using a logical flag vector, just as with index vectors</td><td colspan="3">- The logical flag vector is supplied by applying the condition to the vector<br>- Note that you cannot directly use negative logical flag vectors to delete specific elements, this can be done only with numeric index vectors</td></tr>
<tr><td>which(x= c(…))<br>which(x= c(…), arr.ind= T/F)</td><td colspan="2">Converts a logical flag vector into a numeric index vector, the <i>which</i> function will return the indexes corresponding to the positions of any and all <i>TRUE</i> entries</td><td colspan="3">- The <i>which</i> function can be used to identify and delete elements based on logical flag vectors<br>- For example, <i>myvec[ <b>-</b>which(x= myvec <0) ]</i></td></tr>
</table>

| | | |
|---|---|---|
| | | - When used on 2-D objects or higher, R will treat multi-dimensional objects as single vectors (laid out column after column) and then return the vector of corresponding indexes<br>- Setting the optional argument *arr.ind* (array indexes) to TRUE will make R treat the object as a matrix or an array rather than a vector, and provide the row and column positions of the elements requested |

| **Characters** | | |
|---|---|---|
| Double quotation marks<br>(“ ”) | Entering text between a pair of quotes creates a string in R | - R treats a string as a single entity, hence it will be a vector of length 1<br>- Almost any combinations of characters, including numbers, can be a valid character string<br>- Exceptions are the backslash ( \ ), also called an *escape* |
| nchar(x= mystr) | Counts the number of individual characters in a string – returns length of string | |
| == != <br>> < <br>>= <= | Relational operators can be used on strings too | - R considers letters can come later in the alphabet to be greater than earlier letters, i.e. ***b>a*** is *TRUE*<br>- Uppercase letters are considered greater than lowercase letters, i.e. ***A>a*** is *TRUE* |
| cat(…)<br>cat(…, sep= “…”) | Concatenate one or more strings and send its output directly to the console screen and doesn’t formally *return* anything | - When calling cat or paste, pass arguments to the function in the order you want them combined<br>- For example,<br>*cat(“R“, “is“, “fun“, “!”)*<br>gives *“R is fun !”*<br>- An optional argument, *sep*, is used as a separator between strings as they’re concatenated<br>- Pass *sep* a character string (or an empty string, “”) and R will place this string between all other strings provided to *cat* or *paste*<br>- By default, the *sep* argument is a space<br>- R can automatically coerce results from functions or calculations into character strings<br>- For instance,<br>*cat(“The value is”, **a**, “.”)*<br>can give *“The value is 3.”*<br>when the value stored as ‘*a*’ is 3<br>- Accepts any number of arguments, i.e. can join more than 2 inputs which can also be vectors<br>E.g.<br>*paste(“A”, 1:6, sep= “”)*<br>(“A” recycled to a vector of length 6)<br>gives *“A1” “A2” … “A6”*<br>- Used in labels in plots, tickmarks, data set |
| paste(…)<br>paste(…, sep= “…”) | Concatenates one or more vectors after converting to character and then **returns** the final character string as a usable R object | |
| substr(x= mystr, start= a, stop= b)<br>substr(…) <- … | Extract part of the string *mystr* between the two character positions (inclusive), indicated with numbers *a* and *b* passed as *start* and *stop* arguments | - Function can also be used with the assignment operator to directly substitute in a new set of characters |

| | | |
|---|---|---|
| | | - In this case, the replacement string should contain the same number of characters as the selected area<br>- If replacement string is *longer*: replacement still takes place but cuts off any characters that overrun the number of characters replacing<br>- If replacement string is *shorter*: replacement ends when the string is fully inserted, leaving the original characters up to *stop* untouched |
| sub(pattern= …, replacement= …, x= …) | Searches a given **string** *x* for a smaller string *pattern* contained within and replaces the first instance with a new string *replacement* | - Replacement string need not have the same number of characters as the pattern being replaced<br>- To permanently replace a character or string, use the assignment operator<br>- Check out *?substr*, *?sub* for more details, the *grep* command and its variants with *?grep* |
| gsub(pattern= …, replacement= …, x= …) | Does the same thing as *sub* function except replaces every instance of *pattern* | |

| **Factors** | | |
|---|---|---|
| factor(x= …) | Creates a factor vector where the argument for *x* is a vector of values, in integers (e.g. 0, 1, 2) or in character strings (e.g. male, female) | - Data where all possible values fall into a finite number of categories are best represented in R using factors<br>- Typically created from a numeric or character vector<br>- Can only take the form of vectors, not matrices and multi-dimensional arrays<br>- Functions which can be used: *length* and *which* and relational operators |
| Levels | An important piece of information/attribute that a factor object contains, which stores the possible values in the factor | Printed at the bottom of each factor vector |
| Categorical variables | Nominal | Ordinal |
| | - Categorical variables without an implied order<br>- Impossible to say that "one is worth more than the other"<br>- For example, animals | - Have a natural ordering<br>- For instance, temperature: low, medium, high |
| levels(x=…) | Gets the levels of a factor | |
| levels(x= …) <- … | Relabels a factor's levels with a vector of new labels | |
| factor(x= …, levels= …, ordered= TRUE/FALSE) | Defining and ordering levels by supplying a character vector of all possible values to the *levels* argument and instruct R to order the values precisely as they appear by setting the *ordered* argument to TRUE | - *ordered= TRUE* meaning some levels considered "higher than" or "following" others<br>- The strict ordering of the levels is shown by the ( < ) symbol in the object of the output, i.e. …<…<… |
| myfac[…] | Factor-valued vectors are subsetted in the same way as any other vector | - For example, subsetting using a vector of indexes or logical values<br>- After subsetting a factor object, the object continues to store all defined levels even if some of the levels are no longer represented in the subsetted object<br>- Note that the levels of a factor are stored as character strings, even if the original data vector was numeric, e.g. "2" instead of 2 |
| **1)** myint <- c(mob.fac, new.values)<br>**2)** mob.new <- levels(mob.fac)[myint]<br>**3)** mob.new.fac <- factor(x= mob.new, levels= levels(mob.fac), ordered= TRUE)<br><br>where new.values is a factor too | Combining vector-valued vectors in 3 steps:<br>1) Obtain the numeric index positions of each entry with respect to the factor levels<br>2) Use this numeric vector on the levels of the original factor to result in a vector with all the observations combined (however, this is still stored in strings and not factor values)<br>3) Turn the vector object into a factor object<br><br>*OR*<br><br>*newvalues <- **factor**(x= **c**(…))* | - Note that simply using the c function to combine is not possible because the function interprets factors as integers and results in a numeric vector<br>- E.g. *[1] 4 1 12…*<br>- The steps help ensure that the levels are consistent and the observations are valid in the final product |

<table>
<tr>
<td></td>
<td>*newcombinedfac <-*<br>*factor(***levels***(originalfac)***[***c(originalfac,*<br>*newvalues)***]***)*</td>
<td></td>
</tr>
<tr>
<td>cut(x= …, breaks= …,<br>right= T/F,<br>include.lowest= T/F,<br>labels= …)</td>
<td>Creates a factor from data which was originally measured on a continuum by molding the data into discrete factor categories<br><br>Argument for *x* is the vector of observations<br>*breaks* is a vector of desired breal intervals<br>*right* sets the boundary intervals and when:
<table>
<tr><td>TRUE</td><td>default: inclusive-exclusive, i.e. ***(****a,b***]**</td></tr>
<tr><td>FALSE</td><td>***[****a,b***)**</td></tr>
</table>
*include.lowest* when set can be used to include the _____ value:
<table>
<tr><td>when *right* is:</td><td>TRUE</td><td>FALSE</td></tr>
<tr><td>TRUE</td><td>highest</td><td>lowest</td></tr>
<tr><td>FALSE</td><td>lowest</td><td>highest</td></tr>
</table>
*labels* is a character string vector containing better labels to categories, rather than the default interval levels (order of labels must match the order of the levels in the factor object)</td>
<td>- Used at times when continuous observations need to be grouped (or binned) into categories<br>- E.g. small/medium/large or low/high<br>- For example, if the desired interval is<br>$0 < x \leq 2, 2 < x \leq 4, 4 < x \leq 6,$<br>the argument for breaks is<br>*c(0, 2, 4, 6)*<br>- Help file: *?cut*</td>
</tr>
</table>

| **Lists** | | |
|---|---|---|
| list(…, …, …) | Creates a list using any mix of R structures and objects | - Simply supply the elements that are to be included in the list to the *list* function, separated by commas<br>- Can contain numeric matrices, logical arrays, single character strings, factor objects, another list<br>- Elements in the list are printed in the order they were supplied to the function<br>- Functions which can be used: *length* |
| list(name1= …, name2= …, name3= …, …) | Names the components of a list as it's being created through a label assigned to each component | When specifying, names are entered without quotes |
| mylst[[x]] | Member reference - retrieves components from a list using index *x* | - The component retrieved can be treated like a stand-alone object in the workspace<br>- The double square brackets on a list is always interpreted with respect to a single member<br>- For instance, if *x* is a vector like *mylst[[c(2, 3)]]*, R will access the third element of the second component in the list |
| mylst[[x]] <- … | Overwrites a member of the list using the assignment operator | |
| mylst[c(…)] | List slicing (**multiple** members) | The **result** will be a **list** with its components stored in the order they were requested |
| names(mylst) | Gives a vector of the names of the components in a list | Component names are provided and returned as character strings in double quotes |
| names(mylst) <- (“…”.,“…”. …) | Names or renames the list components | Makes elements more recognisable and easy to work with |
| mylst$… | Gets named member/variable by entering the name after the dollar sign | - Gives an identical result when using index subsetting with double square brackets ( [[ ]] )<br>- Names are **not** entered as strings |
| mylst$… <- … | Add and assign a new component to the list using a new name | |
| newlst <- c(oldlst, newvar= …) | Adding a new component to the list with its elements known | |
| mylst$…$… | Nesting – retrieving elements from components of a list | - Naming ( $ ) and indexes [ x ] can be used in combination to retrieve members of the inner list<br>- As long as you're aware of what is returned at each layer of a subset, you can continue to subset as needed using names and numeric indexes |

| **Data frames** | | |
|---|---|---|
| data.frame(*var1*= c(…), *var2*= c(…), *var3*= c(…) …, stringsAsFactors= T/F) | Creates a data frame where | - The members must be vectors (numeric, factor etc) of the equal length<br>- Each row in a data frame is called a *record*, and each column is a *variable*<br>- R's default behaviour for character vectors is to covert each variable into a factor object, setting *stringAsFactors* to *FALSE* prevents this automatic conversion<br>- Functions that can be used: *nrow*, *ncol*, *dim* |
| mydata[x,y] | Extracts data frame row/columns by specifying row and column index positions | - **Negative** indices can be used to omit records and variables<br>- Variable names as strings can also be used |
| mydata**$**… | Access variables in data frame by using the names of vectors that were passed to *data.frame* | |
| mydata <- rbind(mydata, **newdata**)<br>mydata <- cbind(mydata, **newdata**)<br><br>mydata**$newvar** <- … | Adding data columns and combining data frames | - The **first** step is to **create a new data frame** that contains the new information to add<br>- Make sure the variable names and the data types match the data frame you're planning to add this to<br>- *cbind/($)*: set of observations for a new variable (adding to the number of columns)<br>- *rbind*: more records (adding to the number of rows)<br>- To permanently overwrite the data frame, use the assignment operator |
| mydata[logicalvec, y] | Subset data structures using logical flags where *logicalvec* is obtained with the use of relational operators | - Often useful when examining a subset of entries to meet certain criteria<br>- The logical flag vector, *logicalvec*, has to match the number of records in the data frame<br>- Sometimes, asking for a subset yields no records (i.e. R returns a data frame with zero rows) because there are no records that fulfil the criteria<br>- To check whether a subset will contain any records, *nrow* can be used on the result – if this is equal to **zero**, then no records have satisfied the specified condition(s)<br>- *y* is optional and can be a **negative** index to remove a variable, especially if the criteria was based on a variable (e.g. male and females)<br>- *y* can also be a character vector of variable names |
| view(mydata) | Shows data frame in a new tab in the editor | |
| str(mydata, max.level= x) | Displays internal structure of data frame, i.e. total number of observations, number of variables, variable names, data type of each variable, first observations etc. | - Similar to *summary*<br>- *max.level* will give the $x^{th}$ level of information |

| | | |
|---|---|---|
| head(mydata, n= x) | To see the first few/*x* rows of the data frame | |
| tail(mydata) | To see the last few rows | |
| rankedpos <- order(mydata$...)<br>mydata[rankedpos, ] | Sorts data frame base on a variable | |

