R Primer

Takwanisa Machemedze

2022-08-12

Contents

Learning objectives

This one day course provides an introduction to R for beginners. Course participants will learn the following:

- Familiarise with the R/RStudio interface,
- Able to import various data formats into the R Workspace,
- Perform common data manipulation/transformation task using base R and dplyr,
- Perform basic descriptive data analysis,
- Plot graphics/visualisation using base R and ggplot2 functions

Introduction

R and RStudio

R is a free open source environment for statistical computing and graphics. It runs on most platforms - UNIX platforms, Windows and Mac OS. R can be download at https://cran.r-project.org/. One can use R alone or with an interface like Rstudio - an IDE (integrated Development Environment) which runs on top of R. RStudio provides a rich user interface when compared to R and makes R more accessible and easy to use. RStudio can be downloaded at https://www.rstudio.com/. After installing both R and RStudio, you only need to run RStudio.

This document, A (very) short introduction to R, offers a very good introduction - from installation, getting you started and more.

Why use R?

Pros

- R is open source and free under GNU General Public License.
- It is available on all common platforms.
- It is supported by an active community of users across most scientific disciplines - academia and industry.
  - There are many user contributed packages by experts from various scientific disciplines.
  - New technology and ideas are often first implemented in R and readily available.
- It is a flexible programming language.
  - It can handle most statistical computations, GIS mapping, building interactive web applications, e.t.c.
- Easy integration with other programs.

Cons

- R is renowned for a steep learning curve.
- All objects are stored in the computer memory and can use all the available memory.
- Can be slower than compiled languages.
- Easy to make mistakes, difficult to find the sources of mistakes.
- While there is community support, there is no one to complain if something doesn't work.

Some useful R Resources

There are many useful online resources for learning R and below is a list of some of these in the form of manuals, books, tutorials, presentations and blogs in no particular order.

- The R manuals (https://cran.r-project.org)
- Quick-R homepage - statmethods.net - - a good place to start learning R and an easily accessible reference.
- UCLA website
- An Introduction to R (Venables and Smith 2014)
- R for Data Science, (Wickham and Grolemund 2016)
- Cookbook for R
- CRAN Task Views - links to packages grouped by topic.
- And many more materials, some acknowledged at the end.

Getting Started with R/RStudio

RStudio interface

When you open RStudio for the first time, three panels will appear. However, there are four standard panels that shows when you open, for example, an R Script, as shown below.

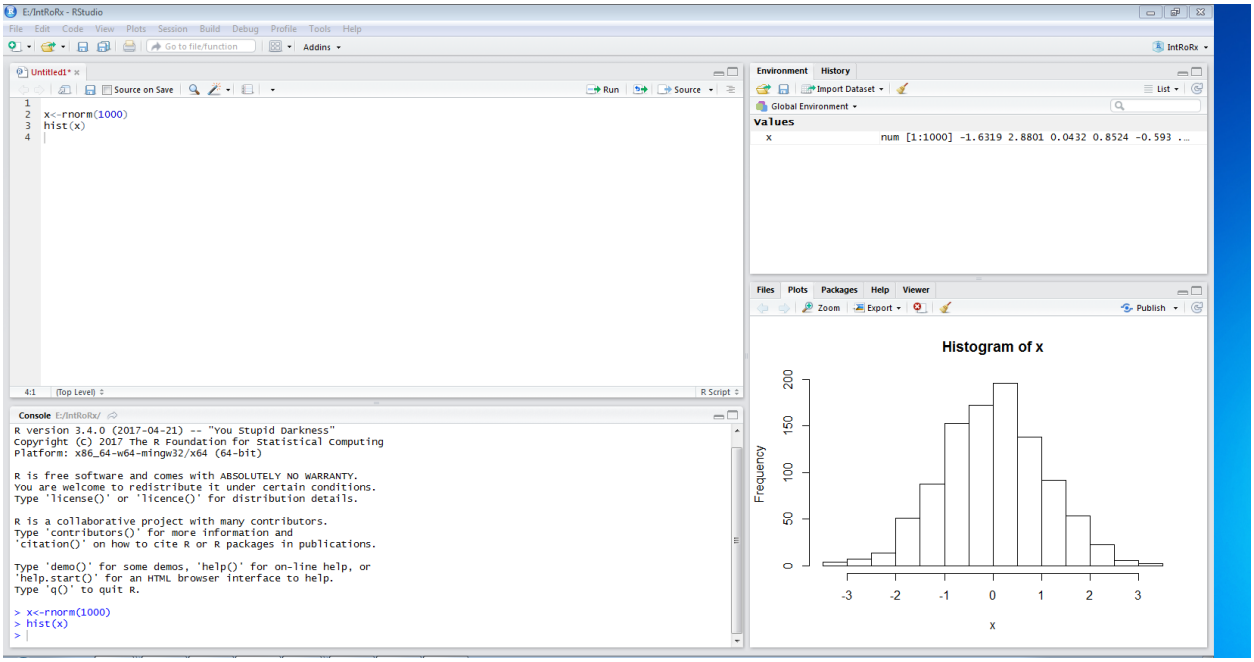


Figure 1: RStudio Layout

Basic layout

Four panels and several tabs

- **Top-left** - the R script where you type source code, like the script editor in Stata.
- **Bottom -left** - is the console, where you see what R is doing. Commands can be entered directly or sent from the script pane (e.g., control/command + enter)
- **Upper- right** - has the following tabs:
  - *Environment*: Lists objects in the workspace (e.g., data you've created or imported)
  - *History*: list of commands run on console.
  - *Tutorial*: for integrated tutorials powered by the `learnr` package.
- **Bottom right** has the following tabs:
  - *Files*: quick access to files in your working directory
  - *Plots*: View current and previous plots you've created
  - *Packages*: Helps with checking, loading and installing packages
  - *Help*: Show built-in help and allow searching for help
  - *Viewer*: displays local web content such as web graphics generated by some packages.

Managing the workspace

Commands

Most base R commands adopt the following syntax:

`> command(argument1, argument2, ...,)`

Any base R command needs to be followed by brackets. Using RStudio, you can either type your commands at the console and press enter or type in the script editor (`Menu -> New File -> R Script`) and press the Run button to the top right corner of the same window. Alternatively, selected/highlighted parts of an R script file can be run by pressing `Ctrl + Enter` and the whole script can be run by pressing `Ctrl + Alt + R`.

Set working directory

One can check or change the working directory in Rstudio (Session) using the following commands:

- Check current working directory

```
getwd()
```

- Set another working directory

```
setwd("path/to/directory")
```

Install (and load) libraries

Often times you will find that you may need to use functions from other user-written packages or libraries. Since these packages do not come installed in base R, you need to do the additional installation by yourself. To install a package called `lattice`, type the following on your R console (and `Enter`) or script file (and `Run`):

```
install.packages("lattice")
```

Developers of these packages usually update them after introducing new functions or even changing names of certain functionality and therefore one might need to keep track of these changes or be aware that it can happen.

You also need to load the package/library into the R workspace before you can use functions from that library as follows:

```
library(lattice)
```

You need to load the library each time you need to use it in a new R session.

Comments in R

`#` denotes a comment in R, e.g.

```
# Check current working directory
getwd()
```

Anything after the # is not evaluated and ignored in R

Getting help

There are several functions for getting help in R as described here, but I have mostly used the following:

- `help(solve)` or `?solve` - to get help for command `solve`, type:
  - this is useful only if you already know the name of the function that you wish to use.
- `help.search("solve")` and `??solve` - scans the documentation for packages installed in your library for commands which could be related to string solve
- `help.start()` - Start the hypertext (currently HTML) version of R's online documentation.
- For tricky questions, error messages and other issues, use Google (include "in R" to the end of your query).
- StackOverflow - great resource with many questions for many specific packages in R and a rating system for answers

Operators in R

Basic arithmetic operators

All basic mathematical operations are the same in R as it is in other languages (using the `+`, `-`, `*`, and `/` operators)

```
2 + 3 # Addition
```

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

```
7 - 4 # Subtraction
```

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

```
3 + 5 * 2 # Multiplication
```

```
## [1] 13
```

```
(3 + 5) * 2 # Multiplication with brackets
```

```
## [1] 16
```

```
#Maths ops follow the normal order of operations.
7/3 # Division
```

```
## [1] 2.333333
```

```r
5^2 # Power
```

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

We can also use modulo operators (integer division & remainder).

For example.

```r
150 %/% 60 ## E.g. whole hours in 150 minutes
```

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

```r
150 %% 60 ## Minutes left over
```

```
## [1] 30
```

Functions like `exp()`, `log()` and `sqrt()` also exist.

Assignment operator

`<-` is the assignment operator, it declares something is something else. For example, we can assign `x` to be 3 as follows:

```r
x <- 3
x
```

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

Sequence operator

`:` is the sequence operator. To get the sequence of numbers/integers from 1 to 10, type the following:

```r
a <- 1:10 # it increments by one
a
```

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

Managing the workspace (more functions)

As we creating more objects in the workspace, we also need to be able to manage and manipulate the 'workspace'. All the R objects that we have created are stored in the memory of the computer and sometimes the unnecessary objects in the workspace can use a significant amount of the memory. R makes organizing the workspace easy. Lets create a few more objects and show how to remove some of these from memory:

```r
z <- 10
```

We can list or ask R to display all the objects in memory using the `ls()` function:

```
ls()  #list all variables
```

```
## [1] "a" "x" "z"
```

Assuming that we no longer need object `x` in the workspace, we can remove it from memory using the `rm()` function:

```
rm(x)  # delete a variable
```

We can list again all the objects in memory and confirm that `x` has been removed.

```
ls()
```

```
## [1] "a" "z"
```

We are now left with objects `a` and `z`.

We can also remove specific objects, `a` and `z` by typing the following:

```
rm(x,y)
```

```
## Warning in rm(x, y): object 'x' not found
```

```
## Warning in rm(x, y): object 'y' not found
```

- To remove everything in the workspace

```
rm(list = ls())
```

Some language features

It might be helpful to be aware of the following R features:

- R is inconsistent in it's naming conventions
  - Some functions are `camelCase`; others are `dot.separated`; others `use_underscores`.
- Function results are stored in a variety of ways across function implementations.
- R has multiple graphics packages that different functions use for default plot construction (base, grid, lattice, and ggplot2)
- R has multiple packages and functions to perform the same tasks.
- Be flexible and be aware of R's flexibility

Objects

Everything in R is an object and even including functions. The next section introduces some common types of objects. But before we move to the next section, we do the following practice question on assignments and operators.

Practice - assignment and operators

1. Assign the following variables for an individual whose `height` is 175cm and `weight` is 80kg.

```
height <-

weight <-
```

2. Calculate their BMI assuming the variables defined above and the BMI formula below:

$$BMI = \frac{weight(kg)}{height(m)^2}$$

```
bmi <-
```

Data types and structures

Data types

- R has five basic or "atomic" classes of objects:
  - **logical** (e.g., TRUE, FALSE)
  - **integer** (e.g., 2L)
  - **numeric** (real or decimal) (e.g, 2, 2.0, π)
  - **complex** (e.g, 1 + 0i, 1 + 5i)
  - **text** or **character** (e.g, "this text", "any text")

Data structures

- R also has many data structures and these include:
  - vector
  - list
  - matrix
  - data frame
  - factors
  - tables
- We will go through some of these

Vectors

Vector

- A vector is the most common and basic data structure in `R`
- Vectors can be of two types:
  - atomic vectors
  - lists

Atomic vectors

- can be a vector of characters, logical, integers or numeric.
- The `c()` function can be used to create vectors of objects.

```
#logical
v_log <- c(TRUE, FALSE, FALSE, TRUE)
v_log
```

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

```
#integer
v_int <- 1:4
v_int
```

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

```
#numeric double precision
v_doub <- 1:4 * 3.14
v_doub
```

```
## [1]  3.14  6.28  9.42 12.56
```

```
#character
(v_char <- c("a","b","c","d"))
```

```
## [1] "a" "b" "c" "d"
```

```
#note -  I enclosed in brackets to print on screen
```

- Each atomic vector has the same type of elements

Check vector type

- use `is.*()` functions to confirm vector type

```
is.numeric(v_doub)
```

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

```
is.integer(v_doub)
```

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

- It is also possible to ask R using the class function - `?class`

```
class(v_doub)
```

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

Explicit coercion

- It is easy to convert from one vector type to another type.
- For explicit coercion, use the `as.*()` functions.

```
v_log
```

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

```
as.integer(v_log)
```

```
## [1] 1 0 0 1
```

```
v_int
```

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

```
as.numeric(v_int)
```

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

```
v_doub
```

```
## [1]  3.14  6.28  9.42 12.56
```

```
as.character(v_doub)
```

```
## [1] "3.14"  "6.28"  "9.42"  "12.56"
```

```
as.character(as.numeric(as.integer(v_log)))
```

```
## [1] "1" "0" "0" "1"
```

Mixing objects

- What happens when a vector consists of multiple data types?

```
x <- c(1.7, "a") ## numeric + character
class(x)
```

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

```r
y <- c(TRUE, 2) ## logical + numeric
class(y)
```

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

```r
z <- c("a", TRUE) ## character + logical
class(z)
```

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

- Coercion occurs so that every element in the vector is of the same class.
- Hierarchy of types: the more primitive ones silently convert to those higher up.
- The order is:

    1. logical
    2. integer
    3. double
    4. character

Some special numbers

- There is also a special number `Inf` which represents infinity;
    - e.g. 1/0

```r
1/0
```

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

- The value `NaN` represents an undefined value ("not a number").
    - e.g. `0/0`

```r
0/0
```

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

Index a vector

- to select specific elements or atoms.
- elements of a vector have an index or position number.
- index using square brackets - `object[index]`.
- there are several ways to express which elements you want.