| | | |
|---|---|---|
| **Special values** | | |
| These special values can be used to mark abnormal or missing values in vectors, arrays, or other data structures | | |
| **Infinity *(Inf)*** | | |
| Inf, -Inf | Value for **±**infinity - a value too large for R to represent will be deemed "*Inf*" | - Special object *Inf* is **case-sensitive**<br>- *Inf* can be associated only with numeric vectors<br>- Although the mathematical concept of infinity (∞) does not correspond to a specific number, R simply has to define an extreme cut-off point before it cannot reliably represent it<br>- Though infinity does not represent any specific value, mathematical operations can be performed on infinite values in R<br>- For example,<br>*Inf * -9 = -Inf*<br>*Inf + 1 = Inf*<br>*Inf – 45.2 = Inf*<br>*Inf + Inf = Inf*<br>*Inf / 23 = Inf*<br>- Any (finite) numeric value divided by infinity, positive or negative, will result in zero, e.g.<br>*59 / 0 = Inf*<br>*Inf / 0 = Inf*<br>- Relational operators work on infinite values as well, e.g.<br>*Inf < Inf is TRUE*<br>*Inf > Inf is FALSE*<br>- More details can be found through *?Inf* |
| is.infinite(x= …) | Element-wise check for *Inf* | - Where argument x is a collection of values, typically a vector<br>- Note that these functions do not distinguish between positive (+) or negative (-) infinity<br>- Also, the result of *is.finite* will always be the opposite (the negation) of the result of *is.infinite* |
| is.finite(x= …) | Element-wise check for finiteness | |
| **Not a Number *(NaN)*** | | |
| NaN | Value for invalid numerics when the result of a calculation is impossible to express using a number or ±*Inf*, i.e. difficult to quantify | - *NaN* values are associated **only** with **numeric** observations<br>- Rarely are *NaN* values defined or included directly<br>- Attempting to cancel representations of infinity in any way will result in *NaN*, e.g.<br>*-Inf + Inf = NaN*<br>*(because ±Inf cannot be interpreted in that numeric sense)*<br>*Inf / Inf = NaN*<br>- When *zero* is divided by zero, the result is *NaN*, i.e. ***0 / 0*** *= NaN*<br>- Any mathematical operation involving *NaN* will simply result in *NaN*<br>- **Relational** operators **cannot** work on *NaN* values *** |

|  |  |  |
|---|---|---|
| is.nan(x= …) | Element-wise check for *NaN* – detect presence of *NaN* values |  |
| **Not Available *(NA)*** |  |  |
| NA | Value for missing observation | - Can exist in both numeric and non-numeric settings<br>- In factor vectors, *NA*s are printed as ***<NA>*** to prevent *NA* from being mistakenly interpreted as one of the levels<br>- Arithmetic calculations with *NA* and using relational operators with either *NaN* or *NA* will result in *NA*<br>- More details on the usage and finer technicalities of *NA* values can be found through *?NA* |
| is.na(x= …) | Element-wise check for *NA* **OR** *NaN* (since numerically, they are similar – there is nothing you can do with either value) | Useful for removing (using negative indices) or replacing *NA* or *NaN* values |
| which(x= is.na(x= …)**&!**is.nan(x= …)) | Identifies *NA* entries only and gives their index positions |  |
| na.omit(object= …) | Delete all *NA*s and *NaN*s (if elements are numeric) | Some additional output will be displayed in printing the returned object, e.g.<br>*attr(, "na.action")*<br>…<br>*attr(, "class")*<br>…<br>which is provided to inform the user that there were elements in the original vector that were removed |
| **Null *(NULL)*** |  |  |
| NULL | Value for "empty" | - Can be used as an useful and flexible tool to facilitate checks on which arguments of a function have been supplied and which are missing or empty (explicitly state or check if a certain object has been defined)<br>- *NULL* cannot take up a position in a vector<br>- In arithmetic or relational operations, *NULL* typically dominates any arithmetic or special values and gives a result of an "empty" vector of a type determined by the nature of the operations attempted<br>- For example,<br>*NULL + 53 = numeric(0)*<br>*53 <= NULL = logical(0)*<br>*NaN – NULL + NA / Inf = numeric(0)*<br>- *NULL* occurs also when examining lists and data frames, i.e. when trying to access a member that doesn't exist (hence, it can be filled with whatever you want)<br>- Same goes when querying a data frame for a non-existent column or variable using the ($) operator<br>- Help file: *?NULL* |

| | | |
|---|---|---|
| is.null(x= …) | Check for *NULL* (gives a single answer, i.e. *T/F* - whether an object is empty or supplied) | |

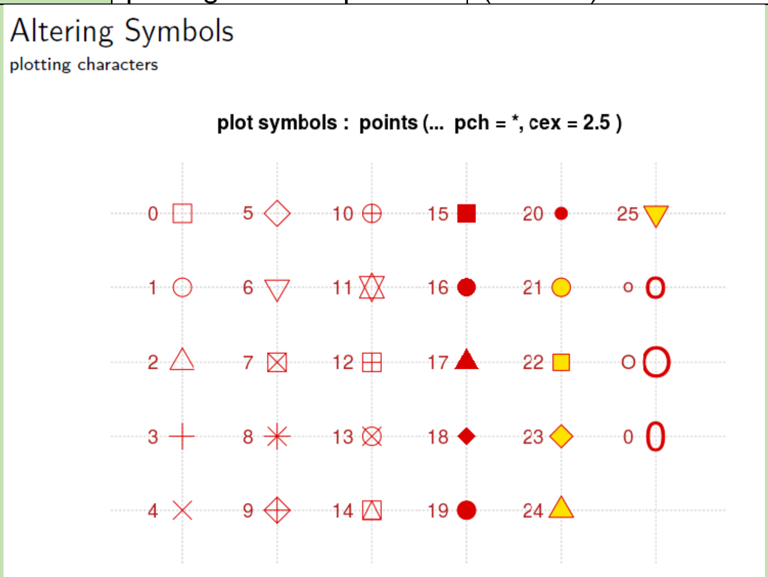
<table>
<tr><th colspan="4">*Date objects*</th></tr>
<tr><td>Applications</td><td colspan="2">- Generate hourly log files on remote servers<br>- Study seasonality changes<br>- Time-series analysis</td><td></td></tr>
<tr><td>*Date* class</td><td colspan="2">Allows R to compute the difference between dates, sequences of dates and divide dates into convenient periods</td><td>Dates are stored internally in R as integers (number of days since 1st Jan 1970)</td></tr>
<tr><td>unclass(my_date)</td><td colspan="2">Gives the number of days since January 1, 1970</td><td></td></tr>
<tr><td>Sys.Date()</td><td colspan="2">Gives the current date</td><td>E.g. *“2015-05-07”*</td></tr>
<tr><td>as.Date(“…”)<br>as.Date(“…”, format= “…”)</td><td colspan="2">Creates Date objects</td><td>- The default way to create is a string of format “%YYYY-%mm-%dd”<br>- To use date of other presentations, indicate to R through the *format* argument<br>- *“11/22/1997”* would use format *“%m/%m/%Y”*<br>- Note that all arguments are enclosed in quotes ( “” )</td></tr>
<tr><td>Conversion symbols</td><td>%Y</td><td>4-digit year</td><td>E.g. “1982”</td></tr>
<tr><td></td><td>%y</td><td>2-digit year</td><td>E.g. “82”</td></tr>
<tr><td></td><td>%m</td><td>2-digit month</td><td>E.g. “01”</td></tr>
<tr><td></td><td>%d</td><td>2-digit day of month</td><td>E.g. “13”</td></tr>
<tr><td></td><td>%A</td><td>Weekday</td><td>E.g. “Wednesday”</td></tr>
<tr><td></td><td>%a</td><td>Abbreviated weekday</td><td>E.g. “Wed”</td></tr>
<tr><td></td><td>%u</td><td>Weekday in numbers</td><td>E.g. “3” for Wednesday</td></tr>
<tr><td></td><td>%B</td><td>Month</td><td>E.g. “January”</td></tr>
<tr><td></td><td>%b</td><td>Abbreviated month</td><td>E.g. “Jan”</td></tr>
<tr><td>format(my_date, format= …)</td><td colspan="2">Converts date to character strings</td><td>E.g. format(Sys.Date(), format= “Today is a %A!”</td></tr>
<tr><td>weekdays(mydate, abbreviate= T/F)<br>months(mydate, abbreviate= T/f)</td><td colspan="2">Extracts out required information, i.e. weekday and month of a date</td><td>*abbreviate = T* will provide a short form of the weekday or the month name</td></tr>
<tr><td>seq(mydate – x, mydate, by= “…”)</td><td colspan="2">Creates a sequence of dates starting *x* days ago</td><td>The argument for *by* can be *“1 week”, “month”, “years”, “quarters”* etc. which specify the length of duration the days are set apart</td></tr>
<tr><td>cut(mydates, breaks= “…”, labels= T/F)</td><td colspan="2">Divides a sequence of data into groups by *month*, *week*, or *quarter*</td><td></td></tr>
<tr><td>my_date + x</td><td colspan="2">Increments *my_date* by *x* days</td><td></td></tr>
<tr><td>my_date2 - my_date</td><td colspan="2">Calculates the **time difference** between 2 dates in **days**</td><td></td></tr>
<tr><th colspan="4">Time objects (*POSIXct*)</th></tr>
<tr><td>unclass(my_time)</td><td colspan="2">Gives the number of seconds since January 1, 1970</td><td></td></tr>
<tr><td>Sys.time()</td><td colspan="2">Gives the current date and time in a vector</td><td>E.g. *“2015-05-07” “10:34:52 CEST”* is of class *“POSIXct”* and *“POSIXt”* respectively</td></tr>
<tr><td>as.POSIXct(…)<br>as.POSIXct(…, format= …)</td><td colspan="2">Creates a time object<br>Converts a character vector to a *POSIXct* object</td><td>Default match is *“%Y-%m-%d” “%H:%M:%S”*</td></tr>
</table>

| | | | |
|---|---|---|---|
| Conversion symbols | %H | Hours as a decimal number (00-23) | Consult *?strptime* for full list of conversion symbols |
| | %I | Hours as a decimal number (01-12) | |
| | %M | Minutes as decimal number | |
| | %S | Seconds as a decimal number | |
| | %T | Shorthand notation for the typical format *“%H:%M:%S”* | |
| | %p | AM/PM indicator | |
| my_time + x | Increments *my_time* by *x* **seconds** | | |
| my_time - my_time2 | Time difference between 2 timings in seconds | | Note that it gives the result in days if the difference is too large |
| R packages useful for *Date* and *POSIXct* objects | - *lubridate*<br>- *zoo*<br>- *xts* | | |

| **Attributes** | | | |
|---|---|---|---|
| Additional information about the nature of the object created in R | | *Explicit* | *Implicit* |
| | | Immediately visible to the user, e.g. *levels* in a factor vector | Determined by R internally, e.g. *dimensions* of a array |
| attributes(foo) | Lists explicit attributes | | |
| attr(x= …, which= "…") | Obtain specific attribute | Where *x* is the object and the attribute for it is *which*, e.g. *"dim"* | |
| **Object Class** | | | |
| Every object created in R is identified, implicitly or explicitly, with at least one class. R – an *object-oriented* language – stores entities as objects and have methods that act upon them. In such a language, class identification is formally referred to as *inheritance* | | The most common classing structure in R is called *S3*. Another structure, *S4*, is essentially a more formal set of rules for the identification and treatment of different objects | |
| class(…) | Gets class of object (S3) – nature of the data *structure* | Example of classes:<br>- "integer", e.g. *1 2 3 4*<br>- "numeric", e.g. *1.0 1.6 2.2 2.8 (floating-point numbers)*<br>- "character", e.g. *"a" "string"*<br>- "logical", e.g. *TRUE FALSE*<br>- "factor"<br>- "matrix" | |
| Multiple classes | Certain objects will have multiple classes – a variant on a standard form of an object | For instance, an ordered vector will have the classes: "ordered" and "factor" | |
| typeof(…) | Reports the type of data contained in an object | - For vectors, matrices and arrays<br>- Output may not match that of *class* function<br>- See *?typeof* for more details on the values it returns | |
| is._(…) | (is-dot) Object-checking functions where ( _ ) is a specific class or data type and it will return a T/F logical value | - For example:<br>*is.integer(…)*<br>*is.numeric(…)*<br>*is.matrix(…)*<br>*is.data.frame(…)*<br>*is.vector(…)*<br>*is.logical(…)*<br>- Note that these checks use more general categories than the formal classes identified with class, i.e. a data frame is intuitively generalised to a list | |
| as._(…) | (as-dot) Object-coercion functions | - Used when coercion won't happen automatically and must be carried out by the user<br>- For example:<br>*as.numeric(…)*<br>*as.character(…)*<br>*as.matrix(…)*<br>*as.data.frame(…)*<br>*as.vector(…)*<br>*as.logical(…)*<br>- Important to note when **coercing** a **factor** (especially those with **numeric levels**) to a numeric data type since R assigns the numeric representation of the factor in the | |

|  |  |  |  |
|---|---|---|---|
|  |  |  | stored order of the factor labels (alphabetical by default):<br>*as.**numeric**(myfac)*<br>- Useful when storing the contents of a matrix/higher-dimensional arrays as a single vector:<br>*as.vector(foo)*<br>$\begin{matrix}1 & 3\\ 2 & 4\end{matrix}$ becomes *1 2 3 4*<br>- *as.matrix(…)* on an array converts it into a matrix with a single column<br>- Note that when coercing a list to a data frame, ensure that the variables (named members) have matching lengths<br>- The variables will be stored in a column-wise fashion, in the order that the list supplies them as members<br>- Help file: *?as* |
| **Utilities** |  |  |  |
| Mathematical functions | mean(…) |  |  |
|  | sum(…) |  |  |
|  | round(…) | Round decimals to integers |  |
|  | abs(…) |  |  |
| Lists/ Vectors | seq(from= …, to= …, by= …) | Creates a sequence |  |
|  | rep(…, times= …, each= …) | Repeats a sequence |  |
|  | sort(…, decreasing= T/F) | Sorts a sequence |  |
|  | append(…, …, after= <index>) | Append values to a sequence/ Merge vectors or lists |  |
|  | rev(…) | Reverses a sequence |  |
|  | unique(…) |  |  |
| str(…) | Compactly display the internal structure of an R object |  |  |
| unlist(…) | Flattens (possibly embedded) lists to produce a vector |  |  |
| is.*(…) | Checks for the class of an R object and whether it is that of class ( * ), returns *TRUE* or *FALSE* |  | For example,<br>*is.vector(…)*<br>*is.lists(…)*<br>*is.numeric(…)* |
| as.*(…) | Converts an R object from one class to another |  | Note that when it is used on a list, it doesn't change the class of the list but rather only changes **each component** of the list |

<table>
<tr><th colspan="5"><b>Base/Traditional R graphics <i>(plot)</i></b></th></tr>
<tr><td>plot(foo, bar)<br>plot(baz)<br>plot(…, [graphical params])</td><td colspan="2">Creates/displays a base R plot where <i>foo</i> is a vector of x coordinates and <i>bar</i> is that for the y coordinates <b>OR</b> <i>baz</i> is a matrix (with the x-values in the 1st column and y-values in the 2nd column) or as a list or a data frame</td><td colspan="2">Graphical parameters can be supplied as optional arguments to the <i>plot</i> function which can invoke simple visual enhancements</td></tr>
<tr><td>barplot(…, border= NA, names.arg= str_to_title(…))</td><td colspan="2">Creates a bar-chart by drawing bars who heights are proportional to the values of a variable being represented</td><td colspan="2">- First argument can be a vector or matrix of values which can be derived from a data frame using the $ operator<br>- <i>names.arg</i> is a vector of names to be plotted below each bar or group of bars<br>- The argument for <i>border</i> sets the colour to be used for the border of the bars<br>- <i>border= NA</i> omits borders and <i>border= T</i> uses the same colour of shading lines for the border</td></tr>
<tr><td rowspan="13">Graphical parameters</td><td>Parameters</td><td></td><td colspan="2">Examples</td></tr>
<tr><td rowspan="6"><b><i>type</i></b></td><td rowspan="6">Sets the plot type – tells R how to plot the supplied coordinates</td><td colspan="2">Stand-alone points, joined by lines or both dots and lines</td></tr>
<tr><td><i>"p"</i></td><td>Default – "points only"</td></tr>
<tr><td><i>"l"</i></td><td>"lines only"</td></tr>
<tr><td><i>"b"</i></td><td>Both points and lines</td></tr>
<tr><td><i>"o"</i></td><td>Overplotting the points with lines (eliminates the gaps between points and lines visible for <i>type= "b"</i>)</td></tr>
<tr><td><i>"n"</i></td><td>No points or lines plotted (useful for complicated plots which must be constructed in steps)</td></tr>
<tr><td><b><i>main, xlab, ylab</i></b></td><td>Sets axis labels – plot title, horizontal axis label, vertical axis label respectively</td><td colspan="2">- Supply text as character strings<br>- To omit any one, set an empty string ( "" ) as the argument<br>- Note that these strings may include escape sequences<br>- By default, a basic plot won't have a main title, and its axes will be labelled with the names of the vectors being plotted</td></tr>
<tr><td rowspan="5"><b><i>col</i></b></td><td rowspan="5">Sets point/line colour – colour(s) to use for plotting points and lines</td><td colspan="2">- Enter <i>colours()</i> at prompt for colour string values recognised by R<br>- 8 possible integer values, around 650 character strings to specify colour<br>- Personal palettes can be created by specifying colours using RGB levels</td></tr>
<tr><td><i>1</i></td><td>Default – <i>"black"</i></td></tr>
<tr><td><i>2</i></td><td><i>"red"</i></td></tr>
<tr><td><i>3</i></td><td><i>"green"</i></td></tr>
<tr><td><i>4</i></td><td>Dark blue</td></tr>
</table>

|  |  |  |  |  |
|---|---|---|---|---|
|  |  |  | *5* | Light blue |
|  |  |  | *6* | *"pink"* |
|  |  |  | *7* | *"yellow"* |
|  |  |  | *8* | *"grey"* |
|  | *rgb*<br>***(col= rgb(red, blue, green, aplpha= …))*** | Create personal palettes by specifying colours using RGB levels | - Arguments for *red, blue and green* are integers<br>- An *alpha* transparency value can also be specified (as an opacity, so ***0*** means **fully transparent** and max means opaque)<br>- If *alpha* is not specified, by default, an opaque colour is generated<br>- For example, *new_colour <- rgb(x, y, z, alpha= 0.4)* gives a semi-transparent colour |  |
|  | ***pch*** | *Point character*, sets point type – selects which character to use for plotting individual points | You can specify a character to use for each point, or specify a value between 1 and 25 (inclusive) |  |
|  |  |  Altering Symbols<br>plotting characters |  |  |
|  | ***cex*** | *Character expansion*, sets point size – controls the size of plotted point characters | - Default size is 1 – to half is 0.5 and to double is 2<br>- ***cex.axis*** afftects axis font size<br>- ***cex.main*** affetcs title font size |  |
|  | ***lty*** | *Line type* – specifies the type of line to use to connect the points | *1* | Solid |
|  |  |  | *2* | Dashed |
|  |  |  | *3* | Small-dashed/ Dotted |
|  |  |  | *4* | Dash-dot-dashed |
|  |  |  | *5* | Wide-dashed |
|  |  |  | *6* | More compressed dash-dotted |
|  | ***lwd*** | *Line width* – controls the thickness of plotted lines |  |  |
|  | ***xlim, ylim*** | Sets plot region limits (plotting region) – limits for the horizontal range and vertical range respectively | - Both parameters require a numeric vector of length 2, provided as *c(lower, upper)*<br>- By default, R sets a range of each axis by using the range of supplied *x* and *y* values (plus a |  |

|  |  |  |  |
|---|---|---|---|
|  |  |  | small constant to pad a little area around the outermost points) |
|  | ***bty*** | *Box-type* – Determines the type of box which is drawn around plots | Argument is character string where the resulting box resembles the corresponding upper case letter<br>“o” Default □<br>“l” └<br>“7” ┐<br>“c” ⊏<br>“u” ⊔<br>“]” ⊐<br>“n” Suppresses the box |
| Add to a plot without refreshing or clearing a window<br><br>fn(…, [graphical params]) | Functions |  |  |
|  | *points*(**x**= *c(…),*<br>**y**= *c(…), …)* | Add points | Similar to *plot*, use two vectors of equal lengths |
|  | *abline*(***h/v***= ***c****(…), …)*<br>*abline(b= …, a= …)* | Add horizontal/vertical lines<br>Add a line (***bx +a***) with a gradient | - Remember that *h= c(…)* for horizontal lines and *v= c(…)* for vertical lines<br>- Argument is a vector of x or y-intercepts of the lines required |
|  | *segments*(***x0***= *c(…),*<br>***y0***= *c(…),*<br>***x1***= *c(…),*<br>***y1***= *c(…), …)* | Add specific line segments | - These lines do **not** span the entire plotting region like *ablines*<br>- The function takes command from **“from”** coordinate(s) (*x0* and *y0*) and **“to”** coordinate(s) (*x1* and *y1*), and draws the corresponding line |
|  | *lines*(***x***, ***y***, *lty= …, …)* | Add lines connecting coordinates | Where *x* and *y* are vectors containing the x and y coordinates of the points you want connected |
|  | *arrows*(***x0***= *…,*<br>***y0***= *…,*<br>***x1***= *…,*<br>***y1***= *…)* | Add arrows | - This function is used just like for *segments*<br>- By default, the head of the arrow is located at the “to” coordinate<br>- Though this (and other options such as angle and length of the head) can be altered using optional arguments described in *?arrows* |
|  | *text*(***x***= *…,* ***y***= *…,*<br>***labels***= *“…”)* | Add texts | The default behaviour is to **center** the **string** supplied as *labels* on the coordinates provided with the arguments *x* and *y* |
|  | *legend*(**[position of legend]**,<br>***legend***= *c(…),*<br>*pch= c(…),*<br>*lty= c(…),*<br>*col= c(…),*<br>*lwd= c(…),*<br>***pt.cex***= *c(…))* | Add/control legend | - The first argument sets where the legend should be placed<br>- Setting the exact x- and y-coordinates is possible but picking a corner using a character string suffices:<br>***“topleft”***<br>***“topright”***<br>***“bottomleft”***<br>***“bottomright”***<br>- Argument for *legend* is the labels as a vector of character strings<br>- Then supply the remaining argument values in the vectors of the **same length** so that the right elements match up with each label, i.e. index |

|  |  |  |
|---|---|---|
|  |  | position of the label corresponds to index position in remaining arguments<br>- For example,<br>if *labels* is *c("too small", …)* and you want a line of type 4 with default thickness and colour for "too small", you write:<br>*pch= c(NA, …)*<br>*lty= c(4, …)*<br>*col= c("black", …)*<br>*lwd= c(1, …)*<br>*pt.cex= c(NA, …)*<br>- Note that you have to fill in some elements in these vectors as *NA* when you don't want to set the corresponding graphical parameter<br>- In addition, *pt.cex* simply refers to *cex* parameter when calling *points* (using just *cex* in legend would expand the text used, not the points) |
| duplicated(…) | Shows which points are the same, i.e. overlapping each other through opacity |  |
| **Grammar of graphics package *(ggplot2)*** |  |  |
| qplot(foo, bar)<br>qplot(foo, bar, main= "…", xlab= "…", ylab= "…")<br>qplot(foo, bar, geom= "blank") | Creates a *ggplot2* "quick plot" | *ggplot2* plots are stored as **objects**, which means they have an underlying, static representation until you *change* the object – what you essentially visualise with *qplot* is the *printed* object at any given time |
| qplot(foo, bar, colour= ptype, shape= ptype) | By splitting a data set into categories using a factor object, ggplot 2 can automatically apply particular styles to different categories | For instance, to generate *ptype*:<br>*ptype <- rep(NA, length(x= foo)*<br>*ptype[*<condition for y-coords>*]* <- "<classification label>"<br>…<br>*ptype[*<condition for x & y-coords>*]* <- "…"<br>…<br>*ptype <- factor(x= ptype)*<br>will give a factor with values sorted into different levels |
| + geom_point(**size**= …, **shape**= …, colour= …)<br><br>+ geom_line(colour= …, linetype= …)<br>+ geom_line(**mapping**= aes(group= …), …) | Geometric modifiers – adds points and lines geom respectively to the *qplot* object | - For default settings, simply write<br>*qplot(foo, bar, geom= "blank")*<br>- *size* is the same as *cex*, *shape* is the same as *pch*<br>- For other geometric modifiers, prompt R with *??geom_*<br>- By default, mapping to ptype will result in line connecting points only of the same categories/levels<br>- To draw lines connecting all the points, from left to right, use:<br>*…aes(group= 1), …*<br>to treat all the observations as one group |
| + geom_hline(mapping= aes(x/yintercept= c(…), …) | Adds horizontal lines geom | The intercept will pass through the points given in the vector argument |
| + geom_segment(mapping= aes(x= …, y= …, xend= …, yend= …), …) | Adds line segments geom | - Arguments *x* and *y*: "from" coordinate<br>- Arguments *xend* and *yend*: "to" coordinate |

| ***ggplot2*** | | |
|---|---|---|
| Grammar of graphics | - By Leland Wilkinson<br>-Abstract method of building up graphics from components<br>- 2 principles:<br>1. Graphics – distinct layers of grammatical elements<br>2. Meaning plots through aesthetic mapping<br>- Types of plots:<br>1. Exploratory plots<br>(Specialist, data-heavy)<br>2. Explanatory plots<br>(General audience, specific data) | |
| Essential Grammatical Elements (Layers) | Element | Description |
| | Data | Dataset being plotted |
| | Aesthetic | Scale onto which we map our data |
| | Geometrics | Visual elements used for our data<br>(Actual shape plot will take) |
| | Facets | Plotting small multiples |
| | Statistics | Representations of our data to aid understanding |
| | Coordinates | The space on which the data will be plotted |
| | Themes | All non-data ink |
| *library(**tidyverse**)* | *ggplot2* is part of the *tidyverse* collection of packages | |
| ***ggplot**()*<br><br>*ggplot(**data**= df, **aes**(x =…, y = …, …))*<br><br>*ggplot(data = <DATA>) + <GEOM_FN>(mapping = aes(<MAPPINGS>))* | | - By default, *ggplot* treats variables to *x* and *y* as continuous variables.<br>- To treat it as a categorical variable, use *factor(var1)*<br>- It is possible to store *ggplot(…)* as an object and use it later to add other layers<br>- Alternative to base R and lattice package |
| **Variables of interest (aesthetics)**<br>(Refers to anything on the graph that can vary according to a variable)<br>(Able to represent another variable by colour, size etc. besides the x and y-axis) | | |
| - Able to represent another variable by colour, size etc. besides the x and y-axis<br>- Scaling: each unique value of the variable is assigned a unique level<br>- A legend that explains what levels of the aesthetic correspond to which values of the data | | |
| *aes(…)* | *<MAPPINGS>* | |
| | *x* | x axis position |
| | *y* | y axis position |
| | *color* | - Colour of dots/outlines of other shapes<br>- Maps variable onto a colour gradient/continuous colour scale<br>- Input can be numbers<br>- In addition, *color = (var > x)* or any other **conditional statement** makes a continuous variable discrete |
| | *size* | Diameter of points/thickness of lines |
| | *fill* | Fill colour |
| | *alpha* | Transparency (0 – 1) |
| | (Applicable to discrete/categorical variables only) | |
| | *group* | |
| | *shape* | - Shape of a point<br>- Note: Finite number of shapes which *ggplot()* can automatically assign to points |
| | *linetype* | Line dash pattern |
| | *labels* | Text on a plot or axes |
| | | |
| | | |