Logical vector

- logical vector: keep elements of `x` for which the index is `TRUE` and drop elements for which the index is `FALSE`.
- remember vectors created previously.

```
v_log <- c(TRUE, FALSE, FALSE, TRUE) #logical
v_int <- 1:4 #integer
v_doub <- 1:4 * 3.14 #numeric double precision
v_char <- c("a","b","c","d") #character
```

- filter elements 3 and 4 from the character vector `v_char`.

```
v_char[c(FALSE, FALSE, TRUE, TRUE)]
```

```
## [1] "c" "d"
```

- filter character vector `v_char` based on the logical vector `v_log`.

```
v_char[v_log]
```

```
## [1] "a" "d"
```

Positive integer vector: the elements specified in the index are kept, e.g., the 2nd and 3rd.

```
v_doub[2:3]
```

```
## [1] 6.28 9.42
```

Negative integer vector: the elements specified in the index are dropped.

```
v_char[-4]
```

```
## [1] "a" "b" "c"
```

Factors

- Factors are used to represent categorical data.
- Factors can be unordered or ordered.
- Factors are stored as integers where each integer has a label associated with it.
- When a character vector is converted to a factor, by default, R sorts levels in alphabetical order.
- A factor is a very special and sometimes frustrating data type in R
- Factors can be created using the `factor()` command. Let us create a character vector, `v_char`, and convert to a factor, `v_fac`:

```r
v_char <- c("Medium","High","Low") # character vector
v_char
```

```
## [1] "Medium" "High"   "Low"
```

```r
v_fac <- factor(v_char) # factor variable
v_fac
```

```
## [1] Medium High   Low
## Levels: High Low Medium
```

- Using the `levels` command, we can see that levels of the above factor are in alphabetical order of "High", "Low", "Medium".

```r
levels(v_fac)
```

```
## [1] "High"   "Low"    "Medium"
```

- Factors are of class `factor`.

```r
class(v_fac)
```

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

- What is happening under the hood?

```r
unclass(v_fac) # In alphabetical order: "Medium=>3,High=>1,Low=>2
```

```
## [1] 3 1 2
## attr(,"levels")
## [1] "High"   "Low"    "Medium"
```

- What if we don't want the default ordering? For example, if you want a lower value for the baseline level in regression, plot ordering, e.t.c.

Ordering the factor

- It may be more meaningful to maintain the order "Low", "Medium", "High"
- The order of the levels can be set using the `levels` argument to `factor()`.
- This can be important in linear modelling because the first level is usually used as the baseline level (another option is to use the function `relevel()`).

```r
v_fac_ord <- factor(v_char, levels = c("Low", "Medium", "High"), ordered = TRUE)
#v_fac_ord <- ordered(v_char, levels = c("Low", "Medium", "High"))
v_fac_ord
```

```
## [1] Medium High   Low
## Levels: Low < Medium < High
```

```
unclass(v_fac_ord)
```

```
## [1] 2 3 1
## attr(,"levels")
## [1] "Low"    "Medium" "High"
```

- Factors can also be created from numeric vectors.
- Assume we have a vector of `1`s and `0`s where 1 represents the presence of some outcome and 0 otherwise. The code below shows how to create a labeled factor from the vector.

```
v_indicator <- c(1, 0, 1, 1, 0, 0,1)

v_fct <- factor(v_indicator,
                levels = c(0, 1),
                labels = c('No', 'Yes'))

summary(v_fct)
```

```
##  No Yes
##   3   4
```

Converting factors

- Converting from a factor to a number can cause problems:

```
f <- factor(c(1, -9, 2))
as.numeric(f)
```

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

- It returns factor levels and this is not what we wanted (and there is no warning).
- The recommended way is to use the integer vector to index the factor levels:

```
levels(f)[f]
```

```
## [1] "1"  "-9" "2"
```

- to get the numeric values

```
as.numeric(levels(f)[f])
```

```
## [1]  1 -9  2
```

- Or less effecient

```
as.numeric(as.character(f))
```

```
## [1]  1 -9  2
```

Dropping fevels

- When you want to drop unused factor levels, use `droplevels()`:
- Lets say you have no observations in one level - "Medium"

```
v_fac_ord <- v_fac_ord[v_fac_ord != "Medium"]
summary(v_fac_ord)
```

```
##    Low Medium   High
##      1      0      1
```

- You can drop that level if desired - e.g. if you want to plot without dropping the level, the category will show up.

```
f_var_new<-droplevels(v_fac_ord)
summary(f_var_new)
```

```
##  Low High
##    1    1
```

Matrices

- **Matrices** are vectors with dimensions - number of rows and columns.
- All rows and columns are of the same length and data type, e.g.,

```
m <- matrix(1:12, nrow = 3, ncol = 4)
m
```

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

```
dim(m)
```

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

```
#Same as:
attributes(m)
```

```
## $dim
## [1] 3 4
```

- In R, matrices are by default constructed column-wise.
- This can be changed by specifying the `byrow` option.

Lists

- **lists** are arbitrary combinations of disparate object types in R.
- The objects do not have to be of the same type or same element or same dimensions.

```
my_list <- list(v_char = c('A', 'B'),
                v_num = c(1, 3.5, 8, 15.91),
                v_comp = 1 +4i,
                v_list = list(c(1L, 2L), c(TRUE, FALSE, FALSE)))

my_list
```

```
## $v_char
## [1] "A" "B"
##
## $v_num
## [1]  1.00  3.50  8.00 15.91
##
## $v_comp
## [1] 1+4i
##
## $v_list
## $v_list[[1]]
## [1] 1 2
##
## $v_list[[2]]
## [1]  TRUE FALSE FALSE
```

Data Frames

- Data frames are used to store tabular data.
- They are represented as a special type of list where every element of the list has to have the same length.
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.
- Unlike matrices, data frames can store different classes of objects in each column (just like lists);
  - Matrices must have every element be the same class
- Data frames also have a special attribute called `row.names`
- Data frames can be created using the `data.frame()` function.
- Can also import external tabular text data using `read.table()` or `read.csv()` into a data frame.
- Can be converted to a matrix by calling `data.matrix()` or `as.matrix()`

Exploring data frames

Here, we demonstrate some functions for handling data frames using an in-built data set called `iris`. We will introduce functions for importing your own data later on, but for now we will call the `iris` data frame using the `data()` function as follows:

```
data(iris)
```

We start by demonstrating functions for showing a few of the observations in a data frame.

The `head()` function shows the first few rows of the data frame. The default number of rows is 6, but this can be changed.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The `tail()` - function shows the last few rows of the data frame. Again, the default is 6 rows.

```
tail(iris)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

`dim()` - see dimensions, i.e., number of rows and columns

```
dim(iris)
```

```
## [1] 150   5
```

`nrow()` - number of rows

```
nrow(iris)
```

```
## [1] 150
```

`ncol()` - number of columns

```
ncol(iris)
```

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

`str()` - structure of each column

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

`names()` - will list column names for a `data.frame` (or any object).

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Handling data in R

Missing values

- denoted by `NA` and/or `NaN` for undefined mathematical operations.
- `NA` values have a class. So you can have both an integer `NA` and a missing character `NA`.
- `NaN` is also `NA`. But not the other way around.

Example 1

- Example `x` vector

```
x <- c(1, 2, NA, 10, 3)
```

- Check elements of `x` that are `NA`

```
is.na(x) # returns logical
```

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

- Check elements of `x` that are `NAN`

```
is.nan(x)
```

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

- All `FALSE`

Example 2

- Another example vector

```
x <- c(1, 2, NaN, NA, 4)
```

- Check elements of `x` that are `NA`

```
is.na(x)
```

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

- Check elements of `x` that are `NAN`

```
is.nan(x)
```

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

Relational operators

- The comparison operators `<`, `>`, `<=`, `>=`, `==`, and `!=` are used to compare values across vectors.
- Consider the following vectors:

```
big <- c(9, 12, 15, 25)
small <- c(9, 3, 4, 2)
```

- Check cases where big is greater than small

```
big > small
```

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

- This returns a vector of logical values
- How about big equal to small?

```
big = small
```

- this reassigns big to small, and this not what we wanted. We wanted cases where big is equal to small

```
print(big)
```

```
## [1] 9 3 4 2
```

```
print(small)
```

```
## [1] 9 3 4 2
```

- Safer to assign using `<-` than comparison operators, `=` or `==`.
- The best way to evaluate these objects is to use brackets `[]` to avoid confusion.
- Lets first correct the `big` vector.

```
big <- c(9, 12, 15, 25)
```

- To get elements of big such that big is equal to small we do the following:

```
big[big == small]
```

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

- The command returns values of "big" where the logical vector created inside the square bracket is true.
- More examples ...

```
big[big > small]
```

```
## [1] 12 15 25
```

```
big[big < small]  # Returns an empty set
```

```
## numeric(0)
```

Logical operators

- Construct complex condition using logical operators such as, (AND) `&`, (OR) `|` and (NOT) `!`.
- Referring back to vectors `small` and `big`

AND

- The `&` operator represents the logical **AND** operation. It returns `TRUE` if both conditions on the left and right of `&` are TRUE.

```
small>=9 & big<14
```

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

- To get elements of `big` that match the `&` condition

```
big[small>=9 & big<14]
```

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

OR

The | operator represents the logical **OR** operation. It returns TRUE if either one of the conditions on the left and right of | are TRUE.

```
#|
small>=9 | big<14
```

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

- To get elements of big that match the | condition

```
big[small>=9 | big<14]
```

```
## [1]  9 12
```

NOT

- Assume we have a vector x as follows:

```
x <- c("a", NA, 4, 9, 8.7)
!is.na(x)  # Returns TRUE for non-NA
```

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

```
class(x)
```

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

- Using the logical vector to subset x

```
x1 <- x[!is.na(x)]
x1
```

```
## [1] "a"   "4"   "9"   "8.7"
```

```
class(x1)
```

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

Importing and exporting data

Data sources

The data that we usually want to analyze in R come in different file formats and from different sources that includes:

- Built in Data (+ data in packages)
- Formatted text: .csv, .txt, .xls
- Specialised software: e.g. SPSS, SAS, Stata
- Databases
- Web Scraping

Built in Data

Many packages in R have built in data. Authors of the packages use this data to demonstrate functions in their packages. This data can be a great resource for demonstrating other use cases. To check what data you have in R, run the following command:

```
data()
```

An **"R data sets"** window will pop up on the top left pane and display a list of the data as well as what packages the data is in.

It is also possible to call data from a specific package, e.g. data from the `datasets` package.

```
data(package="datasets")
```

To load a specific built-in dataset, e.g. the `iris` data, type the following:

```
data(iris)
```

Native R data formats

R has two native data formats - `.RData` and `RDS`. These formats are used when R objects are saved for later use. `RData` is used to save multiple R objects, while RDS is used to save a single R object.

We can load the `demo_data.RData` data set in the `data` subfolder.