| **Geometric objects/layers (Geom/ <GEOM_FN>)** | | | | |
|---|---|---|---|---|
| **+ geom_…** | *mapping = aes(…)* | – *aes* can be used as an argument in an geom object – helps controls *aes* mappings of each layer independently<br>- Mapping argument defines how variables in dataset are mapped to visual properties | | |
| | *_point()* | Scatterplots - draws points on the plot | | |
| | | *x, y* | (Essential) | |
| | | *(Optional)* | | |
| | | *alpha* | | |
| | | *shape* | *= 1-20* | Only accepts a *color* aes |
| | | | *= 21-25* | Accepts both *color* and *fill* aes |
| | | | *= 19* | - Default<br>- Solid, same outline colour |
| | | | *= 1* | Hollow |
| | | | *= 16* | Solid, no outline |
| | | | *= 21* | If mapped in *aes: fill* (inside), *color* (outline) |
| | | *position* | string or posn.j | Where posn.j is a position adjustment function |
| | | *color, fill, size (in mm)* | | |
| | | To set a property of a geom to a single value, pass it as an argument (e.g. transparency of points will be set to a fixed value) | | |
| | *_smooth()* | - Draws a smoothed line through points<br>- Allows patterns in the data to be seen and depict:<br>1) Trends in time series data<br>2) (Non-linear) relationships between variables | | |
| | | *se* | *= F* | (option to not have error shading) |
| | | *span* | | Default = *0.75* (uses nearest 75 points from x*) |
| | | *method* | *= "lm"* | (simple) Linear regression model<br>- Blue line: line of best fit<br>- Gray regions: 95% CI for mean |
| | | | *= "glm'* | Logistic regression – binary data |
| | | | *= "gam"* | (uses splices) |
| | | | *= "loess"* | Default – loess smoother<br>(locally weighted regression smoother) |
| | | | *= "rlm"* | |
| | | *line_type* | | |
| | | *show.legend* | *= T/F* | |
| | | *x, y, alpha, color, fill, linetype* | | |
| | | If *aes(group = 1)*:<br>tells R to draw a single linear model through all the points<br>By default, R draws a line per subset of the data frame | | |
| | Bar charts | | | |
| | *_bar()* | Makes height of bars proportional to the number of cases in each group (computes values) | | |
| | | *aes(y= …)* | *..prop..* | Computes proportion within each group |
| | | *aes(group=…)* | int | E.g. *= 1*<br>Controls how the data is split up before the statistics in *y = …* are computed |
| | | *position* | *= "stack"* | Default |
| | | | *= "fill"* | Shows relative proportions |
| | | | *= "dodge"* | Principles of similarity and proximity |
| | *_col()* | Makes height of bars represent values in data | | |
| | | | | |
| | Histograms | - Visualise distribution of a continuous variable<br>- *geom_histogram()* and *geom_freqpoly()* | | |

| | | | | |
|---|---|---|---|---|
| | *_histogram()* | - Cuts up continuous variables into discrete bins<br>- Display counts in each bin with bars | | |
| | | *aes(y = …)* | *..density..* | Proportional frequency of bin in relation to whole dataset |
| | | | *..count..* | Counts how many values there are in each bin |
| | | *binwidth* | *= 0.1 (e.g)* | |
| | | *position* | Specifies how to draw bars of the plot | |
| | | | *= “stack”* | - (Default)<br>- Place bars on top of each other |
| | | | *= “dodge”* | Place bars next to each other (uses counts) |
| | | | *= “fill”* | Place bars on top of each other (uses proportion) |
| | | | *= “identity”* | |
| | | *stat* | *= “bin”* | Default: 30 evenly-sized bins |
| | | | *= “identity”* | |
| | *_freqpoly()* | - Displays counts with lines<br>- More appropriate than *geom_histogram()* if comparing the distribution of a variable conditioned on a categorical one (overlaying multiple histograms) | | |
| | | *x* | | |
| | | *alpha, color, fill* | | |
| | Time series | | | |
| | *_line()* | Line charts – connects observations in the order of the variable on the x-axis | | |
| | | x, y | | |
| | | *linetype, size, alpha, color, group* | | |
| | *_boxplot()* | Boxplots<br>- Visual representation of 3 of the 5 numbers in the 5-number summary<br>*(men/median, upper quartile, lower quartile + min, max)*<br>- Identify which points could be outliers (observations that are very different from the majority of the data – falls > (1.5 times IQR) below LQ or above UQ) | | |
| | | *aes()* | *x, lower, upper, middle, ymin, ymax* | |
| | | *color, fill, linetype* | | |
| | *_jitter()* | | | |
| | *_text()* | Adds text directly to the plot | | |
| | | *x, y* | | |
| | | *label* | | Used in *aes* mapping |
| | | *hjust, vjust* | int | Number between 0 (right/bottom) and 1 (left/top) |
| | | | *= “left”/”middle”/”right”/”bottom”/”center”/”top”* | |
| | | | *= “inward”* | Aligns text towards the center |
| | | | *= “outward”* | Aligns text away from the center |
| | | *nudge_x* | decimal | E.g. *0.1* |
| | | *size* | decimal | E.g. *3.5* |
| | Reference lines | | | |
| | *_vline()* | Adds vertical lines | | |
| | | *xintercept* | | |
| | | *lty, size* | | |
| | *_hline()* | Adds horizontal lines | | |
| | | *yintercept* | | |
| | *_abline()* | Adds straight lines defined by a slope and an intercept *(y = ax + b)* | | |
| | | *slope* | | |
| | | *intercept* | | |

| | | | | |
|---|---|---|---|---|
| | *_rug()* | A compact visualisation designed to supplement a 2D display with marginal distributions<br>(draws a ‘tickmarks’ (x,y) for every point)<br>- Good for when points are close to each other (indicates where the points are concentrated) | | |
| | | *position* | | |
| | | *alpha, color, group, linetype, size* | | |
| | *_label()* | | | |

| **Common geom attributes/arguments** | | | |
|---|---|---|---|
| geom_…(…) | *data* | = … | - Allows a different data set to be specified to overlay on top of a previous plot<br>- *geom* inherits *aes* from *ggplot aes* |
| | *alpha* | = *x*<br>(where 0 ≤ *x* ≤ 1) | - Makes points 100*x*% transparent<br>- *alpha* = *1* indicates opaque |
| | *position* | Specifies how *ggplot* will adjust for overlapping bar/points | |
| | | = *“identity”* | - (Default)<br>- Value in data frame exactly where value will be in plot |
| | | = *“dodge”* | |
| | | = *“stack”* | |
| | | = *“fill”* | |
| | | = *“jitter”* | Adds random noise on both *x* and *y* axes to see regions of high density |
| | | = *“jitterdodge”* | |
| | *show.legend* | = *T/F* | |

| **Scale functions (modifying scale)** | | | |
|---|---|---|---|
| *scale_x_…* | + *_continuous()*<br>+ *_discrete()*<br>+ *_manual()* | 1st argument: name of scaled variable | |
| *scale_y_…* | | Addition arguments: | |
| *scale_colour_…* | | *limits = c(x, y)* | Describes scale limits |
| *scale_fill_…* | | *breaks = seq(…)* | Control breaks in the guide |
| *scale_colour_…* | | *expand = c(a, b)* | - Expands range of scales<br>- Helps give space between points and grid<br>- *a*: by multiple of<br>- *b*: with addition of |
| *scale_shape_…* | | *labels = c(”…’”, …)* | Adjusts category names (in legend) |
| *scale_linetype_…* | | *name = “…”* | Adjusts name of legend |
| | | *values = x : y* | |
| *scale_colour_brewer()* | *palette=* | string | - Use *display.brewer.all()* |

| **Other functions** | | | | |
|---|---|---|---|---|
| | Argument | Type | | |
| *position.jitter()* | *width=*<br>*height=* | decimal | - Position adjustment function<br>- Sets a specific argument for position<br>- Helps maintain consistency in jitter across plots<br>- E.g (*width = 0.1*) | |
| *labs()* | *x, y* | string | | |
| | *title* | | Left-justified to make it seem like a paragraph (seams into words/text) | |
| | *subtitle* | | | |
| | *label* | | | |
| *xlab(), ylab()* | | string | | |
| Facets | - Using a facet splits your data into subsets according to levels of the factor, and one sub-plot is created for each level of the factor<br>- The faceting variable is usually a categorical one<br>- Enables creation of small multiples | | | |
| *facet_grid()* | facets | *. ~ var*<br>*var1 ~ var2* | - Lays out panels in a grid<br>- Most useful with 2 discrete variables, and all combinations of the variables exist in the data<br>- ( . ) used to indicate there should be no faceting on this dimention (row ~ col) | |
| *facet_wrap()* | facet | *~ var* | - Facets by a single variable | |
| | *nrow* | int | | |
| | *labeller* | | E.g.<br>*as_labeller(c(`female` = "Female", `male` = "Male"))* | |
| *rainbow()* | | int | - E.g. *rainbow(7)* | |
| *xlim()* | 2 arguments | lowest and highest limits | - Extend limits of the graph | |
| *coord_flip()* | | | - Useful in bar charts when categories have long labels<br>- Note: Axis **does not** change internally | |
| *coord_cartesian()* | *xlim* | *c(x,y)* | - Zoom in on a particular section of the histogram<br>- Similar to using a magnifying glass to focus on a particular section of the graph | |
| Themes | - Controls all the non-data ink part of the plot including visual elements not part of the data | | | |
| *theme()* | *legend.position* | *"bottom" etc.* | | |
| | *legend.direction* | | | |
| | *plot.background* | *= element_rect()* | *fill = …* | |
| | | | *colour = …* | Adds a **border** to the background |
| | | | *size = …* | |
| | | | etc. | |
| | *axis.ticks* | *= element_line()* | *colour = …* | Change **tick marks** to a specified colour |
| | *axis.line* | | *colour = …* | Adds **axis lines** of specified colour |
| | *panel.grid* | | *= element_blank()* | Removes **grid lines** |
| | *strip.text* | *= element_text()* | *size = …* | Change appearance of **strip text** (i.e. text in facet strips) |
| | | | *colour = …* | |
| | *axis.title.x, axis.title.y* | | *colour = …* | Change axis title |
| | | | *hjust = 0 (puts text in bottom left corner)* | |
| | | | *face = "italic"/"bold"…* | |

| | | | | |
|---|---|---|---|---|
| | *axis.text* | | *colour* = … | Change colour of text at axis |
| | Applicable to all | = *element_blank()* | | |
| *theme_bw()* | | | White background with grid lines | |
| *theme_light()* | | | Light axes and grid lines | |
| *theme_classic()* | | | Classic theme, axes but no grid lines | |
| *theme_linedraw()* | | | Only black lines | |
| *theme_dark()* | | | Dark background for contrast | |
| *theme_minimal()* | | | Minimal theme, no background | |
| *theme_gray()* | | | Grey background (default) | |
| *theme_void()* | | | Empty theme, only geoms visible | |

| **Limitations of base plots** | |
|---|---|
| Base plot | ggplot |
| 1) Plot does not get redrawn<br>(Axis does not change to adjust for new points added) | Plotting square is adjusted |
| 2) Plot drawn as an image (not an object) | *ggplot2* produces an object |
| 3) Need to manually add legend | |
| 4) No unified framework for plotting<br>(E.g. *hist*, *plot* etc) | |

(Add to original main Base R plot table)

| **Base plots** | | |
|---|---|---|
| Lines must be run together in the console for plot of multiple objects to be generated correctly | | |
| *x <- lm(var1 ~ var2, data = df)* | Calculates a linear model of var1 described by var2 | |
| *abline(x, lty= …)* | Adds line to plot already generated | |
| | | |
| | | |

| **Tips** | | |
|---|---|---|
| Things to look out for when plotting: | | |
| Nice grid spacing of points | Be suspicious of overlapping of points | Check unique values against number of observations |
| Plotting histograms | Consider:<br>1) Widths of bins used<br>2) Number of bins<br>3) Location of bins | *ggplot* has a default of 30 bins |
| Zooming in on a histogram | May easy to use *xlim()* instead but *ggplot* will display a conditional distribution which will affect and change the plot (to make area under bars to sum up to one) | Use *coord_cartesian()* |
| Plotting time series | Plots are preferred to be wide than narrow | Try to do away with the legend and consider labelling lines instead |
| Plotting bar charts | Consider:<br>1) Arrange bars in order of tallest to shortest/ vice versa instead of alphabetical order (default)<br>Questions to ask:<br>1) What do we observe from the data?<br>2) How else could we have chosen to represent the data?<br>3) Which is the correct choice?<br>4) What other geoms could we have used for this set of variables?<br>5) Is it an improvement over this? | Reorder levels of the factor:<br>*reorder(df, var)*<br>*OR*<br>*reorder(df, -var)*<br>(** negative ( - ) sign for descending values)<br>*OR*<br>*reorder(df, var, FUN=median)*<br>(calculates and reorders after applying the function to the variable) |
| Plotting boxplots | The bad:<br>- Does not potray certain features of a distribution<br>- For example, distinct mounds and possible gaps in the data | Consider:<br><br>Ordering a series of boxplots by e.g. their median (maximum to smallest) |

| | | |
|---|---|---|
| | The good:<br>- Gives an indication about the skew of a distribution if it is indded unimodal<br>- Useful for identifying potential outliers<br>- And good for comparing groups with respect to their “center” and “spread”<br>(i.e. When comparing the same variable between populations, create a series of boxplots, side by side | |
| Using facets | Consider:<br>1) The variable to facet by choosing the one that answers the question (what will each facet represent) | |
| Deciding *ggplot2* themes | Reasons for a (default) light gray background for plots:<br>1) White grid lines are visible yet easy to tune out<br>(keeps the data prominent)<br>2) Gives a similar colour to typographic text<br>(prevents it from jumping out)<br>3) Creates a continuous field of colour which ensures that the plot is perceived as a single entity | |

| **Troubleshooting problems** | | |
|---|---|---|
| Problem | Solution(s) | |
| Overplotting of points | 1) Jitter the points | |
| Too many categories for variable | Reason: Default scale that *ggplot2* uses only has 6 entries<br>1) Create more manually - edit scale that maps variables to shapes | + *scale_shape_manual(values = x : y)* such that *y* - *x* is the number of shapes needed |
| Histogram: Left-most rectangle centred at 0 even though there are only positive values | Desired outcome: Lower limit of the left-most bin to be 0 (i.e. start exactly at 0) | *geom_histogram():*<br><br>*…binwidth = …,*<br>*boundary = 0…* |
| Histogram: Distracting borders on bars | | *geom_histogram():*<br><br>*…colour=* |

| | | |
|---|---|---|
| **Writing functions** | | |
| Things to decide first | 1. Arguments it should take<br>2. Whether these arguments have defaults, and if so, what they are<br>3. What the function should return | |
| Things to take note | 1. R passes arguments by value, i.e. R function will not change the variable that you input to the function (unless it is re-assigned explicitly outside the function call)<br>2. When writing a function with if and else-statements, write the least probable event last (more efficient)<br>3. Last statement evaluated in a function becomes a return value if it isn't a assignment operation | |
| single_game <- function() { …} | Takes **0** arguments | |
| single_game <- function(n_dice= …) | Makes an argument **optional** and sets **default** value for it | |
| function(x) {…} | Anonymous function | - Note that **small *x*** is used, which can refer to each row or column in the matrix<br>- Write the function *("…")* behind *"function(x)"* |
| Statements | | |
| **if** statement | if (condition) {<br>    expr<br>} else {<br>    expr2<br>}<br><br>OR<br>if (condition) {<br>    expr<br>} else if {<br>    expr2<br>} else {<br>    …<br>} | The *else if* and *else* block is optional |
| **while** statement | while (condition) {<br>    expr<br>} | |
| **for** statement | for (var in seq) {<br>    expr<br>} | - Argument for *seq* is a vector of numbers<br>- Note that looping over a list requires double *[[ ]]* |
| Control statements: | | |
| break | When R encounters a *break*, the *for*/*while* loop is abandoned completely | |
| next | Skips to next iteration | |
| Other: | | |
| return(…) | | Note that a *return* statement requires parentheses *( )* |
| **Debugging** | | |
| 3 ways to debug | 1. Inserting print statements | E.g. {… *cat("A won.\n" …)* |
| | 2. Inserting a breakpoint in the function to keep track, i.e. "stop here and enter debugging mode" | *browser()*<br><br>Browser puts you in a debugging environment. |

| | | |
|---|---|---|
| | | In this environment, you can inspect the variables that were created within the function.<br>Use the following keys to navigate the function:<br>- **n**: evaluate next statement, stepping over function calls<br>- **s**: evaluate next statement, stepping into function calls<br>- **q**: quit the browser |
| | 3. Stepping through the function from start till finish | |
| debug(single_game)<br>single_game() | To debug function from its first line | |
| undebug(single_game) | When you have fixed the error and wish to return to normal execution of the function | |

| ***apply* family of functions** | | |
|---|---|---|
| apply(X, 1/2, fn, …) | Applies function to each row or column of a matrix separately | - Argument for *X* is the matrix<br>- To apply function row-wise (use *1*), column-wise (use 2)<br>- Argument for *fn* is the function you want to apply<br>- Additional arguments for the function can be added at "…"<br>- For example,<br>*col_means <- apply(X, 2, mean, trim= 0.1)*<br>where *trim* is an additional argument for *mean()* which specifies that outliers be removed |
| Apply function over a list or vector: | | |
| lapply()<br><br>lapply(x, fn, <optional additional arguments of fn>) | Iterates over each element of a list/vector *x* and **returns a list** (of same length as *x*) | - Returning a list is especially helpful when the **output** of each function call are **not** vectors/objects of the **same length**<br>- (Since lists can contain heterogenous components)<br>- E.g. Using on a list with components of different data types and length |
| sapply()<br><br>sapply(x, fn, USE.NAMES= T/F, …) | Similar to *lapply* but **returns a simplified** version of the list by turning it into a nicely-formatted vector/array/matrix<br><br>If unable to simplify, *sapply* will return the same output as *lapply* | - *USE.NAMES = T* uses *x* as names for the result unless it had names already<br>- Cases when *sapply* is unable to simplify its output:<br>1. When length of output changes for different input vector of the list (vectors of different sizes)<br>2. When the function *fn* returns *NULL* (E.g. a list of *NULLs* simplified would be a vector of one element – *NULL*, hence no longer a vector with the same length as input) |
| unlist(…) | Turns a list into a single vector | |
| vapply()<br><br>vapply(x, fn, FUN.VALUE, …, USE.NAMES = T/F) | Similar to *sapply* but requires the output format to be explicitly stated in the 3rd argument, *FUN.VALUE* | - E.g.<br>*FUN.VALUE = numeric(3)*<br>specifies the length of the returned vector as 3 or<br>*FUN.VALUE = logical(1)*<br>- *USE.NAMES = T* by default and generates a named array |

<table>
<tr><td colspan="4"><b>Additional</b></td></tr>
<tr><td rowspan="2"><i>History</i> pane</td><td colspan="2">[Click on line of code] + (TO SOURCE) button</td><td>Saves code to source code from <i>history</i></td></tr>
<tr><td colspan="2">[Click on line of code] + ENTER</td><td>To directly run code in console</td></tr>
<tr><td><i>Viewer</i> pane</td><td colspan="2"><i>rsconnect</i> package</td><td>Publishes <b>Shiny</b> apps to <i>shinyapps.io</i></td></tr>
<tr><td rowspan="3">Packages</td><td>dplyr package</td><td>Manipulates data</td><td></td></tr>
<tr><td>ggvis package</td><td>Visualises data</td><td>Creates graphs as HTML objects</td></tr>
<tr><td>ggplot2 package</td><td></td><td>Creates graphs in PDF</td></tr>
<tr><td></td><td colspan="2"></td><td></td></tr>
<tr><td></td><td colspan="2"></td><td></td></tr>
<tr><td>Single values</td><td colspan="2">Can be treated as vectors of length 1</td><td></td></tr>
<tr><td>Matrix index</td><td colspan="2">The index of the entries in a matrix is read column by column, from top to bottom</td><td>This is when there is only a single value in the <i>[x]</i></td></tr>
<tr><td>Logicals are numbers</td><td colspan="2">Because of the binary nature of logical values, they're often represented with TRUE as 1 and FALSE as 0</td><td>For example,<br>- <i>FALSE – TRUE = -1</i><br>- <i>T + T + F + T + F + F + T = 4</i><br>- <i>1&&1 = TRUE</i><br>- <i>0&&1 = FALSE</i><br>- <i>1||0 = TRUE</i></td></tr>
<tr><td>String formats in R</td><td colspan="2">1. Extended regular expression (default)<br>2. Perl<br>3. Literal regular expressions</td><td>For more technical details, enter <i>?regex</i> at the prompt.</td></tr>
<tr><td rowspan="7">Escape sequences ( \ )</td><td colspan="2">The \ is used to invoke an escape sequence which lets you enter characters that control the format and spacing of a string, rather than being interpreted as normal text</td><td rowspan="7">A full list can be found by entering <i>?Quotes</i> at the prompt</td></tr>
<tr><td>Escape sequence</td><td>Result</td></tr>
<tr><td>\n</td><td>Starts a newline</td></tr>
<tr><td>\t</td><td>Horizontal tab</td></tr>
<tr><td>\b</td><td>Invokes a backspace</td></tr>
<tr><td>\\</td><td>Used as a single backslash</td></tr>
<tr><td>\"</td><td>Includes a double quote</td></tr>
<tr><td>Levels, names</td><td colspan="2">R attributes</td><td></td></tr>
<tr><td>Lists</td><td colspan="2">- Often used to return output from various R functions. But they can quickly become large objects in terms of system resources to store. - Generally recommended that when you have only one type of data, you should stick to using basic vector, matrix, or array structures to record and store the observations</td><td></td></tr>
<tr><td>NA vs NULL<br><i>("missing" vs "empty" entity)</i></td><td colspan="2">An instance of <i>NA</i> clearly denotes an existing position (i.e. index position provided) that can be accessed and/or overwritten if necessary – not so for <i>NULL</i></td><td>An example,<br><b><i>c(NA, NA, NA)</i></b><br><i>"3 possible slots with unrecorded observations"</i><br><b><i>c(NULL, NULL, NULL)</i></b><br><i>"emptiness 3 times – a single, unsubsettable, empty object"</i></td></tr>
</table>

<table>
<tr><td>Coercion</td><td colspan="2">Converting from one object or data type to another</td><td colspan="3">For instance,<br>1:4 + c(T, F, F, T)<br>= 2 2 3 5<br>paste(“Definitely”, 4, “you”)<br>= “Definitely 4 you”</td></tr>
<tr><td rowspan="6">Words/letters reserved by R and to be avoided as names</td><td>FALSE</td><td>break</td><td>next</td><td>t</td><td rowspan="6">Note that R is case-sensitive</td></tr>
<tr><td>TRUE</td><td>else</td><td>repeat</td><td>C</td></tr>
<tr><td>NA</td><td>for</td><td>while</td><td>D</td></tr>
<tr><td>NaN</td><td>function</td><td>c</td><td>F</td></tr>
<tr><td>NULL</td><td>if</td><td>q</td><td>I</td></tr>
<tr><td>Inf</td><td>in</td><td>s</td><td>T</td></tr>
<tr><td rowspan="20">Operator precedence</td><td colspan="4">Listed from highest to lowest precedence:</td><td rowspan="20">??precedence</td></tr>
<tr><td colspan="2">:: , :::</td><td colspan="2">Access variables in namespace</td></tr>
<tr><td colspan="2">$ , @</td><td colspan="2">Component / slot extraction</td></tr>
<tr><td colspan="2">[ , [[</td><td colspan="2">Indexing</td></tr>
<tr><td colspan="2">^</td><td colspan="2">Exponentiation (right to left)</td></tr>
<tr><td colspan="2">- , +</td><td colspan="2">Unary minus and plus</td></tr>
<tr><td colspan="2">:</td><td colspan="2">Sequence operator</td></tr>
<tr><td colspan="2" rowspan="2">%any%</td><td colspan="2">Special operator ( including %% and %/% )</td></tr>
<tr><td colspan="2">Modulo ( %% ) returns remainder</td></tr>
<tr><td colspan="2">* , /</td><td colspan="2">Multiply, divide</td></tr>
<tr><td colspan="2">+ , -</td><td colspan="2">(Binary) add, subtract</td></tr>
<tr><td colspan="2">< , > , <=, >= , == , !=</td><td colspan="2">Ordering and comparison</td></tr>
<tr><td colspan="2">!</td><td colspan="2">Negation</td></tr>
<tr><td colspan="2">& , &&</td><td colspan="2">And</td></tr>
<tr><td colspan="2">| , ||</td><td colspan="2">Or</td></tr>
<tr><td colspan="2">~</td><td colspan="2">As in formulae</td></tr>
<tr><td colspan="2">-> , ->></td><td colspan="2">Rightwards assignment</td></tr>
<tr><td colspan="2"><- , <<-</td><td colspan="2">Assignment (right to left)</td></tr>
<tr><td colspan="2">=</td><td colspan="2">Assignment (right to left)</td></tr>
<tr><td colspan="2">?</td><td colspan="2">Help (unary and binary)</td></tr>
<tr><td>Element-wise sum</td><td colspan="2">Summing up elements of 2  or more arrays according to their index positions</td><td colspan="3">E.g.<br>vec1 <- c(1, 2, 3)<br>+ + +<br>vec2 <- c(4, 5, 6)<br>gives c(5, 7, 9)</td></tr>
</table>

| **String manipulation** | | |
|---|---|---|
| library(“tidyverse”)<br>library(“stringr”) | Packages required | |
| “…”<br>“…‘…’…” | String creation | Convention is to use double quotes, and to use single quotes within a string if necessary |
| str_length(…) | Computes length of string | Argument can be a vector of strings |
| str_split(…, split= “…”) | Takes in a string (or a vector of strings) and splits each string, returning a list (where each component is the splitted form of each string) | - When *split*= *“”*, *str_split* splits a string into individual characters (including space)<br>- Note that when splitting just one string, a list of one component will still be returns (i.e. use *[[1]]* to retrieve result out) |
| str_c(…)<br>str_c(…, sep= “…”) | Combines strings | - Alternative to *paste()*<br>- There can be more than 2 arguments, or a combination of vector of strings and strings<br>E.g. ***<br>*str_c(“x”, c(“a”, “y”), “z”, sep= “.”)*<br>gives *“x,a,z”* and *“x,y,z”*<br>E.g. ***<br>*strc_c(“hawker”, “ctre”, 1:3, sep= “_”)*<br>gives *“hawker_ctre_1”*,<br>*“hawker_ctre_2”*<br>and *“hawker_ctre_3”*<br>(in addition, if *“…001”, “002”* etc. is desired, use<br>*sprintf( “%03d”, 1:3*) |
| str_sub(x, start= …, end= …) | Subsets a string where arguments for *start* and *end* are indices | - Not necessary to give *start* and *end* together<br>- E.g. *start*= *-1* gives the last character<br>- E.g. *end*= *-1* gives entire string |
| str_view(x, pattern= …, match= T/F) | When *match*= *T*, R will enclose the matching character/*pattern* of the string *x* with a grey rectangle in the *Viewer* pane | - Used mainly for testing<br>- For more detailes, refer to *?str_view* |
| Basic matches/possible forms of ***pattern*** argument: | | |
| str_view(x, “^a”) | To match “*a*” at the beginning of a string | |
| str_view(x, “a$”) | To match pattern *a* at the end of a string | |
| str_view(x, “[ae]$”)<br>str_view(x, “[1:5]”)<br>str_view(x, “[f-z]”) | To match “*a*” or ”e” at the end of a string<br>To match numbers 1 to 5 in a string<br>To match letters “*f*” to “z” in a string | |
| str_view(x, “.a.”) | To match a string of 3 characters with “*a*” in the middle | |
| str_detect(x, pattern= “…”) | Actually detects the matching pattern in a string and gives a logical value for each string on whether it matches | - Use *which()* to obtain the indices of matching strings in a vector<br>- Use *which.min()* to obtain the index of the 1st *FALSE* logical in the vector<br>Use *which.max()* to obtain the index of the 1st *TRUE* logical in the vector |