```
load("./data/demo_data.RData")
str(my.df)
```

```
## 'data.frame':    18 obs. of  7 variables:
##  $ hhid: int  1 1 1 1 1 2 2 2 3 3 ...
##  $ pid : int  1 2 3 4 5 1 4 5 1 2 ...
##  $ age : int  72 88 76 89 46 17 73 99 3 15 ...
##  $ sex : int  1 2 2 1 1 2 2 1 1 2 ...
##  $ grp : chr  "C" "C" "A" "A" ...
##  $ inc1: num  61.9 96.1 65.5 51 15 ...
##  $ inc2: num  43.6 69.4 97.9 69.5 61.7 ...
```

We can save this data set, `my.df` and the `iris` dataset to `demo_data1.RData` as follows:

```
# save specific objects to a file
save(my.df, iris, file = "./data/demo_data1.RData")

# save the whole workspace to the file .RData in the current working directory (cwd)
#save.image()
```

Text delimited format

Text delimited file format is one of the common formats in which data is shared. The most common ones are comma separated (csv) or a tab delimited file. There are several functions that can read these data into R.

Base R functions

- We can import `.csv`/`.txt` data using the `read.table` function from the base R `utils` package.
- The `read.table` function is the general function for reading text formatted data. It also has the following variants:
  - `read.csv` - for reading standard CSV
  - `read.csv2` - for reading fields separated by a semi-colon
  - `read.delim` - for tab delimited files where by default, point (".") is used as decimal point
  - `read.delim2` - for tab delimited files where by default, comma (",") is used as decimal point
- The default arguments of the `read.csv` function are as follows:

```
function (file, header = TRUE, sep = ",", quote = "\"", dec = ".", fill = TRUE,
    comment.char = "", ...)
```

- `file` - name of file to read/import
- `header` - a logical value indicating whether the file contains the names of the variables as its first line. Default if `TRUE`
- `sep` - the field separator character. default is ","
- `quote` - the set of quoting characters
- `dec` - the character used in the file for decimal points
- Here we assume these defaults and only specify the `file` path as follows:

```
my.df <- read.csv(file = "./data/demo_data.csv")
head(my.df)
```

```
##   hhid pid age sex grp  inc1  inc2
## 1    1   1  72   1   C 61.89 43.56
## 2    1   2  88   2   C 96.14 69.38
## 3    1   3  76   2   A 65.50 97.94
## 4    1   4  89   1   A 51.03 69.46
## 5    1   5  46   1   C 15.01 61.71
## 6    2   1  17   2   B 87.04 92.20
```

- The CSV file is in the `data` sub directory of the parent working directory
- We only need to use the relative path when referring to a subdirectory of the working directory
- The corresponding function for saving to CSV is `write.csv()`.

```
write.csv(my.df, "./data/demo_data_new.csv")
```

readr package functions

- The `readr` package is part of the `tidyverse` and contains a number of functions for importing data into R.
- It has its own version of the base R functions described above - `read_csv`, `read_delim`
- `read.csv` and `read_csv` behaves in similar fashion, with a few notable differences:
    1. `read.csv()` stores data as a `data.frame`, where as `read_csv()` stores data as a `tibble`. **Tibbles** are data frames, but they tweak some older behaviours to make life a little easier.
    2. `read.csv()` coerces column names with spaces and/or special characters to different names
    3. The `read_csv()` function loads faster and offers some advanced features compared to `read.csv`
- To use functions within packages, we first need to load the package/library. To load the `readr` package:

```
library(readr)
my_df <- read_csv(file = "./data/demo_data.csv")
head(my_df)
```

```
## # A tibble: 6 x 7
##    hhid   pid   age   sex grp    inc1  inc2
##   <dbl> <dbl> <dbl> <dbl> <chr> <dbl> <dbl>
## 1     1     1    72     1 C      61.9  43.6
## 2     1     2    88     2 C      96.1  69.4
## 3     1     3    76     2 A      65.5  97.9
## 4     1     4    89     1 A      51.0  69.5
## 5     1     5    46     1 C      15.0  61.7
## 6     2     1    17     2 B      87.0  92.2
```

- The corresponding function for saving to CSV is `write_csv()`.

```
write_csv(my_df, "./data/demo_data_new.csv")
```

data.table package

- There is also the `data.table` package that has a function named `fread()`. This function works faster than both `read.csv()` and `read_csv()`.

MS Excel format

- There are several packages for importing data from MS Excel such as **gdata**, **RODBC**, **XLConnect**, **RExcel**, **readxl**.
- The **readxl** package, a member of the `tidyverse`, fulfills most of basic requirements for importing Excel files into R.
- The **readxl** package contains two import functions - `read_excel()` and `excel_sheets()`. `read_excel()` allows you to import an excel file into R and `excel_sheets()` allows you to read different sheets present within the excel book.
- The **readxl** package can import both the old formatted ".xls" files and the latest formatted ".xlsx".

Specialiased statistical software - SAS, SPSS, Stata

haven package

The `haven` package has functions for importing data files from various statistical software systems. This package support SAS, Stata and SPSS software. We can use the following functions to import different file format into R:

- STATA: `read_dta()` or `read_stata()` - the two functions are identical

```
library(haven)
#import
stata_df <- read_dta("./data/demo_data.dta")
#export
write_dta(stata_df, "./data/demo_data.dta")
```

- SAS: `read_sas()`

```
#import
sas_df <- read_sas("./data/demo_data.sas7bdat")
#export
write_sas(sas_df, "./data/demo_data.sas7bdat")
```

- SPSS: `read_sav()` or `read_por()`.

```
#import
spss_df <- read_sav("./data/demo_data.sav")
#export
write_sav(spss_df, "./data/demo_data.sav")
```

foreign package

There is also the foreign package, an older base R library for importing SAS, Stata and SPSS file formats. The `haven` package introduced above is an improvement to the foreign package. The foreign package functions for importing the various file formats are:

- STATA: `read.dta()`
- SPSS: `read.spss()`
- SAS file format are a bit more complicated to import with the `foreign` package

For the packages introduced above, you can read more from the package websites:

R manuals: R Data Import/Export

readr package page

data.table vignettes

readxl package page

haven package page

You can check more on R packages for data import and other functions from the following blogs:

Garrett Grolemund - Quick list of useful R packages

Sharon Machlis - Great R packages for data import, wrangling, and visualization

Manipulating data

Data frame manipulation with base R functions

Data

Here we are going to import the `demo_data.csv` file, a fictional data that we just introduced above.For now, we are going to stick to base R functions.

```
my_df<-read.csv("./data/demo_data.csv")
```

We are going to make use of some of the functions introduced in the previous sections. We can use the `head()` function to return the first 10 elements or observations

```
head(my_df, n=10L)
```

```
##    hhid pid age sex grp  inc1  inc2
## 1     1   1  72   1   C 61.89 43.56
## 2     1   2  88   2   C 96.14 69.38
## 3     1   3  76   2   A 65.50 97.94
## 4     1   4  89   1   A 51.03 69.46
## 5     1   5  46   1   C 15.01 61.71
## 6     2   1  17   2   B 87.04 92.20
## 7     2   4  73   2   B  1.92 40.09
## 8     2   5  99   1   A 14.45 37.21
## 9     3   1   3   1   C    NA 68.76
## 10    3   2  15   2   A 82.57 50.68
```

Since I created this data, this can be seen as individual level information. Individuals are identified by the `hhid`(Household ID) and `pid`(Person ID). There is also information about their `age`, some grouping variable (`grp`) and two sources of income, `inc1` and `inc2`.

By visual inspection, we can see that `pid` 1 from `hhid` 3 has missing `inc1`. To see the structure of each column or variable we use the `str()` function.

```
str(my_df)
```

```
## 'data.frame':    18 obs. of  7 variables:
##  $ hhid: int  1 1 1 1 1 2 2 2 3 3 ...
##  $ pid : int  1 2 3 4 5 1 4 5 1 2 ...
##  $ age : int  72 88 76 89 46 17 73 99 3 15 ...
##  $ sex : int  1 2 2 1 1 2 2 1 1 2 ...
##  $ grp : chr  "C" "C" "A" "A" ...
##  $ inc1: num  61.9 96.1 65.5 51 15 ...
##  $ inc2: num  43.6 69.4 97.9 69.5 61.7 ...
```

- The `summary()` function returns basic descriptive statistics for each variable from the data frame.

```
summary(my_df)
```

```
##       hhid            pid             age             sex
##  Min.   :1.000   Min.   :1.000   Min.   : 0.00   Min.   :1.000
##  1st Qu.:1.250   1st Qu.:2.000   1st Qu.:39.00   1st Qu.:1.000
##  Median :3.000   Median :3.000   Median :55.00   Median :2.000
##  Mean   :2.611   Mean   :3.111   Mean   :54.17   Mean   :1.556
##  3rd Qu.:3.750   3rd Qu.:4.750   3rd Qu.:75.50   3rd Qu.:2.000
##  Max.   :5.000   Max.   :5.000   Max.   :99.00   Max.   :2.000
##
##      grp                 inc1            inc2
##  Length:18          Min.   : 1.92   Min.   :37.21
##  Class :character   1st Qu.:14.87   1st Qu.:50.45
##  Mode  :character   Median :56.46   Median :69.07
##                     Mean   :51.54   Mean   :68.11
##                     3rd Qu.:81.26   3rd Qu.:82.81
##                     Max.   :96.14   Max.   :98.72
##                     NA's   :2
```

- We use the `names()` function to print names of all columns in the data frame.

```
names(my_df)
```

```
## [1] "hhid" "pid"  "age"  "sex"  "grp"  "inc1" "inc2"
```

- `mean()` function calculates the mean of a vector

```
#remember $
mean(my_df$age)
```

```
## [1] 54.16667
```

- **What happens with the mean for `inc1` which has some missing values**

```
mean(my_df$inc1)
```

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

- R does not calculate the mean (or other summary statistics) when there are `NA`s in the vector

- We need to tell R to remove/ignore `NA`s before performing any computations by specifying the option, `na.rm=TRUE`

```
mean(my_df$inc1, na.rm=TRUE)
```

```
## [1] 51.5375
```

- We do not necessarily need to specify `na.rm=TRUE` if the variable has no `NA`s
- More summary statistics - `sd`, `var`, `min`, `max`, `median`, `range` and `quantile`.

```
sd(my_df$age) #standard deviation
```

```
## [1] 31.56366
```

```
var(my_df$age) #variance
```

```
## [1] 996.2647
```

```
min(my_df$age) #minimum
```

```
## [1] 0
```

```
max(my_df$age) #maximum
```

```
## [1] 99
```

```
median(my_df$age) #median
```

```
## [1] 55
```

```
range(my_df$age) #range
```

```
## [1]  0 99
```

```
quantile(my_df$age) #quantile
```

```
##   0%  25%  50%  75% 100%
##  0.0 39.0 55.0 75.5 99.0
```

Basic subsetting

- Subsetting is a way(s) to keep or delete variables and observations and to take random samples from a dataset.

Selecting (keeping) variables

- A data frame (`df`) is of the form, `df[rows, columns]`
- Therefore, the following specification evaluates if a vector of column names (`mycol`) is contained in the names of the data frame.
- `df[,mycol]`
- However, R thinks column-wise and the following also works since it defaults to evaluating columns
- `df[mycol]`
- We want to select variables `inc1`and `inc1`:

Method 1

- Using a vector of variable names

```
myvars <- c("inc1", "inc2")
newdata <- my_df[myvars]
#newdata <- my_df[c("inc1","inc2")]
head(newdata,5)
```

```
##    inc1  inc2
## 1 61.89 43.56
## 2 96.14 69.38
## 3 65.50 97.94
## 4 51.03 69.46
## 5 15.01 61.71
```

Method 2

- We can also use index position of column names
- For example, to select 3rd and 6th thru 7th variables:

```
newdata <- my_df[c(3,6:7)]
head(newdata,5)
```

```
##   age  inc1  inc2
## 1  72 61.89 43.56
## 2  88 96.14 69.38
## 3  76 65.50 97.94
## 4  89 51.03 69.46
## 5  46 15.01 61.71
```

Excluding (dropping) variables

Method 1

- Excluding columns works as above except that we need to negate(`!`) the condition
- To exclude variables `inc1` and `inc2`

```
myvars <- names(my_df) %in% c("inc1", "inc2")
newdata <- my_df[!myvars]
head(newdata,5)
```

```
##   hhid pid age sex grp
## 1    1   1  72   1   C
## 2    1   2  88   2   C
## 3    1   3  76   2   A
## 4    1   4  89   1   A
## 5    1   5  46   1   C
```

`%in%` is used for value matching. It "returns a logical vector indicating if there is a match or not for its left operand." (See `help('%in%')`)

Method 2

- We can also use the column index position and negate (`-`)
- Exclude 6th and 7th variable

```
newdata <- my_df[c(-6,-7)]
head(newdata, n = 5)
```

```
##   hhid pid age sex grp
## 1    1   1  72   1   C
## 2    1   2  88   2   C
## 3    1   3  76   2   A
## 4    1   4  89   1   A
## 5    1   5  46   1   C
```

Method 3

- We can also use the `NULL` special function as follows:

```
#my_df$inc1 <- NULL
```

Selecting observations

square brackets

- We can select observations by the index of row numbers
- To select first 5 observations

```
newdata <- my_df[1:5,]
newdata
```

```
##   hhid pid age sex grp  inc1  inc2
## 1    1   1  72   1   C 61.89 43.56
## 2    1   2  88   2   C 96.14 69.38
## 3    1   3  76   2   A 65.50 97.94
## 4    1   4  89   1   A 51.03 69.46
## 5    1   5  46   1   C 15.01 61.71
```

A defined condition

- For example, to select rows where `grp ==‘C’` and `age` greater than 65

```
newdata <- my_df[my_df$grp=='C'& my_df$age > 65, ]
head(newdata,5)
```

```
##    hhid pid age sex grp  inc1  inc2
## 1     1   1  72   1   C 61.89 43.56
## 2     1   2  88   2   C 96.14 69.38
## 11    3   3  74   2   C 50.23 80.42
## 17    4   5  95   1   C 71.48 45.66
```

Some data aggregation functions

Here we introduce built-in base R functions for data aggregation. They are known as the `apply` family of functions from the base package.

- these are vectorized functions compared to `for` loop.
- functions to manipulate slices of data from matrices, arrays, lists and data.frames in a repetitive way.
- helps to perform operations with very few lines of code.
- The family comprises: `apply`, `lapply` , `sapply`, `vapply`, `mapply`, `rapply`, and `tapply`.
- use depends on the structure of the data we wish to operate on, and the format of the output we need.

apply

- apply - applies a function to a matrix’s rows or columns (or, more generally, to dimensions of an array)
- apply is used to a evaluate a function (often an anonymous one) over the margins of an array.

```
## function (X, MARGIN, FUN, ..., simplify = TRUE)
```

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be “retained.” For a matrix 1 indicates rows, 2 indicates columns.
- `FUN` is a function to be applied
- … optional arguments to FUN

Row sum - total individual income (`inc1` + `inc2`)

To calculate row totals, the margin argument should be equal to 1

```
apply(my_df[,6:7], 1, sum, na.rm=TRUE)
```

```
##  [1] 105.45 165.52 163.44 120.49  76.72 179.24  42.01  51.66  68.76 133.25
## [11] 130.65  50.38  89.66 185.26 179.54  74.92 117.14 116.58
```

We can also create a new variable in the data frame and using the `$` sign:

```
my_df$tot_inc<-apply(my_df[,6:7], 1, sum, na.rm=TRUE)
head(my_df, 12L)
```

```
##    hhid pid age sex grp  inc1  inc2 tot_inc
## 1     1   1  72   1   C 61.89 43.56  105.45
## 2     1   2  88   2   C 96.14 69.38  165.52
## 3     1   3  76   2   A 65.50 97.94  163.44
## 4     1   4  89   1   A 51.03 69.46  120.49
## 5     1   5  46   1   C 15.01 61.71   76.72
## 6     2   1  17   2   B 87.04 92.20  179.24
## 7     2   4  73   2   B  1.92 40.09   42.01
## 8     2   5  99   1   A 14.45 37.21   51.66
## 9     3   1   3   1   C    NA 68.76   68.76
## 10    3   2  15   2   A 82.57 50.68  133.25
## 11    3   3  74   2   C 50.23 80.42  130.65
## 12    3   4   0   2   C    NA 50.38   50.38
```

Column averages - mean income for each (inc1, inc2)

- To calculate column average, the margin argument should be equal to 2

```
apply(my_df[,6:7], 2, mean, na.rm=T)
```

```
##    inc1    inc2
## 51.5375 68.1150
```

There are also simplified base R functions to compute some column and row statistics. These are: `colSums()`, `rowSums()`, `colMeans()` and `rowMeans()`.

lapply and sapply Assume we want to round `inc1` and `inc2` to 1 decimal place. Instead of rounding one variable at a time, we can use either of `lapply` or `sapply`. We also define our own function to round.

```
lapply(my_df[6:7], function(x) round(x,1))
```

```
## $inc1
##  [1] 61.9 96.1 65.5 51.0 15.0 87.0  1.9 14.4   NA 82.6 50.2   NA  6.1 92.8 80.8
## [16]  7.9 71.5 39.8
##
## $inc2
##  [1] 43.6 69.4 97.9 69.5 61.7 92.2 40.1 37.2 68.8 50.7 80.4 50.4 83.6 92.5 98.7
## [16] 67.0 45.7 76.8
```

```
#my_df[6:7] <- lapply(my_df[6:7], function(x) round(x,1))
```

tapply `tapply` is used to apply a function over subsets of a vector.

```
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed to FUN
- `simplify` should we simplify the result?
- For example, to sum individual level income within households to total household income

```
tapply(my_df$tot_inc, my_df$hhid, sum, na.rm=T)
```

```
##      1      2      3      4      5
## 631.62 272.91 472.70 556.86 116.58
```

Other related functions

- `ave`
- `aggregate`

Data frame manipulation with `dplyr` functions

What is dplyr?

So far, we have seen that R has most of the fundamental functions to manipulate data. However, the functions lacks consistent coding. Functions outputs do not always flow together such that in some cases you need to create interim objects before merging them back to the main data frame. Some would find the bracket notation difficult to read especially for complicated operations.

This is where `dplyr` comes in to provide consistent ways to manipulate data. One can chain operations such that you do not need to unnecessarily clutter your workspace with temporary files. The `dplyr` work-flow is considered intuitive to write and easy to read.

While this lesson introduces `dplyr`, it might be helpful to introduce the `tidyverse` as well.

What is the `tidyverse`?

- `tidyverse` is a collection of modern R packages that share a common philosophy, embed best practices, and are designed to work together
- Most of the tidyverse packages were created by Hadley Wickham.
- As indicated on the tidyverse homepage, "All packages share an underlying design philosophy, grammar, and data structures"
- The core `tidyverse` packages and their purpose are as listed below:

Core tidyverse packages

- `ggplot2` - for data visualisation or creating graphics
- `dplyr` - for data manipulation using a consistent set of verbs. Regarded as the data wrangling workhorse*
- `tidyr` - for data tidying
- `readr` - for data import
- `purrr` - for functional programming
- `tibble` - for tibbles, a modern reimagining of data frames
- `stringr` - for string manipulation
- `forcats` - for solving common problems with factors
- There are also other important `tidyverse` packages for different functions as listed on the tidyverse page

Installation

- `dplyr` can be installed as a stand alone package or through the `tidyverse` package.
- To install `dplyr` as a stand alone package:

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

- To load `dplyr` into the workspace

```
library("dplyr")
```

- In some cases, the work-flow involves incorporating functions from the various `tidyverse` packages and installing all the `tidyverse` packages has been made easier by one function call as follows:

```
install.packages("tidyverse")
```

- To load core `tidyverse` packages:

```
library("tidyverse")
```

```
## Warning: package 'tidyverse' was built under R version 4.1.3
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.8     v stringr 1.4.0
## v tidyr   1.2.0     v forcats 0.5.1
```

```
## Warning: package 'ggplot2' was built under R version 4.1.3
```

```
## Warning: package 'tibble' was built under R version 4.1.3
```

```
## Warning: package 'tidyr' was built under R version 4.1.3
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Fundamental `dplyr` verbs

- The `dplyr` package has five fundamental data wrangling verbs that works as follows:
  - `filter( )` - rows by their values
  - `arrange( )` - reorder rows
  - `select( )` - pick and/or exclude columns
  - `mutate( )` - create new columns
  - `summarize( )` - collapse rows with summaries

And any of the above may be extended over a group of rows with:

- `group_by( )` - define groups based on categorical variables

Using dplyr functions

- `dplyr` function arguments for the verbs generally work as follows:

```
verb(df, ...)
```

- The first argument is the data frame to work on
- Following arguments define what to do on which named columns in the data frame. The result is always another data frame.
- This is unlike the Base R approach which forces you to repeat the data frame's name each time you want to refer to a variable.

In this section, we explore some variables from a subsample of the National Income Dynamics Study (NIDS) data. You can find more about NIDS In the link, but for now, it might be helpful to know that NIDS is a panel study of individuals in South Africa.The subset used in this course was created from the first wave conducted in 2008. Therefore, the sub-sample is for illustration purposes only and NOT the official data set. If you are interested in the full NIDS dataset, you can access it from the DataFirst website.

This subset has the following variables variables and the description is obtained from the study documents

```
##
##  w1_hhid         'Household identifier'
##  pid             'Individual identifier'
##  w1_prov2011     'Sampled Province (2011 Census)'
##  w1_geo2011      'Sampled GeoType (2011 Census)'
##  w1_r_relhead    'b3 - Household member's relationship to Resident head'
##  w1_best_gen     'Best gender'
##  w1_best_race    'Best population group'
##  w1_best_edu     'Best education'
##  w1_best_age_yrs 'Best age - years'
##  w1_hhincome     'Household monthly income - full imputations'
##  w1_expenditure  'Household Expenditure with full imputations'
```

Now, we import the data set:

```
nids_df <- read_csv("./data/nids_subset.csv")
```

```
## Rows: 28226 Columns: 11
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (11): w1_hhid, pid, w1_prov2011, w1_geo2011, w1_r_relhead, w1_best_gen, ...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Show data frame structure using `glimpse()`

```
glimpse(nids_df)
```

```
## Rows: 28,226
## Columns: 11
## $ w1_hhid         <dbl> 101012, 101013, 101013, 101013, 101013, 101013, 101014~
## $ pid             <dbl> 314585, 314544, 314550, 406295, 406296, 406297, 301454~
## $ w1_prov2011     <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ~
## $ w1_geo2011      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ w1_r_relhead    <dbl> 1, 1, 3, 4, 4, 4, 4, 4, 3, 4, 1, 4, 1, 4, 13, 1, 12, 3~
## $ w1_best_gen     <dbl> 2, 1, 2, 2, 1, 2, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 1, 2, ~
## $ w1_best_race    <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ w1_best_edu     <dbl> 25, 25, 25, 2, 2, 3, 3, 4, 25, 8, 25, 2, 5, 2, 9, 25, ~
## $ w1_best_age_yrs <dbl> 51, 45, 32, 9, 13, 11, 15, 25, 60, 22, 62, 13, 47, 9, ~
## $ w1_hhincome     <dbl> 1035.9819, 568.9643, 568.9643, 568.9643, 568.9643, 568~
## $ w1_expenditure  <dbl> 508.9820, 841.9644, 841.9644, 841.9644, 841.9644, 841.~
```