| **Regular expression *(?regex)*** | | |
|---|---|---|
| Pattern existence | | |
| grep()<br>grep(pattern, x) | Pattern matching – Search for matches to argument *pattern* within each element of a character vector *x*<br>Returns a vector of **indices** of elements which matched | |
| grepl()<br>grep(pattern= <regex>, x= <string>) | Similar to *grep()* but returns a **logical** vector, *TRUE* or *FALSE* (match or not for each element of *x*) | Output of *grepl()* is similar when *which()* is applied on result of *grep()* |
| Pattern replacement/extraction | | |
| sub()<br>sub(pattern= <regex>, replacement= <str>, x= <str>) | Pattern replacement – only looks for 1st occurrence of pattern | |
| gsub() | Similar to *sub()* but looks for **all** occurrences of pattern in *x* and replaces them | |
| regex for argument pattern | | |
| "\\s" | Matches a space | "s" is normally a character but escaping it ( \\ ) makes it a metacharacter |
| "\\." | Escapes a full-stop *( . )* to make it a regular character | Note that forward-slash ( \\ )is used |
| "[0-9]+" | Matches numbers 0 to 9 at least once *( + )* | |
| "([0-9]+)" | The added parentheses make parts of matching string available to define replacement | - A \\1 in the replacement argument of *sub()* gets set to the string that is captured by the regular expression *[0-9]+*<br>- \\1 references content inside parentheses *( )* and has the entire match get replaced by this number<br>- For example:<br>x is *"Won 3 wombats."*<br>and pattern is ".*\\s([0-9]+)\\s.*$"<br>and replacement is \\1<br>gives *"3"* |
| ".*" | Any character that is matched zero or more times (i.e. matches any character(s)) | For example, the ( .* )<br>.*\\s([0-9]+)…<br>in *"Won 1 Oscar."*<br>is *"Won"* |
| "a\|i\|o" | Match any of the characters, for example, "a", "i" or "o", found | |
| | | |
| | | |
| | | |

| **R Markdown (*.Rmd)***<br>(R – File > New File > R Markdown) | | |
|---|---|---|
| #, ##, ### | Makes a title/header<br># : 1st-level header<br>## : 2nd-level header<br>### : 3rd-level header | |
| *…*, **…** | *Italic*, **Bold** text | |
| [name](Link) | Creates a link | |
| * item1<br>* item2<br>* item3<br>…<br>OR<br>1. …<br>2. …<br>3. …<br>… | Makes a list | Note that you have to put a **blank line** before any list |
| $$…$$ | Embeds equations in its own centered-block | Standard LaTeX math symbols can be used |
| $…$ | Embed equations **inline** | |
| **Knitr** | | |
| ` **r** …`<br>OR | Embed a line of code *within* text | - R will run the code and replace it with its result if necessary, e.g. character string, number |
| ``` {r}<br>…<br>```<br>OR<br>``` {r **engine**= …}<br>…<br>```<br>OR<br>``` {r **warning**= F,<br>**error**= F,<br>**message**= F}<br>…<br>```<br>OR<br>``` {r **echo**= F,<br>**eval**= F,<br>**results**= 'hide'}<br>…<br>```<br>OR | Embed a chunk of code | - When the code is rendered, R will execute the code<br>- If the code returns any results, R will add them to the report<br>- Each R Markdown document is given a fresh empty R session, hence take note to:<br>1) Define any R objects the document uses<br>2) Load any packages it uses, i.e. *library(…)*<br>- *{r engine= …}* allows the code to be written in another language, e.g. "…= *python*"<br>- By default, R Markdown will include error messages in the report<br>- Setting *warning*, *error*, *message* to *FALSE* will tell R to not include the corresponding type of messages in the output<br>- To ensure that messages when generating packages do not appear in the report, separate *library(…)* into its own code chunk at the beginning<br>- *{r echo= F}* - Will **not display code** in the final document<br>- Will run and display results unless told otherwise) |

| | | | |
|---|---|---|---|
| ``` {r **fig.height**= …, **fig.width**= …, **out.width**= “50%”}<br>…<br>```<br><br>OR<br>``` {r <label>, …}<br>…<br>```<br>AND<br>``` {r **ref.label**= **“**label**”**, echo= F}<br>…<br>```<br><br>OR<br>``` { r **cache**= TRUE}<br>…<br>``` | | - {*r eval= F*} | - Will **not run code** or include results<br>- Will display code unless told otherwise) |
| | | - {*r results=* ***‘hide’***} | - Will **not display results** of the code<br>- Will run code and display code itself unless told otherwise) |
| | | - {*r fig.height*= …, *fig.width*= …} controls the size of figures in the document<br>- *out.width*= *“50%”* will assign 2 plots side by side | |
| | | - {*r <label>*, …} : assigns the code chunk a label<br>- There is no need for the label to be in string (“ “)<br>- {*r ref.label*= …} : Helps to refer to previously defined and labelled code chunks<br>- The label has to be written in string<br>- Knitr will **copy the code** chunk referred to and **repeat it** in the current code chunk<br>- Useful when separating R code and R output in the output document without code duplication | |
| | | {*r cache*= *TRUE*} prevents the code chunk from running all the time we knit the file especially when the code chunk is computationally expensive | |
| **Pandoc** | | | |
| Yet Another Markup Language **(YAML)** header | ---<br>title: “…”<br>author: …<br>date: …<br>output: …<br>--- | - Contains some metadata<br>- *output* can be *html_document, pdf_document, word_document, beamer_presentation* (PDF format for slides), HTML slideshows like: *slidy_presentation, ioslides_presentation,* and *md_document* (markdown file) | |

| | | | | |
|---|---|---|---|---|
| Overwriting default code highlight style | **Document type** | **YAML** | | |
| | PDF | ---<br>title: …<br>output:<br>pdf_document:<br>highlight: …<br>--- | Examples for *highlight*:<br>- zenburn | |
| | HTML | ---<br>title: … | Example for *theme*: | |
| | | | - default | - spacelab |

| | | | | |
|---|---|---|---|---|
| | | output:<br>html_document:<br>theme: …<br>--- | - cerulean<br>- flatly<br>- united | - journal<br>- readable<br>- cosmo |
| | | ---<br>title: …<br>output:<br>html_document:<br>toc: **t**rue<br>number_sections: **t**rue<br>--- | - *toc* being "Table of Contents"<br>- Note that '*true*' is with a lowercase 't' | |
| | Shiny | ---<br>title: …<br>output:<br>slidy_presentation:<br>incremental: true<br>runtime: **shiny**<br>--- | - Makes R Markdown interactive with web apps. like data explorer and dashboards<br>- Ensure that output is HTML-based | |
| #, ## | Creates a new slide at each 1st and 2nd header in the document | | | |
| *** | Insert additional slide breaks without a header | Uses Markdown's horizontal rule syntax | | |

| Importing Data into R | |
|---|---|
| ***Text files*** | |
| - Files with an optional header (listing column names), and observations separated by commas within each row<br>After reading it into R, check the following:<br>1. Were the **correct number of rows and columns** read in?<br>2. Were the column names and the **column classes** correctly assigned to their variables?<br>3. Were the **missing values** correctly read in? | - Tip: Try to open file in a text editor first<br>- Gives some indication of the amount of metadata, presence/absence of headers, and how many columns there are in your data<br>- Sometimes it is easier to clean the data there than in R<br>- **Note down how many lines** there are in csv file and if it corresponds to the **number of observations** there should be in the resulting data frame |

| | | | | |
|---|---|---|---|---|
| Commands to read in a text-file into R | read.csv() | "Comma-separated values" | Default *header* = *T* | |
| | read.delim() | "Tab-delimited" ( \t ) | | |
| | read.table() | Exotic file format | Default *header* = *F*<br>*sep* = "" | |
| | read.csv2() | Due to regional differences | *sep* = ";" | Decimal:<br>*dec* = ","<br>(Decimal points in commas) |
| | read.delim2() | | *sep* = "\t" | |
| Arguments<br>( <arg> = … ) | *file* | string | Filename | |
| | *header* | *T/F* | Absence/presence of a header row<br>(i.e. Read 1st row as header?) | |
| | *skip* | int | Number of comment lines at the beginning<br>(Esp. if there is metadata) | |
| | *stringAsFactors* | *T/F* | Whether to read string values in as factors or not | |
| | *sep* | string | ** Only for *read.table()* **<br>E.g. *sep* = "/" | |
| | *colClasses* | string vector | Specify the column types/classes with a vector of strings representing classes<br>E.g.<br>*"character"*<br>*"integer"*<br>*"factor"*<br>*"numerical"*<br>*"logical"*<br>*"NULL"*<br>(** Skips the column, i.e. doesn't load in data frame) | |
| | *col.names* | char vector | Gives names to variables | |
| | *na.strings* | string/ char vector | - Specifies values which are to be interpreted as *NA* values<br>- For example, *na.strings* = "-" | |
| When read.csv() *fails* | readLines() | - Works easiest when data is **text data**, not binary<br>- Gets data into R as a character vector (i.e. vector of strings where each line is read in as a string)<br>- Continue to manipulate data by parsing the strings using *stringr* functions like *str_detect_all()*, *str_split()* etc | | |
| Arguments | *file* | string | Path to file | |
| Using ***readr*** package (gives a *tibble*) | read_csv() | "Comma" | Default *stringsAsFactors* = *F* | |
| | read_tsv() | "Tab" | | |
| | read_delim() | | | |

<table>
<tr><td rowspan="6">Arguments</td><td><i>file</i></td><td>string</td><td>Filename – gives path to file</td></tr>
<tr><td><i>col_names</i></td><td>char vector/F</td><td>When set to <i>F</i> gives column names of “<i>X1, X2, X3…</i>”</td></tr>
<tr><td><i>col_types</i></td><td>string<br>OR<br>list (with collectors as members – see below)</td><td>- Manually decide classes<br>E.g. “<i>ccdd</i>”<br>- Else, by default (when set to <i>NULL</i>), R decides by the first 30 rows
<table>
<tr><td>c</td><td>character</td><td>i</td><td>integer</td></tr>
<tr><td>d</td><td>double</td><td rowspan="2">l (lowercase L)</td><td rowspan="2">logical</td></tr>
<tr><td>_ (underscore)</td><td>Skips column</td></tr>
</table></td></tr>
<tr><td><i>delim</i></td><td>String</td><td>** Only for <i>read_delim()</i> **<br>Equivalent to <i>sep</i> in <i>read.table()</i></td></tr>
<tr><td><i>skip</i></td><td>int</td><td>- Specifies the number of lines to ignore in the flat-file before actually starting to import the data<br>- For instance, <i>skip</i> = <i>x</i> skips the first <i>x</i> row(s)<br>- Note that it will also skip the first line which may contain the header (remedy it by specifying <i>col_names</i>)</td></tr>
<tr><td><i>n_max</i></td><td>int</td><td>Specifies the number of rows to read/ number of lines actually being imported</td></tr>
<tr><td rowspan="3">++ Collectors</td><td colspan="3">Another way of setting types of imported columns</td></tr>
<tr><td><i>col_integer()</i></td><td>Column should be interpreted as an integer</td><td></td></tr>
<tr><td><i>col_factor(levels, ordered = T/F)</i></td><td>Column should be interpreted as a factor with <i>levels</i></td><td>Argument for levels is in the form of a vector</td></tr>
<tr><td>Using <b><i>data.table</i></b> package (Extremely fast, good for huge files)</td><td>fread()</td><td colspan="2">- Similar to <i>read.table()</i><br>1. Automatically formats to read column names if present (if not, new column names are created)<br>2. Able to infer column types and separators<br>3. Possible to specify numerous parameters<br>4. An improved version of <i>read.table()</i><br>5. Fast, convenient and customisable</td></tr>
<tr><td rowspan="3">Arguments</td><td>path</td><td>string</td><td>Path to file</td></tr>
<tr><td><i>drop</i></td><td rowspan="2">vector (index(s)/ names of variable(s))</td><td>Variables you wish to drop</td></tr>
<tr><td><i>select</i></td><td>Variable you wish to keep</td></tr>
</table>

<table>
<tr><td colspan="3"><b><i>Excel</i> files</b></td><td></td></tr>
<tr><td colspan="3">Excel data – typical structure: different sheets with tabular data</td><td>File formats: <i>xls, xlsx</i></td></tr>
<tr><td rowspan="2">Using <b><i>readxl</i></b> package</td><td><i>excel_sheets()</i></td><td colspan="2">- List different sheets<br>- Good for finding out which sheets are available in the workbook<br>- Argument is the path to file<br>- Returns character vector of names of sheets</td></tr>
<tr><td><i>read_excel()</i></td><td colspan="2">- Import data into R<br>- Gives a <i>tibble</i> – an improved version of a data frame</td></tr>
<tr><td rowspan="2">Arguments<br>(* for <i>read_excel</i> *)</td><td>path</td><td>string</td><td></td></tr>
<tr><td><i>sheet</i></td><td>int/string</td><td>- Imports sheet of given number or by name of sheet<br>- Use with<br><i>lapply(excel_sheets(“…”), read_excel, path = “…”)</i></td></tr>
</table>

| | | | | | | |
|---|---|---|---|---|---|---|
| | *col_names* | *T/F/* char vector | - Manually specify with a character vector of the column names<br>- If *col_names* = *F*, R chooses and assigns names itself | | | |
| | *col_types* | *NULL/* string | *NULL* | Default: R guesses data-type | *“numeric”* | |
| | | | | | *“date”* | |
| | | | *“text”* | | *“blank”* | Ignores that column |
| | *skip* | int | Note to specify column names if it skips the row specifiying it or to set *col_names* = *F* | | | |
| | *n_max* | int | Maximum number of rows to look up | | | |
| | *range* | string | Sets **precise region to use** where empty cells will be filled with *NA* (e.g. *“B3:D87”*) | | | |
| Using ***gdata*** package | - Supports *XLS, XLSX* with additional driver<br>- Elegantly extends *utils* package<br>( XLS ---<Perl script>---> CSV ---<read.csv()---> … ) | | | | | |
| | *read.xls()* | | | | | |
| Arguments | path | string | | | | |
| | *sheet* | int/string | An int, string argument refers to the sheet number and name of the sheet respectively | | | |
| | + arguments for *read.csv()* | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| Using ***XLConnect*** package | | | Arguments | | |
| | *loadWorkbook(“…”)* | | path | string | Creates connection to workbook |
| | *getSheets(…)* | | workbook | object | Lists sheets in an Excel file |
| | *readWorksheet(…)* | | workbook | | Actually imports data from a sheet |
| | | | *sheet* | string | |
| | | | *startRow* | int | |
| | | | *endRow* | | |
| | | | *startCol* | | |
| | | | *header* | T/F | |
| | Adapting sheets (1st argument is *book*) | *createSheet()* | *name* | string | Creates a new sheet of *name* in book |
| | | *writeWorksheet()* | *data* | | Writes data to a worksheet (adds *data* to a sheet in an Excel file) |
| | | | *sheet* | string | |
| | | *saveWorkbook()* | *file* (new file name) | | Help save work as a new file to prevent overwriting the old version |
| | | *renameSheet()* | *oldname* | | Renames sheet in a Excel file |
| | | | *newname* | | |
| | | *removeSheet()* | *sheet* | | Removes sheet in an Excel file |

| ***JSON files*** | |
|---|---|
| JSON – text format for storing structured data | - Full description of format at http://www.json.org/<br>- Packages to generate and parse JSON files:<br>*rjson, RJSONIO, jsonlite* |

| | | | |
|---|---|---|---|
| Built on two structures:<br>1. **object** – unordered collection of name/value pairs<br>2. **array** – ordered list of values (of any type)<br>Repeated stacking of these structures can store quite complex data structures | | **object** | {string : value,<br>string : value, …}<br>{*members*} |
| | | *members* | *pair*<br>*pair, members* |
| | | *pair* | string : *value* |
| | | **array** | [value, value, … ]<br>[*elements*] |
| | | *elements* | *value*<br>*value, elements* |
| | | *value* | String (in **double quotes**), number, **object**, **array**, *true, false, null* (represents values that are missing of unset) |
| Using ***jsonlite*** package | *fromJSON(txt)* | - Read *JSON* objects from text files (using file path), the web (URL), or even straight from the console<br>- Used directly when there is ***only one*** *JSON* object<br>- ** Reading ***multiple*** *JSON* objects from a file: read each line into R, then apply *fromJSON()* to each of them (since *JSON* objects are separated by commas and not new-line characters (*n*))<br>- For example,<br>*all_lines <- readLines(“../data/read_json_02.txt”)*<br>*json_list <- lapply(all_lines, fromJSON)*<br>where each object is a component of the list | |
| | | - **Single quotes** are used to indicate **text** (e.g. ‘ *[12, 3, 7]* ‘ )<br>- Sometimes *jsonlite* cannot flatten text and will put objects in a list<br>- If the values are **not homogenous** (of the **same type**)<br>(e.g. ‘ *[12, “a”, 7]* ‘ ),<br>jsonlite reads/coerces them in as a character vector<br>(e.g. *c(“12”, “a” , “7”)*)<br>- Missing values (‘*null*’) are appropriately codes as *NA* within R | |
| | *toJSON()* | Converts *R* object into *JSON* object | |
| | *prettify()* | Prints a *JSON* string (obtained using *readLines*) with indentation for easier understanding | |
| | *minify()* | Opposite of *prettify()*, condenses the file | |
| Example of data from the web | *https://data.gov.sg/* | - **Developers** sub-page of website: instructions on downloading data<br>- **Data API** link on dataset: shows the resource id for the data<br>- Essentially what is needed: **identify** resource **id** for data set and **tag** it onto a **template URL**<br>- However, there is a **limit** on the number of **records** that can be **retrieved per query** hence necessary to **run a loop** until all records have been retrieved | |
| | *https://api.nasa.gov/index.html* | Brief summary of steps on how to use NASA API:<br>1. Go to link to apply for an API key<br>2. Fill up your name and last name and click Signup<br>3. Copy and save your key into a text file<br>- Each API request for data is in the form of a web URL | |

|  |  |  |
|---|---|---|
|  |  | - The key that have been assigned must be a part of that URL<br>- Note to make sure that there is no spaces in a https request (if necessary, replace it with a ‘%’)<br>- The ***curl*** package is needed to download images through URLs<br>- For example,<br>*library(curl)*<br>*curl_download(img_url$url, “../figs/nasa_1.jpeg”)*<br>saves the image in a local file |
|  | http://www.weather.gov.sg/climate-historical-daily/ | - Link to website contains historical records for weather conditions in Singapore<br>- Location of CSV file containing the weather for each month of each station can be found by **placing your mouse over the link** to the CSV file<br>- Since there is no API to retrieve the data, the **general format of the URL** must be checked to automate the collection of data using a script<br>- For example, to create a loop that downloads files from Jan 2009 to Dec 2016:<br>*yrs <- 2009:2016*<br>*mths <- sprintf(“%02d”, 1:12)*<br># where “%02d” represents an integer<br>root_s <- http://www.weather.gov.sg/files/dailydata/<br>for (yy in yrs) {<br>for (mm in mths) {<br>uu <- paste(root_s, “DAILYDATA_S106_”, yy, mm, “.csv”, sep = “”)<br>outname <- paste(“../data”, yy, mm, “.csv”, sep = “”)<br># Gives name of file saved<br>cat(outname, “\n”)<br>try(download.file(uu, outname) }}<br># *** try function allows the loop to continue if something causes the download to fail (e.g. no file for that month or option for the file not in website) for a particular month (gives a warning and goes to next iteration) |

| **Histogram** |  |  |  |
|---|---|---|---|
| hist() | Divides the range of values into bins, then counts the number of values that fall into each bin |  |  |
| Arguments | *x* | int vector | The vector of values for which the histogram is desired |
|  | *main, xlab, ylab* | string |  |
|  | *xlim, ylim* | int vector | Specifies the range of *x* and *y* values |
|  | *freq* | *T/F* | - *freq* = *F* alters the histogram such that the height of each bar does not **represent a count** – instead, it is the height such that the area of all bars add up to 1<br>- Makes histogram closer in spirit to a probability density function (**pdf**) |

| | | | |
|---|---|---|---|
| | *col* | string/int/*NULL* | - Specifies the colour used to fill the bars<br>- By default, *col = NULL* yields unfilled bars |
| | *border* | string | Specifies the colour of border around the bars (e.g. *“white”*) |
| | *breaks* | int vector | - Specifying the **breakpoints between** histogram **cells** (i.e. controls the thickness of bars)<br>- For example,<br>*breaks = seq(0, 200, by = 10)* |
| | *axes* | *T/F* | *axes = T* draws axes if the plot is drawn |
| | *plot* | | - If *plot = T*, a histogram is plotted<br>- Else, a list of breaks and counts is returned |
| | *labels* | *T/F*/char vector | Gives the option to draw labels on top of bars |

<table>
<tr><td colspan="4">**_dplyr_ package**</td></tr>
<tr><td colspan="4">A grammar of data manipulation</td></tr>
<tr><td rowspan="2">*tibble*</td><td colspan="3">- Makes data easier to look at and also easier to work with<br>- Changes how R displays data without changing the data's underlying data structure<br>- A tibble inherits the original class of its input so manipulating it as its original class is possible (E.g. both a tibble and a data frame)<br>Differences of *tibble* from data frame:<br>1. When printing a *tibble*, it does not print all the rows and all the columns (fits what it can in a console) – making it better for inspecting a data frame<br>2. It does not do partial matching when extracting columns (E.g. *dataframe$var1*)<br>3. If you request for a column that does not exist, it will generate a warning (in contrast, a data frame will return *NULL*)</td></tr>
<tr><td>*tbl_df()*</td><td>Input: a dataset</td><td>Columns are variables and rows are observations</td></tr>
<tr><td rowspan="2">pipe operator ( %>% )</td><td colspan="3">- *"then"*<br>- From *magrittr* package but R auto-loads it through *tidyverse* package<br>- Allows extraction of the *1st argument* of a function from the argument list to put it in front of the function<br>- Solves the Dagwood Sandwich problem<br>- Under the hood, it does:<br>*x %>% f(y)* into *f(x, y)*<br>*x %>% f(y) %>% g(z)* into *f(x, y) %>% g(z)* which is *g( f(x, y), z)*</td></tr>
<tr><td colspan="2">*object %>% fn(arg2, arg3, …)*<br>*OR*<br>object %>% fn1(arg2, …)<br>%>% fn2(arg2, …)<br>%>% fn3(arg2, …)<br>…</td><td>- Passes object as first argument of function<br>- If want to use it in another position, use period ( . )</td></tr>
<tr><td colspan="4">**_dplyr_ verbs:**<br>(Note: The verbs do not change the data it is called on – to store the result, explicitly assign it to a variable)<br>Common properties:<br>1. First argument is data frame<br>2. Subsequent arguments describe what to do with the data frame, using variable names *without* quotes<br>3. Output is a new data frame</td></tr>
<tr><td>**_select_**</td><td colspan="3">Returns a subset of the **columns**<br>(Pick variables/columns by name)</td></tr>
<tr><td rowspan="8">Arguments<br>*(df, var1, var2, …)*</td><td>*data set*</td><td>object</td><td>Can be a tibble or data frame</td></tr>
<tr><td>*variables*</td><td>variable names<br>(*without* quotes)<br>/ int</td><td>Indexing variables is possible<br>Examples:<br>*select(df, var1 : var5)*<br>selects all columns between *var1* and *var5* (inclusive)<br>*select(df, 1 : 4, - 2)*<br>*select(df, - c(varX : varY))*</td></tr>
<tr><td colspan="3">Helper functions to assist selection: (Note: *with* quotes)</td></tr>
<tr><td>*starts_with()*</td><td rowspan="4">string</td><td>Matches column names that begin with specified string</td></tr>
<tr><td>*ends_with()*</td><td>Matches column names that end with specified string</td></tr>
<tr><td>*contains()*</td><td>Matches column names that contain specified string</td></tr>
<tr><td>*matches()*</td><td>Matches columns whose names match the provided **regular expression**</td></tr>
<tr><td>*num_range()*</td><td>string, int<br>vector</td><td>Example:<br>*num_range("X", 1:5)* gives</td></tr>
</table>

| | | | |
|---|---|---|---|
| | | | *X01, X02, …, X05* |
| | *one_of()* | char vector | Finds variable name(s) that appears in character vector given |
| ***filter*** | Returns a subset of the **rows**<br>(Pick observations/rows by the values in their columns) | | |
| | Note: There is no limit to number of criteria satisfied (R combines them with the *AND* operator) | | |
| Arguments<br>*(df, condition1, condition2, …)* | *data set* | | |
| | *logical test/condition* | expression/ a logical vector | - Note: To use other operations, like the *OR* operator, then there is a need to manually specify them<br>- Some examples,<br>*var1 %in% c("a", "b", "c")*<br>returns *TRUE* if *x* is in the vector<br>and is equivalent to:<br>*var1 == "a" \| var1 == "b" \| var1 == "c"*<br>OR<br>*var2 == 8*<br>OR<br>*var3 == 11 \| var3 == 12*<br>- By default, only columns where condition is *TRUE* is kept – if a cell is missing (*NA*), then that row is dropped<br>- *(Keep NA values)* To include missing values observations, use an expression of the form:<br>*is.na(var4) \| var4 == …* |
| ***arrange*** | Reorders the **rows** according to single or multiple variables<br>(Reorder rows) | | |
| Arguments<br>*(df, var1, var2, …)* | *data set* | object | |
| | *variables to arrange by* | Variable/column name(s) | - Sequence of variables given will be reflected in the reordered columns<br>- Data can be arranged by multiple variables (in instances of **tie-breaks**)<br>- If a **factor** variable is passed in, R will order the data by the **order of levels** in the factor<br>- By default, R reorders from **smallest to largest**<br>- To reorder rows from the **largest to smallest** (descending) values of a variables, use:<br>***desc****(varX)*<br>- Missing values will always be placed at the end, whether in ascending or descending order |
| ***mutate*** | Adds/creates new **columns** from existing data – element-wise operation<br>(Create new variables/columns) | | |
| | - Adds new columns to the **end** of the dataset | | |
| Arguments<br>*(df, newVar = <fn of old vars>, …)* | *data set* | object | |
| | *new variable expression* | expression | For example,<br>*newVar = var1 – var2*<br>where "*newVar*" will be the name of the new column |
| | *<variable name> = NULL* | | Drops the variable/column<br>(Alternative to negative selection) |
| | Helper functions to assist creation: | | |