Arrange

- The `arrange()` function order rows by values of a column(s) in ascending or descending order.
- To re-order the `nids_df` first by `hhid` then `pid`, we type:

```
arrange(nids_df, w1_hhid, pid)
```

```
## # A tibble: 28,226 x 11
##    w1_hhid    pid w1_prov2011 w1_geo2011 w1_r_~1 w1_be~2 w1_be~3 w1_be~4 w1_be~5
##      <dbl>  <dbl>       <dbl>      <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1  101012 314585           2          1       1       2       1      25      51
##  2  101013 314544           2          1       1       1       1      25      45
##  3  101013 314550           2          1       3       2       1      25      32
##  4  101013 406295           2          1       4       2       1       2       9
##  5  101013 406296           2          1       4       1       1       2      13
##  6  101013 406297           2          1       4       2       1       3      11
##  7  101014 301454           2          1       4       1       1       3      15
##  8  101014 314575           2          1       4       1       1       4      25
##  9  101014 314580           2          1       3       2       1      25      60
## 10  101014 314581           2          1       4       2       1       8      22
## # ... with 28,216 more rows, 2 more variables: w1_hhincome <dbl>,
## #   w1_expenditure <dbl>, and abbreviated variable names 1: w1_r_relhead,
## #   2: w1_best_gen, 3: w1_best_race, 4: w1_best_edu, 5: w1_best_age_yrs
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

- use `desc(variable)` to sort variable in descending order

Questions:

- Which household id (`w1_hhid`) has the oldest person?

Filter

- The `filter` function extract rows that meet a logical criteria
- We can filter participants with age older than 100 as follows:

```
filter(nids_df, w1_best_age_yrs>100)
```

```
## # A tibble: 3 x 11
##   w1_hhid    pid w1_pr~1 w1_ge~2 w1_r_~3 w1_be~4 w1_be~5 w1_be~6 w1_be~7 w1_hh~8
##     <dbl>  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1  110037 317499       5       1       1       2       1      25     105   1041.
## 2  111237 301694       4       2       1       2       1      25     101   3484.
## 3  112003 310295       8       1       1       2       1      25     101   1550.
## # ... with 1 more variable: w1_expenditure <dbl>, and abbreviated variable
## #   names 1: w1_prov2011, 2: w1_geo2011, 3: w1_r_relhead, 4: w1_best_gen,
## #   5: w1_best_race, 6: w1_best_edu, 7: w1_best_age_yrs, 8: w1_hhincome
## # i Use `colnames()` to see all variable names
```

- The first argument given to `filter` (and the other `dplyr` verbs) is always the data frame, followed by logical criteria that the returned cases must pass. In this example, the criteria is `w1_best_age_yrs > 100`.
- It is also possible to include multiple criterias:

```
filter(nids_df, w1_best_age_yrs>100 & w1_r_relhead==1)
```

```
## # A tibble: 3 x 11
##   w1_hhid    pid w1_pr~1 w1_ge~2 w1_r_~3 w1_be~4 w1_be~5 w1_be~6 w1_be~7 w1_hh~8
##     <dbl>  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1  110037 317499       5       1       1       2       1      25     105   1041.
## 2  111237 301694       4       2       1       2       1      25     101   3484.
## 3  112003 310295       8       1       1       2       1      25     101   1550.
## # ... with 1 more variable: w1_expenditure <dbl>, and abbreviated variable
## #   names 1: w1_prov2011, 2: w1_geo2011, 3: w1_r_relhead, 4: w1_best_gen,
## #   5: w1_best_race, 6: w1_best_edu, 7: w1_best_age_yrs, 8: w1_hhincome
## # i Use `colnames()` to see all variable names
```

It is also possibole to separate by `,` which is read as `&`

```
filter(nids_df, w1_best_age_yrs>100, w1_r_relhead==1)
```

```
## # A tibble: 3 x 11
##   w1_hhid    pid w1_pr~1 w1_ge~2 w1_r_~3 w1_be~4 w1_be~5 w1_be~6 w1_be~7 w1_hh~8
##     <dbl>  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
```

```
## 1  110037 317499       5       1       1       2       1      25     105   1041.
## 2  111237 301694       4       2       1       2       1      25     101   3484.
## 3  112003 310295       8       1       1       2       1      25     101   1550.
## # ... with 1 more variable: w1_expenditure <dbl>, and abbreviated variable
## #   names 1: w1_prov2011, 2: w1_geo2011, 3: w1_r_relhead, 4: w1_best_gen,
## #   5: w1_best_race, 6: w1_best_edu, 7: w1_best_age_yrs, 8: w1_hhincome
## # i Use `colnames()` to see all variable names
```

If you want the filter to pass at least one test, use the | operator:

```
filter(nids_df, w1_best_age_yrs>100 | w1_r_relhead==1)
```

```
## # A tibble: 6,874 x 11
##    w1_hhid    pid w1_prov2011 w1_geo2011 w1_r_~1 w1_be~2 w1_be~3 w1_be~4 w1_be~5
##      <dbl>  <dbl>       <dbl>      <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1  101012 314585           2          1       1       2       1      25      51
##  2  101013 314544           2          1       1       1       1      25      45
##  3  101014 314582           2          1       1       1       1      25      62
##  4  101015 314570           2          1       1       2       1       5      47
##  5  101016 314110           2          1       1       2       1      25      50
##  6  101017 314531           2          1       1       1       1      25      38
##  7  101018 314578           2          1       1       1       1       7      37
##  8  101020 314542           2          1       1       2       1      25      79
##  9  101021 314567           2          1       1       2       1       6      68
## 10  101022 314558           2          1       1       1       1      25      44
## # ... with 6,864 more rows, 2 more variables: w1_hhincome <dbl>,
## #   w1_expenditure <dbl>, and abbreviated variable names 1: w1_r_relhead,
## #   2: w1_best_gen, 3: w1_best_race, 4: w1_best_edu, 5: w1_best_age_yrs
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Questions:

- How many male household heads are in the `nids_df` data frame?
- How many female household heads aged under 16 are in the `nids_df` data frame?

Select

- The `select` function extract columns by name.
- We can select the `w1_hhid` and `pid` columns as follows:

```
select(nids_df,w1_hhid,pid)
```

```
## # A tibble: 28,226 x 2
##    w1_hhid    pid
##      <dbl>  <dbl>
##  1  101012 314585
##  2  101013 314544
##  3  101013 314550
##  4  101013 406295
##  5  101013 406296
```

```
##  6  101013 406297
##  7  101014 301454
##  8  101014 314575
##  9  101014 314580
## 10  101014 314581
## # ... with 28,216 more rows
## # i Use `print(n = ...)` to see more rows
```

Pipe

- %>% is the pipe operator.
- It takes what is on the left hand side and pushes it into the first argument of the right hand side.
- Piping or chaining commands is thought to increase readability when there are many commands. The %>% operator is imported automatically from the magrittr package
- For example:

```
nids_df %>%
  filter(w1_r_relhead==1 & w1_best_age_yrs>0) %>%  #filter
  arrange(w1_best_age_yrs) %>% #arrange
  select(w1_hhid, pid, w1_best_gen, w1_best_age_yrs)  #select
```

```
## # A tibble: 6,851 x 4
##    w1_hhid    pid w1_best_gen w1_best_age_yrs
##      <dbl>  <dbl>       <dbl>           <dbl>
##  1  102588 302378           1              15
##  2  105525 320852           2              15
##  3  106582 316334           2              15
##  4  107142 306517           1              15
##  5  170743 307022           1              15
##  6  103846 317538           1              16
##  7  107643 303935           1              16
##  8  110544 316922           1              16
##  9  107995 304126           2              17
## 10  108249 307028           2              17
## # ... with 6,841 more rows
## # i Use `print(n = ...)` to see more rows
```

The above takes the `nids_df` data frame **and then** order/sort by `hhid` and `pid`, **then** select to keep their `hhid`, `pid` and `w1_r_best_age_yrs` columns, **then** filter to keep rows with age older than 65. In mathematical function notation, your can interpret the pipe as:

```
x %>% f %>% g %>% h
```

Compare the above with base notation (based on nesting commands):

```
h(g(f(x)))
```

```
nids_hh <- nids_df[nids_df$w1_r_relhead==1 & nids_df$w1_best_age_yrs>0, c('w1_hhid', 'pid','w1_best_gen
nids_hh[order(nids_hh$w1_best_age_yrs),]
```

```
## # A tibble: 7,059 x 4
##    w1_hhid    pid w1_best_gen w1_best_age_yrs
##      <dbl>  <dbl>       <dbl>           <dbl>
##  1  102588 302378           1              15
##  2  105525 320852           2              15
##  3  106582 316334           2              15
##  4  107142 306517           1              15
##  5  170743 307022           1              15
##  6  103846 317538           1              16
##  7  107643 303935           1              16
##  8  110544 316922           1              16
##  9  107995 304126           2              17
## 10  108249 307028           2              17
## # ... with 7,049 more rows
## # i Use `print(n = ...)` to see more rows
```

Mutate

- The `mutate` function create new variables that can be functions of existing variables or a constant.
- The `group_by` command tells R which variable to group by before the next function call.
- Below, we group the `nids_df` by `hhid`, then sort `pid`s within the `hhid` and then generate a row number for each individual within the `hhid`.

```
nids_df <- nids_df %>%
  group_by(w1_hhid) %>%
  arrange(w1_hhid, pid) %>%
  mutate(pnum = row_number()) #1:n()
```

Before we introduce the next functions, we need to clean our data set a little bit by adding factor labels:

```
nids_df <- nids_df %>%
  mutate(w1_prov2011 = factor(w1_prov2011,
                              levels = 1:9,
                              labels = c("Western Cape","Eastern Cape","Northern Cape",
                                         "Free State", "KwaZulu-Natal", "North West",
                                         "Gauteng","Mpumalanga","Limpopo"))) %>%
  mutate(w1_best_race = factor(w1_best_race,
                               levels = 1:4,
                               c("African","Coloured","Asian_Indian", "White"))) %>%
  mutate(w1_best_gen = factor(w1_best_gen, levels = 1:2, c("Male","Female")))
```

```
glimpse(nids_df)
```

```
## Rows: 28,226
## Columns: 12
## Groups: w1_hhid [7,296]
## $ w1_hhid         <dbl> 101012, 101013, 101013, 101013, 101013, 101013, 101014~
## $ pid             <dbl> 314585, 314544, 314550, 406295, 406296, 406297, 301454~
## $ w1_prov2011     <fct> Eastern Cape, Eastern Cape, Eastern Cape, Eastern Cape~
## $ w1_geo2011      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
```

```
## $ w1_r_relhead    <dbl> 1, 1, 3, 4, 4, 4, 4, 4, 3, 4, 1, 4, 1, 4, 13, 1, 12, 3~
## $ w1_best_gen     <fct> Female, Male, Female, Female, Male, Female, Male, Male~
## $ w1_best_race    <fct> African, African, African, African, African, African, ~
## $ w1_best_edu     <dbl> 25, 25, 25, 2, 2, 3, 3, 4, 25, 8, 25, 2, 5, 2, 9, 25, ~
## $ w1_best_age_yrs <dbl> 51, 45, 32, 9, 13, 11, 15, 25, 60, 22, 62, 13, 47, 9, ~
## $ w1_hhincome     <dbl> 1035.9819, 568.9643, 568.9643, 568.9643, 568.9643, 568~
## $ w1_expenditure  <dbl> 508.9820, 841.9644, 841.9644, 841.9644, 841.9644, 841.~
## $ pnum            <int> 1, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 1, 2, 1, 2, ~
```

group_by() and summarise()

- This combination is useful when computing grouped summaries.
- `summarise` uses the provided aggregation function to summarise each group.

We can `summarise` household information - `w1_hhincome`. This variable is repeated or duplicated within household members and we need to filter such that we consider one observation per household. We can filter to keep the first member of the household.

Summarise single variable

```
hhinc_by_prov <- nids_df %>% #create a new data frame from nids_df
  filter(pnum==1) %>% #filter to keep data at the level of the first HH member
  group_by(w1_prov2011) %>% #group by the province variable
  summarize(mean_hhinc = mean(w1_hhincome, na.rm=TRUE)) # compute the mean of hhincome

hhinc_by_prov # print the new data frame
```

```
## # A tibble: 9 x 2
##   w1_prov2011   mean_hhinc
##   <fct>              <dbl>
## 1 Western Cape       7034.
## 2 Eastern Cape       2622.
## 3 Northern Cape      4551.
## 4 Free State         3490.
## 5 KwaZulu-Natal      2829.
## 6 North West         4327.
## 7 Gauteng            7054.
## 8 Mpumalanga         5074.
## 9 Limpopo            3642.
```

Summarise multiple variables

We can also summarise multiple columns, i.e., `w1_hhincome` and `w1_expenditure`:

```
vars_by_prov <- nids_df %>%
  filter(pnum==1) %>% #filter to keep data at the level of the first HH member
    group_by(w1_prov2011) %>% #group by province
    summarize(mean_hhinc = mean(w1_hhincome, na.rm=TRUE),
              sd_hhinc = sd(w1_hhincome, na.rm=TRUE),
              mean_hhexp = mean(w1_expenditure, na.rm=TRUE),
              sd_hhexp = sd(w1_expenditure, na.rm=TRUE))

vars_by_prov
```

```
## # A tibble: 9 x 5
##   w1_prov2011   mean_hhinc sd_hhinc mean_hhexp sd_hhexp
##   <fct>              <dbl>    <dbl>      <dbl>    <dbl>
## 1 Western Cape       7034.    9255.      6782.   10358.
## 2 Eastern Cape       2622.    4928.      2310.    4796.
## 3 Northern Cape      4551.    6390.      3811.    5512.
## 4 Free State         3490.    6108.      3012.    4424.
## 5 KwaZulu-Natal      2829.    4998.      2634.    6760.
## 6 North West         4327.    8112.      3421.    5112.
## 7 Gauteng            7054.   11913.      5367.    7639.
## 8 Mpumalanga         5074.    9118.      4580.    8049.
## 9 Limpopo            3642.    8873.      3121.    5915.
```

across

However, there is an efficient way of summarising multiple variables - `across()`:

```
nids_df %>%
  filter(pnum==1) %>% #filter to keep data at the level of the first HH member
  group_by(w1_prov2011) %>% #group by province
  summarise(across(c(w1_hhincome,w1_expenditure), list(mean = mean, sd = sd), na.rm=TRUE))
```

```
## # A tibble: 9 x 5
##   w1_prov2011   w1_hhincome_mean w1_hhincome_sd w1_expenditure_mean w1_expendi~1
##   <fct>                    <dbl>          <dbl>               <dbl>        <dbl>
## 1 Western Cape             7034.          9255.               6782.       10358.
## 2 Eastern Cape             2622.          4928.               2310.        4796.
## 3 Northern Cape            4551.          6390.               3811.        5512.
## 4 Free State               3490.          6108.               3012.        4424.
## 5 KwaZulu-Natal            2829.          4998.               2634.        6760.
## 6 North West               4327.          8112.               3421.        5112.
## 7 Gauteng                  7054.         11913.               5367.        7639.
## 8 Mpumalanga               5074.          9118.               4580.        8049.
## 9 Limpopo                  3642.          8873.               3121.        5915.
## # ... with abbreviated variable name 1: w1_expenditure_sd
```

It is also possible to have more than one variable in the `group_by()`. In this example, we also introduce the helper function `n()` which counts the number of rows in a group. We first create a household size variable that counts individuals within the `w1_hhid`, then keep information for the household head, generate per capita household income and expenditure, and then summarise the per capita variables to caluculate their `mean` and `sd` by race and gender:

```
incexppc_byrace_bygen <- nids_df %>%
  group_by(w1_hhid) %>% #group by hhid
  mutate(hhsize = n()) %>% # count number of members
  filter(w1_r_relhead==1) %>% #keep info for HHhead
  mutate(hhinc_pc = w1_hhincome/hhsize, hhexp_pc = w1_expenditure/hhsize) %>% # scale pc
  group_by(w1_best_race, w1_best_gen) %>% #group by race and gender
  summarise(across(c(hhinc_pc,hhexp_pc), list(mean = mean, sd = sd), na.rm=TRUE))
```

```
## `summarise()` has grouped output by 'w1_best_race'. You can override using the
## `.groups` argument.
```

```
incexppc_byrace_bygen
```

```
## # A tibble: 8 x 6
## # Groups:   w1_best_race [4]
##   w1_best_race w1_best_gen hhinc_pc_mean hhinc_pc_sd hhexp_pc_mean hhexp_pc_sd
##   <fct>        <fct>               <dbl>       <dbl>         <dbl>       <dbl>
## 1 African      Male                1454.       3429.         1054.       1692.
## 2 African      Female               761.       1112.          714.       1450.
## 3 Coloured     Male                1537.       2903.         1283.       1812.
## 4 Coloured     Female              1084.       1234.          955.       1145.
## 5 Asian_Indian Male                3902.       4182.         3771.       3689.
## 6 Asian_Indian Female              2858.       2348.         3887.       4613.
## 7 White        Male                7367.       7138.         6912.       7101.
## 8 White        Female              6744.       5390.         7059.       4807.
```

count()

- The count function helps to count observations by group:

```
nids_df %>%
  group_by(w1_prov2011) %>%
  count(sort = TRUE)
```

```
## # A tibble: 9 x 2
## # Groups:   w1_prov2011 [9]
##   w1_prov2011       n
##   <fct>         <int>
## 1 KwaZulu-Natal  8011
## 2 Western Cape   3626
## 3 Eastern Cape   3531
## 4 Gauteng        2778
## 5 Limpopo        2477
## 6 Northern Cape  2224
## 7 Mpumalanga     2060
## 8 North West     1875
## 9 Free State     1644
```

Recoding

if_else We can recode to a binary variable using the `if_else` function. Below we show one way to recode participants into those in the working age group (`wkgrp`) and dependencies (`dep`):

```
nids_df <- nids_df %>%
  mutate(wkgrp = if_else(w1_best_age_yrs>=15 & w1_best_age_yrs<=65,1,0),
         dep = if_else(w1_best_age_yrs<15 | w1_best_age_yrs>65,1,0))

glimpse(nids_df)
```

```
## Rows: 28,226
## Columns: 14
```

```
## Groups: w1_hhid [7,296]
## $ w1_hhid         <dbl> 101012, 101013, 101013, 101013, 101013, 101013, 101014~
## $ pid             <dbl> 314585, 314544, 314550, 406295, 406296, 406297, 301454~
## $ w1_prov2011     <fct> Eastern Cape, Eastern Cape, Eastern Cape, Eastern Cape~
## $ w1_geo2011      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ w1_r_relhead    <dbl> 1, 1, 3, 4, 4, 4, 4, 4, 3, 4, 1, 4, 1, 4, 13, 1, 12, 3~
## $ w1_best_gen     <fct> Female, Male, Female, Female, Male, Female, Male, Male~
## $ w1_best_race    <fct> African, African, African, African, African, African, ~
## $ w1_best_edu     <dbl> 25, 25, 25, 2, 2, 3, 3, 4, 25, 8, 25, 2, 5, 2, 9, 25, ~
## $ w1_best_age_yrs <dbl> 51, 45, 32, 9, 13, 11, 15, 25, 60, 22, 62, 13, 47, 9, ~
## $ w1_hhincome     <dbl> 1035.9819, 568.9643, 568.9643, 568.9643, 568.9643, 568~
## $ w1_expenditure  <dbl> 508.9820, 841.9644, 841.9644, 841.9644, 841.9644, 841.~
## $ pnum            <int> 1, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 1, 2, 1, 2, ~
## $ wkgrp           <dbl> 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, ~
## $ dep             <dbl> 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, ~
```

Q: Calculate the dependency ratio by province.

case_when The `dplyr` package also has the `case_when` function for recoding a variable. For example, to recode the age variable to create age bins, i.e. age bin is 1 for ages between 20 and 29, 2 for ages 30 - 39, e.t.c.

```
nids_df<-nids_df %>%
  mutate(age_bins=case_when(w1_best_age_yrs >= 20 & w1_best_age_yrs <=29 ~ 1,
                            w1_best_age_yrs > 29 & w1_best_age_yrs <=39 ~ 2,
                            w1_best_age_yrs > 39 & w1_best_age_yrs <=49 ~ 3,
                            w1_best_age_yrs > 49 & w1_best_age_yrs <=59 ~ 4,
                            w1_best_age_yrs > 59 & w1_best_age_yrs <=69 ~ 5,
                            w1_best_age_yrs > 69 & w1_best_age_yrs <=120 ~ 6))
```

Making age_bins a factor variable with labels

```
nids_df <- nids_df %>%
  mutate(age_bins = factor(age_bins,
                           levels = 1:6,
                           labels = c("20 - 29 yrs","30 - 39 yrs", "40 - 49 yrs", "50 - 59 yrs",
                                      "60 - 69 yrs", "70 - 120 yrs")))
```

Inspect our age bins using `group_by` and `summarise` - `n()`

```
nids_df%>%
  group_by(age_bins)%>%
  summarise(freq = n())
```

```
## # A tibble: 7 x 2
##   age_bins      freq
##   <fct>        <int>
## 1 20 - 29 yrs   4545
## 2 30 - 39 yrs   3279
## 3 40 - 49 yrs   2870
## 4 50 - 59 yrs   2227
```

```
## 5 60 - 69 yrs   1414
## 6 70 - 120 yrs  1097
## 7 <NA>         12794
```

```
#nids_df %>%  group_by(age_bins) %>% count
```

And more…

Data visualisation - Base R

Base R has functions to produce a wide range of graphics. For purposes of fast data exploration, R base graphics are easy and quick to produce. Base R graphics are created interactively and therefore publication quality figures need more customisation of graphical parameters like fonts, colors, line styles, axes, reference lines, etc. The concepts and commands to achieve base R graphing are explained in the R Manuals' Graphical Procedures section and it is best to refer to this for more details. We will go through some of the concepts using examples.

The base R plotting system

- Plotting commands are divided into three basic groups:
- **High level** graphical commands- these functions create a new plot on the graphics device, possibly with axes, labels, titles and so on.
- The following is a list of some of the high level plotting functions and the output graphics.
  - `plot()` - Scatter plot and general plotting
  - `hist()` - Histogram
  - `stem()` - Stem-and-leaf
  - `boxplot()` - Boxplot
  - `qqnorm()` - Normal probability plot
  - `mosaicplot()` - Mosaic plot
- **Low level** graphical commands - these functions add more information to an existing plot, such as extra points, lines and labels.
- The following is a list of some of the low level plotting functions and their purpose
  - `points()` - add points
  - `lines()` - add lines defined by data points
  - `text()` - add text
  - `abline()` - add lines defined by parameters
  - `legend()` - add a legend
- **Interactive** graphics functions allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.

In addition, R maintains a list of *graphical parameters* which can be manipulated to customize your plots.

Below, we first create a subset of the `nids_df` data frame by keeping selected household information on income, expenditure, population group and of the household head.

```
inc_exp <- nids_df %>%
  filter(w1_r_relhead==1) %>%
  select(w1_hhid, w1_hhincome,w1_expenditure, w1_best_race, w1_best_gen)
```

High-level plotting functions

`plot()`

- The `plot()` function is one of the frequently used plotting functions in base R.
- It is a generic function where the type of plot produced is dependent on the type or class of the first argument.
- The basic inputs are as follows:

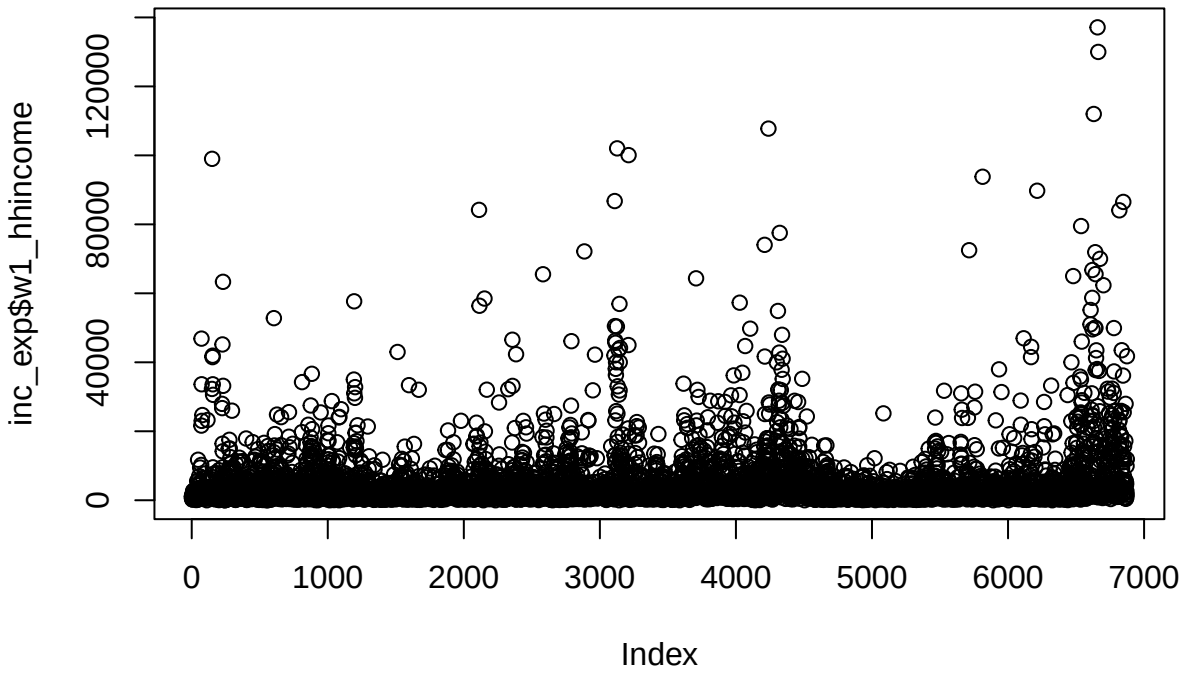
```
str(plot)
```

```
## function (x, y, ...)
```