| | | | |
|---|---|---|---|
| | *(Any arithmetic operation)* | | - The classic: ( + ) , ( - ), ( / ), ( ** )<br>- Logarithmic: *log*, *log10*<br>(Useful for transformations when variable is highly skewed) |
| | *lag()* | vector | - Shifts the elements down in *x* where *x[i]* becomes *x[i+1]* and *x[1] = NA*<br>- Compute running differences: *x – lag(x)*<br>- Find when a value has changed: *x != lag(x)*<br>- If you call *lag()* without loading *dplyr*, R will apply the *lag()* from the *stats* package – lags the **time index** of a time series |
| | *lead()* | | - Shifts the elements up in *x* where *x[i] = x[i-1]* and *x[n] = NA* where n is the length of *x* |
| | *cumsum()*<br>*cumprod()*<br>*cummin()*<br>*cummax()* | | Computes cumulative and rolling aggregates |
| | *min_rank()* | | - Assigns rank 1 to the smallest number, rank 2 to the next, and so on<br>- For example,<br>*x <- c(1, 2, 3, NA, 3, 4)*<br>*min_rank(x)*<br>gives *c(1, 2, 3,* ***NA****, 3,* ***5****)* |
| | *<your own function>* | | |
| **transmute** | Keeps only the new variables/columns created<br>(Reflection of *mutate()*) | | |
| Arguments | Same as mutate | | |
| **group_by** | - Defines groups within data set<br>(Splits a dataset by values in a variable) | | |
| | - Does not change dataset – adds *Groups* attribute to data frame<br>- To remove/ redefine groups, use: ***ungroup****(df)*<br>- When used with ***summarise()***: summarising statistics are calculated for the different groups separately<br>- When used with ***mutate():*** new variables are calculated independently for each group<br>- Particularly useful when *mutate()* uses ***rank()*** function: takes a group of values and calculates the rank of each value within the group (smallest to largest)<br>- When used with *dplyr* verbs, actions are automatically applied "by group" | | |
| Arguments<br>*(df, var1, …)* | *data set* | object | |
| | *variables to group by* | variable names | |
| **summarise** | Reduces each group to a **single row** by calculating aggregate measures<br>(Collapses many values down to a single one) | | |
| | - Resulting data set consists of a **single row** instead of an entire new column (like *mutate*)<br>- *summarise* will not return an altered copy of the data set it is summarising, instead it **builds a new data set** that contains only the summarising statistics<br>- When paired with *group_by()*, the unit of analysis is changed from the complete dataset to individual groups | | |
| Arguments | *data set* | object | |

| | | | |
|---|---|---|---|
| *(df, newRow = <fn>, …)* | *aggregating function* | expression | - Note to use only **functions** which **only return 1** result/**output**/vector of length 1<br>- Classic aggregating functions:<br>Measures of rank:<br>*min()*<br>*max()*<br>*quantile(…, p)*<br>where pth quantile of input<br>Measures of location:<br>*mean()*<br>*median()*<br>Measures of spread:<br>*mad()*<br>*sd()*<br>*var()*<br>*IQR()*<br>i.e. inter-quantile range of input<br>Others:<br>*diff(range())*<br>finds the total range of input<br>- Example:<br>*sum = var1 + var2*<br>where "*sum*" will be the new column name |
| | ***dplyr*'s own** aggregating functions: | | |
| | Measures of position: | | |
| | *first()* | variable/column name | Returns first element |
| | *last()* | | Returns last element |
| | *nth(…, n)* | | Returns nth element |
| | Measures of count: | | |
| | *n()* | Variable/column name | Returns **number of rows** in data frame or group of observations that *summarise* describes |
| | *n_distinct()* | | Returns **number of *unique* rows** |
| | *count(…, sort = T/F)* | | - Returns the frequency of each value appearing<br>- When used with … *%>% filter(n>1)*, if empty data frame returned means all values appear once (no duplicates)<br>- *sort = T* sorts frequencies/counts in descending order |
| | Turning a **logical test** into an **aggregating function** with: | | |
| | *sum()* | | Returns *total number* of observations/rows which fulfill logical test |
| | *mean()* | | Returns *proportion* of observations that fulfill the logical test |
| ***rename()*** | Renames column's names<br>(Especially when original column names are too vague/not suitable) | | |
| Arguments<br>*(df, newCol1 = oldCol1, newCol2 = oldCol2, …)* | *data set* | object | |
| | *new column names* | expression | E.g. *station = V1* |

| **Relational data** |
|---|
| Multiple tables of data where relations are always defined between a pair of tables<br>(which can be of different dimensions/number of variables) |

<table>
<tr><td>Rough guide to working with relational data</td><td colspan="2">1) Identify the primary keys in each table<br>2) Check that none of the variables in the primary key are missing (Good to sketch out table)<br>3) Check that foreign keys match primary keys in another table</td></tr>
<tr><td rowspan="3">Keys</td><td colspan="2">- Variables that connect each pair of tables<br>- A key is a variable (or a set of variables) that uniquely identifies an observation unit<br>- A variable can be both a primary and a foreign key at the same time<br>- Sometimes, the best identifier of an observation is still not unique<br>- Once you have identified the keys for your tables, it is good to double-check if they are indeed unique<br>Verifying uniqueness of keys:<br>- For example, when …<i>df %>% count(varX) %>% filter(n>1)</i> returns an empty data frame, it means the variable <i>varX</i> is a unique identifier of each observation/row<br>- If not empty, then there may be a need to spread out the data yourself or add an ID to each observation (i.e. numbering the rows from 1 to …)</td></tr>
<tr><td>Primary keys</td><td>Foreign keys</td></tr>
<tr><td>Uniquely identifies an observation in its <b>own</b> table</td><td>Uniquely identifies an observation in <b>another</b> table</td></tr>
<tr><td>Relations</td><td colspan="2">- A primary key and the corresponding foreign key form a relation<br>- Relations are typically one-to-many</td></tr>
<tr><td rowspan="3"><i>Tribble</i></td><td colspan="2">- A simple, stripped down data frame<br>- Way of data entry is specified <i>row-wise</i><br>For example,</td></tr>
<tr><td><i>x <- tribble(<br>~key , ~val_x ,<br>1 , “x1” ,<br>2 , “x2” ,<br>3 , “x3”<br>)</i></td><td><i>y <- tribble(<br>~key , ~val_x ,<br>1 , “y1” ,<br>2 , “y2” ,<br>4 , “y3”<br>)</i></td></tr>
<tr><td colspan="2">(Draw here)</td></tr>
<tr><td><b>Joins</b></td><td colspan="2">- A way of connecting each row in <i>x</i> to 0, 1 or more rows in <i>y</i><br>- From <i>dplyr</i> package<br>(Draw here)</td></tr>
<tr><td rowspan="4">Mutating joins</td><td colspan="2">Add new <b>variables</b> to a data frame from matching observations in another</td></tr>
<tr><td><b><i>inner_join</i></b><i>(x, y, by = “<var/key name>”, …)</i></td><td>- Matches pairs of observations whenever their keys are equal/ keeps observations that appear in both tables<br>- Unmatched rows are dropped<br>- If unspecified, R looks for column names and merges by those</td></tr>
<tr><td colspan="2"><b>Outer joins</b></td></tr>
<tr><td colspan="2">- Keeps observations that appear in at least one of the tables<br>- Add variables to our existing data frame from another table</td></tr>
</table>

| | | |
|---|---|---|
| | - If one table has duplicate keys, then the matching row (the value corresponding to the key) is duplicated as well<br>*("For every row in x table, look for matching rows in y table – not caring if duplicates happen or not")*<br>- If both tables have duplicate keys, then the cartesian product of keys is created<br>- R puts a missing value (*NA*) for tables without the corresponding key value | |
| | ***left_join****(x, y,*<br>*by = "<var/key name>")*<br>*OR*<br>***left_join****(x, y,*<br>*by = c("<varA>" = "<varB>"))*<br>*OR*<br>***left_join****(x, y)* | - Keeps all the observations in *x*<br>- Most common join<br>- Using a character vector as the *by* argument can limit the number of variables used to match observations (i.e. match variable *A* in table *x* to variable *B* in table *y*)<br>- The default of leaving *by* argument empty lets the function use **all** the **variables** that appear in **both** tables |
| | ***right_join****(x, y, by = …)* | - Keeps all the observations in y |
| | ***full_join****(x, y, by = …)* | - Keeps all the observations in x, and all the observations in *y* |
| Filtering joins | Filter observations/rows from one data frame based on whether they **match** an observation in the other table (like *filter()* based on whether keys are matching) | |
| | ***semi_join****(x, y)* | - **Keeps** all observations in *x* that have a match in *y*<br>- If there are duplicate keys in *x*, then all those rows are kept |
| | ***anti_join****(x, y)* | - **Drops** all observations in *x* that have a match in *y*<br>- Useful for looking for mismatches ** |
| **Set operations** | Treat observations as if they were set elements | |
| | ***intersect****(x, y)* | Returns only observations in **both** *x* and *y* |
| | ***union****(x, y)* | Returns **unique** observations in *x* and *y* |
| | ***setdiff****(x, y)* | Returns observations in *x* and but **not** in **y** |

| **Tidy data** | | |
|---|---|---|
| A consistent/standard way of organising/structuring data | | |
| Definition | Requires that:<br>1. Every ***variable*** forms a column<br>2. Every ***observation*** forms a row<br>3. Each type of observational unit forms a table<br>(Multiple tables are fine – relational data) | Note:<br>- A dataset is a collection of **values**<br>- Every value belongs to a ***variable*** and an ***observation***<br>- *Variable*: contains all values that measure the same underlying attribute across units<br>(E.g. height, temperature, duration)<br>- *Observation*: contains all values measured on the same unit across all attributes<br>(E.g. a person, a day) |
| Ordering variables<br>(General guideline) | Good ordering of variables makes it easier to scan raw values:<br>1) **Fixed variables** – those that describe the experimental design (typically known in advance)<br>- Comes first<br>2) **Measured variables** – what we measure in the study (what is unknown prior to the experiment)<br>- Comes last or at the most RHS column | |
| Messy data | Data can be untidy in many different ways, but 2 most common ones are:<br>1) Column headers are values, not actually variable names<br>(i.e. one variable might be spread across multiple columns) | |

| | | | |
|---|---|---|---|
| | - E.g. *year: 2010, 2011, …*<br>- Sometimes, some variables are stored across columns *and* another across rows (then there is a need to *gather()* in both dimensions)<br>- Solve using *gather()*<br>2) Multiple variables are stored in one column (i.e. a single observation being scattered across multiple rows)<br>- OR i.e. two variables in one column<br>- E.g. variables *cases* and *population* under column *type* (multiple rows where *type* column contains two different measurements on each unit *country* in the *year*)<br>- Solve using *spread()* | | |
| ***gather()*** | Gathers columns into a new pair of variables<br>*(Narrower and taller)* | | |
| Arguments | *<set of columns>* | vector | - The set of columns that represent values, not variables<br>- E.g. Columns 1999 and 2000, use *`1999`:`2000`*<br>- Backticks are used to refer to names/combinations of symbols that are otherwise reserved or illegal (sometimes due to R adding a letter in front of column name (e.g. *x1999, x2000, …*) when reading csv) |
| | *key = …* | string/variable name<br>*(works with and without quotes)* | The name of the variable/column that will be created whose values form the **column names** now |
| | *value = …* | | The name of the variable/column that will created whose **values** are currently residing in the **cells** of the dataset (e.g *value = "cases"*) |
| ***spread()*** | Spread columns out into a single row for each observation unit<br>*(Shorter and wider)* | | |
| Arguments | *key = …* | variable name<br>*(without quotes)* | The column the currently contains variable names (variables to spread out) |
| | *value = …* | | The column that currently contains values from multiple variables<br>(column associated with *key*) |
| ***separate()*** | Pulls apart one column into multiple columns, by splitting wherever a separator character appears | | |
| Arguments | *df* | object | |
| | *col* | variable name | - The column to separate<br>- E.g. *rate* where values are of the form:<br>*numCases/numPopulation* |
| | *into = …* | char vector | The names of the new variables |
| | *convert* | *T/F* | Convert columns to integer/numeric/logical since output columns by default, are characters |
| ***unite()*** | Combines multiple columns into one, using a separator character<br>(Reflection of *separate()*) | | |
| Arguments | *df* | object | |
| | *col* | new variable/column name<br>*(without quotes)* | |
| | *<columns to combine>* | variable/column names to combine | E.g. *… year:day. -month …* where *month* is excluded |

|  |  |  |  |
|---|---|---|---|
|  | *sep* | string | E.g. “/” |
|  | *remove* | *T/F* |  |