- `x` - the x coordinates of points in the plot
- `y` - the y coordinates of points in the plot, optional if x is an appropriate structure.
- `...` - Arguments to be passed to methods, such as graphical parameters

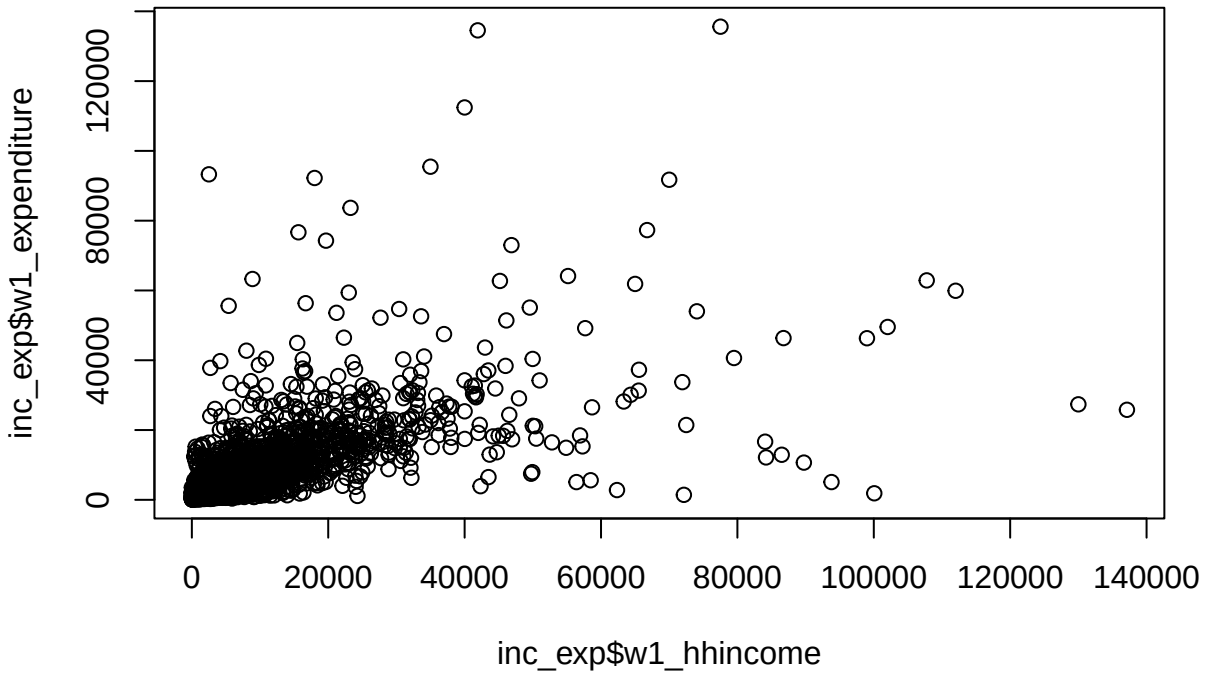
We start by plotting a single numerical variable - `plot(x)`

```
plot(inc_exp$w1_hhincome)
```



- The above produces a scatter plot of values of `x` against an index of their position in the data set.
- It is also possible to plot two numeric variables - `plot(x,y)`

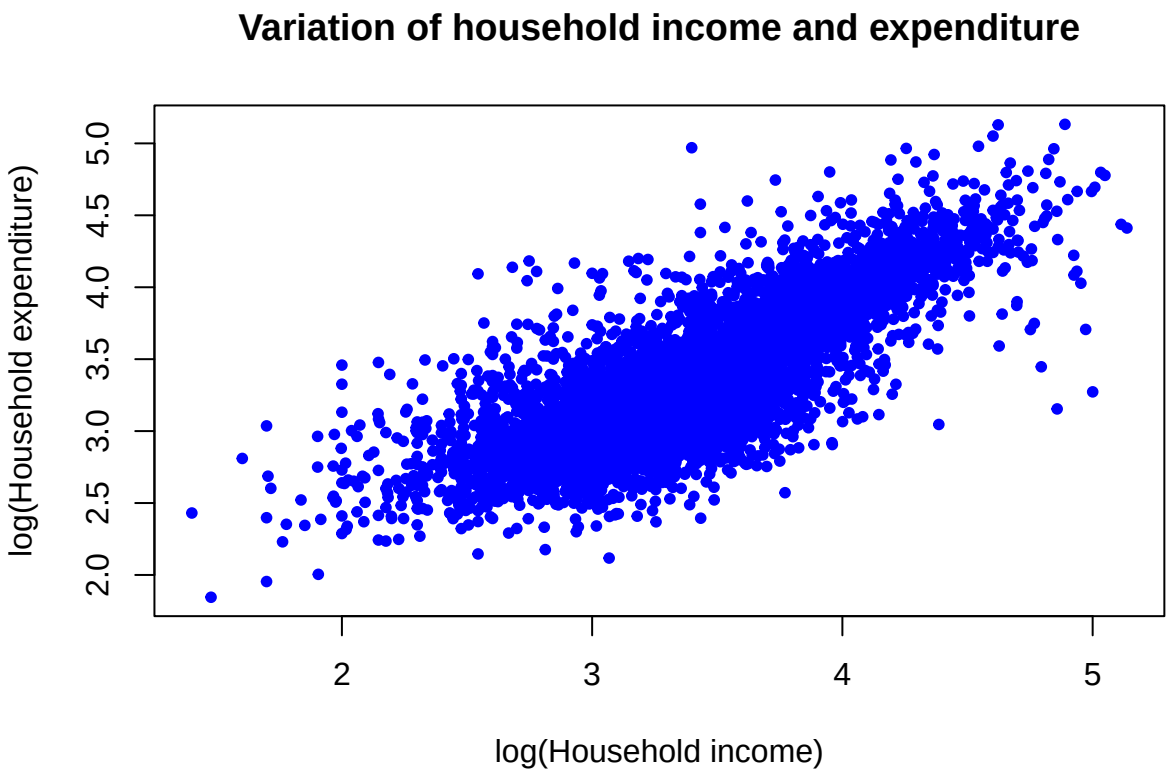
```
plot(inc_exp$w1_hhincome, inc_exp$w1_expenditure)
```



For any plot you can customize many features of your graphs (fonts, colors, axes, titles) through graphic options For example, we can change the shape of the data point using the `pch` option.

It is also possible to transform the data while plotting - this might be convenient if one wants a quick view of the transformed relationship.

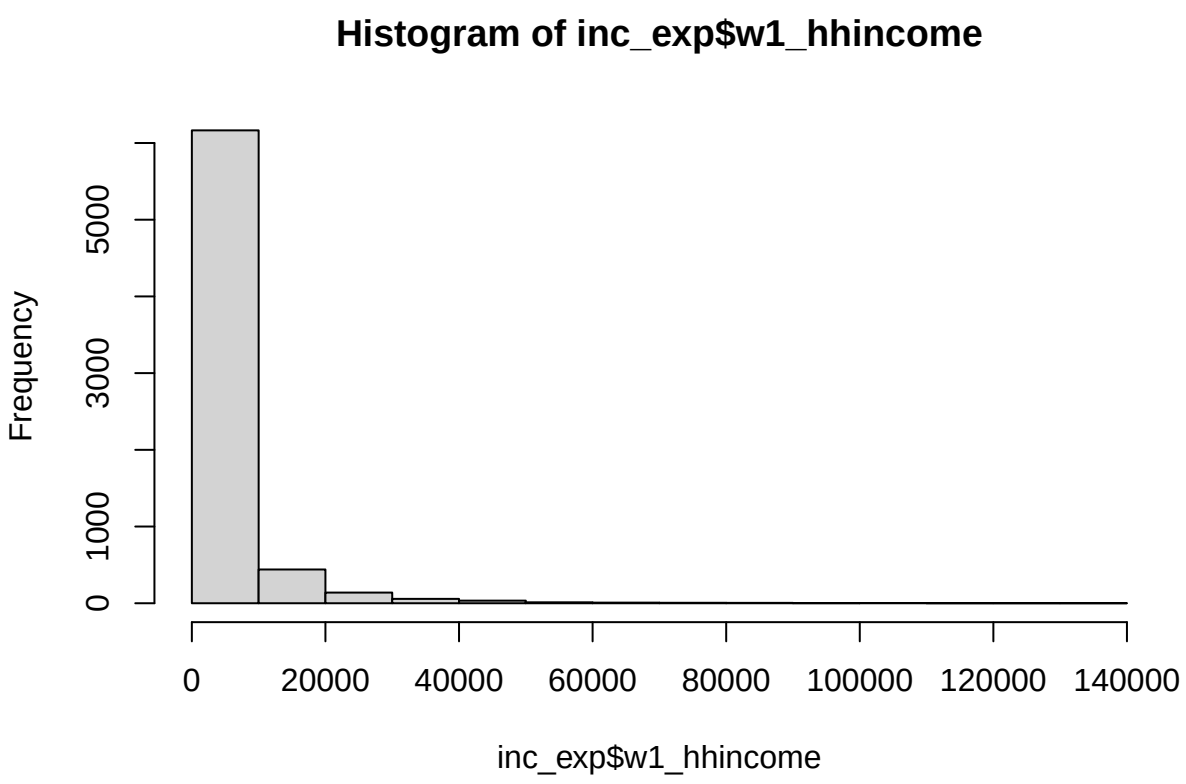
```
plot(log10(inc_exp$w1_hhincome),log10(inc_exp$w1_expenditure),
     xlab = "log(Household income)", #horizontal axis label
     ylab = "log(Household expenditure)", #vertical axis label
     main = "Variation of household income and expenditure", #plot title
     col = "blue", #colour
     pch = 20) #sympol
```



`hist()`

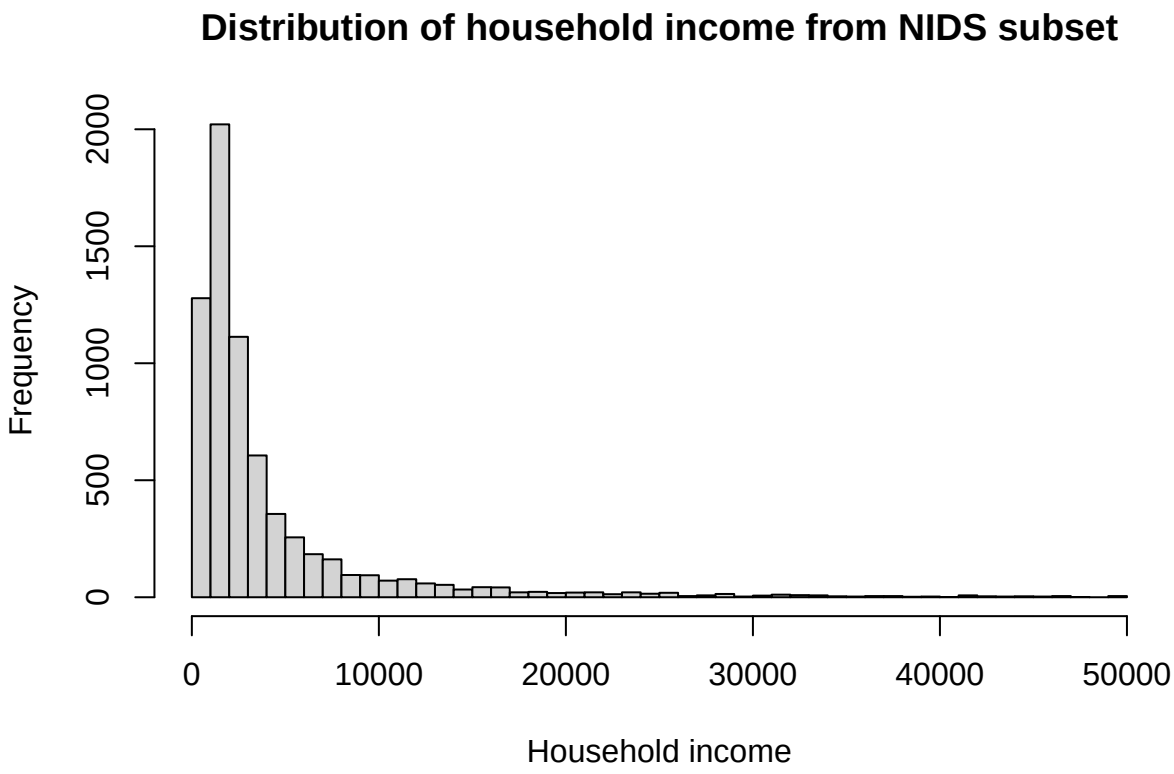
- A histogram provides a visual representation of a distribution.
- `hist()` is the function for plotting a histogram in R.
- The function bin all values and plot frequency/proportions of bins.
- Lets take a look at the distribution of the household income variable.

```
hist(inc_exp$w1_hhincome)
```



There are very few households that earn more than R 50 000. We can filter out these households and plot the distribution for the remaining households

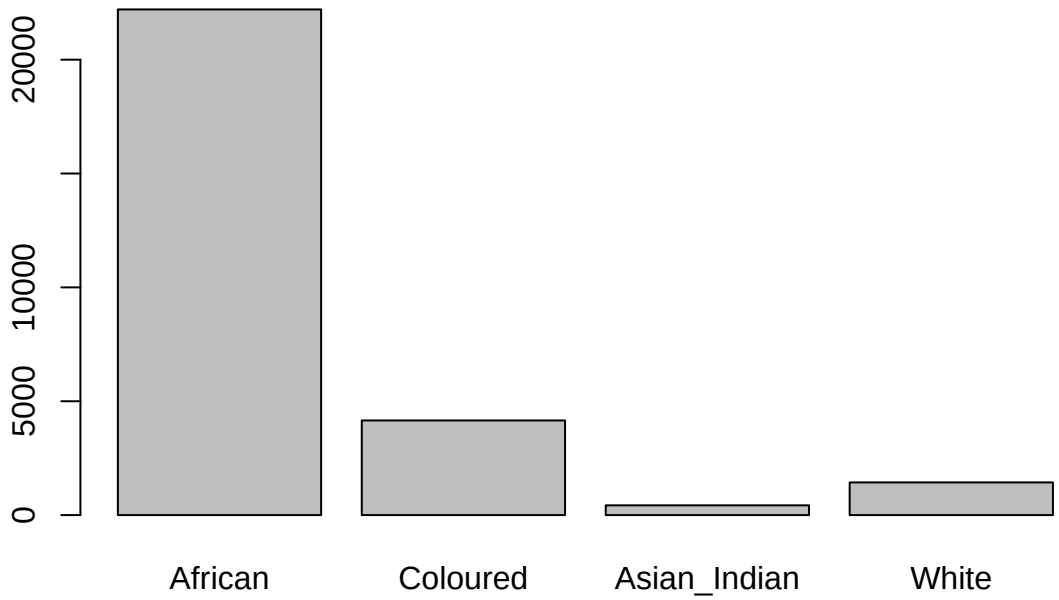
```
hist(inc_exp$w1_hhincome[inc_exp$w1_hhincome<=50000],
     breaks=50,
     xlab = "Household income",
     main = "Distribution of household income from NIDS subset")
```



`barplot()`

- This is the function to plot a bar graph other than `plot()` a categorical variable
- The input is either a vector or matrix of values describing the bars which make up the plot
- We can plot a bar graph of the `w1_best_race` from the table object as follows:

```
counts<-table(nids_df$w1_best_race)
barplot(counts)
```

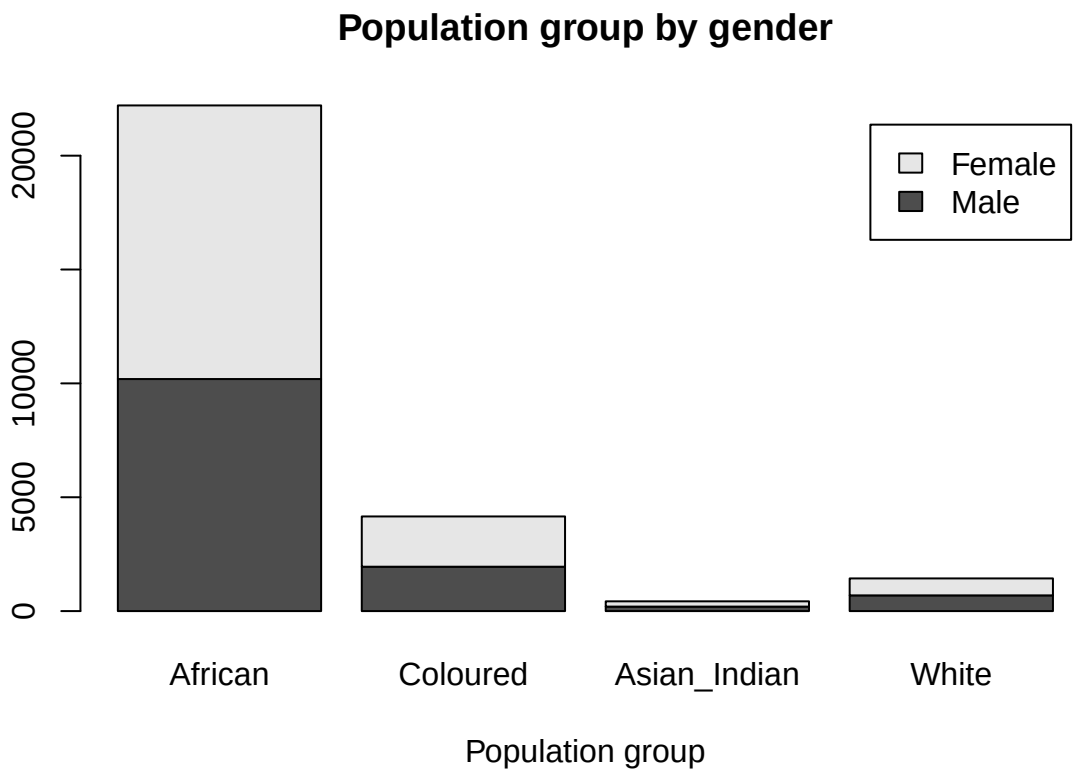


We can also stack the bar graph by gender

Stacked bar plot

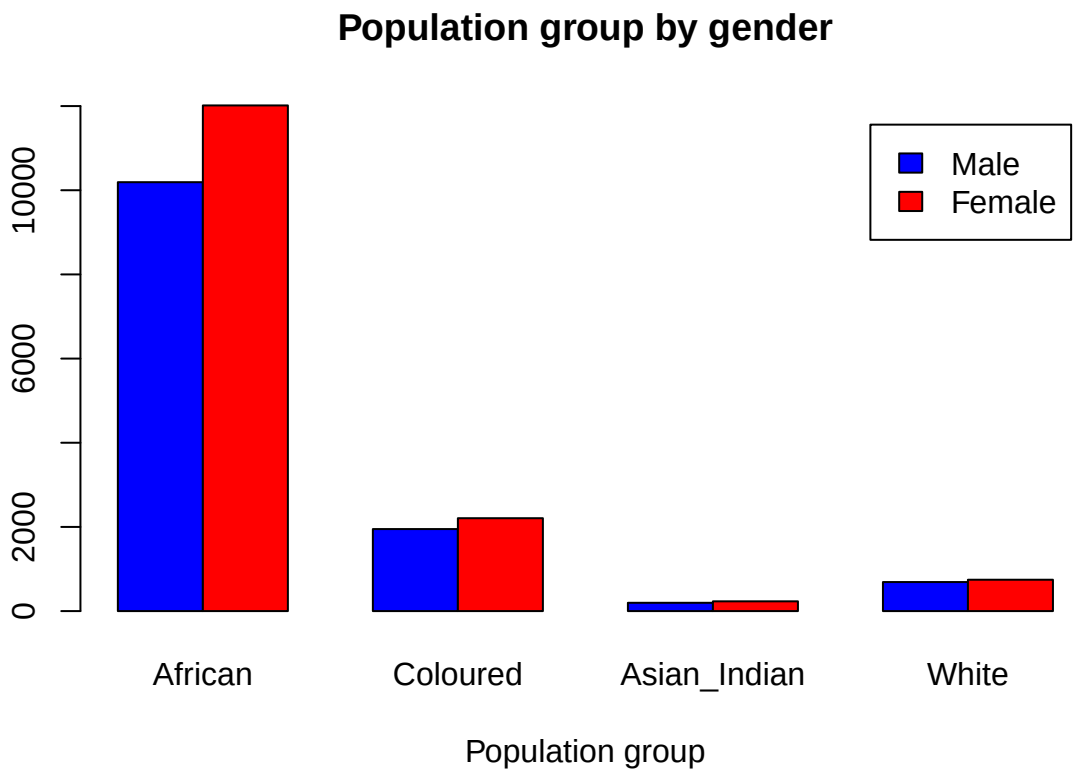
```
counts1<-table(nids_df$w1_best_gen, nids_df$w1_best_race)

barplot(counts1,
        main="Population group by gender",
        xlab="Population group",
        #col=c("blue","red"),
        legend = rownames(counts1))
```



Grouped bar plot (dodge)

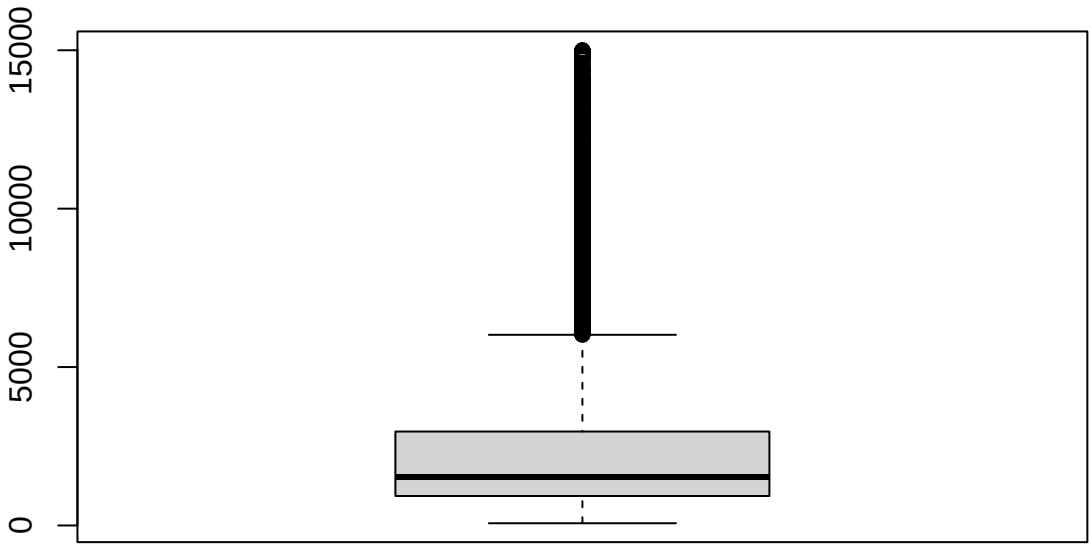
```
barplot(counts1,
        main="Population group by gender",
        xlab="Population group",
        col=c("blue","red"),
        legend = rownames(counts1), beside=TRUE)
```



`boxplot()`

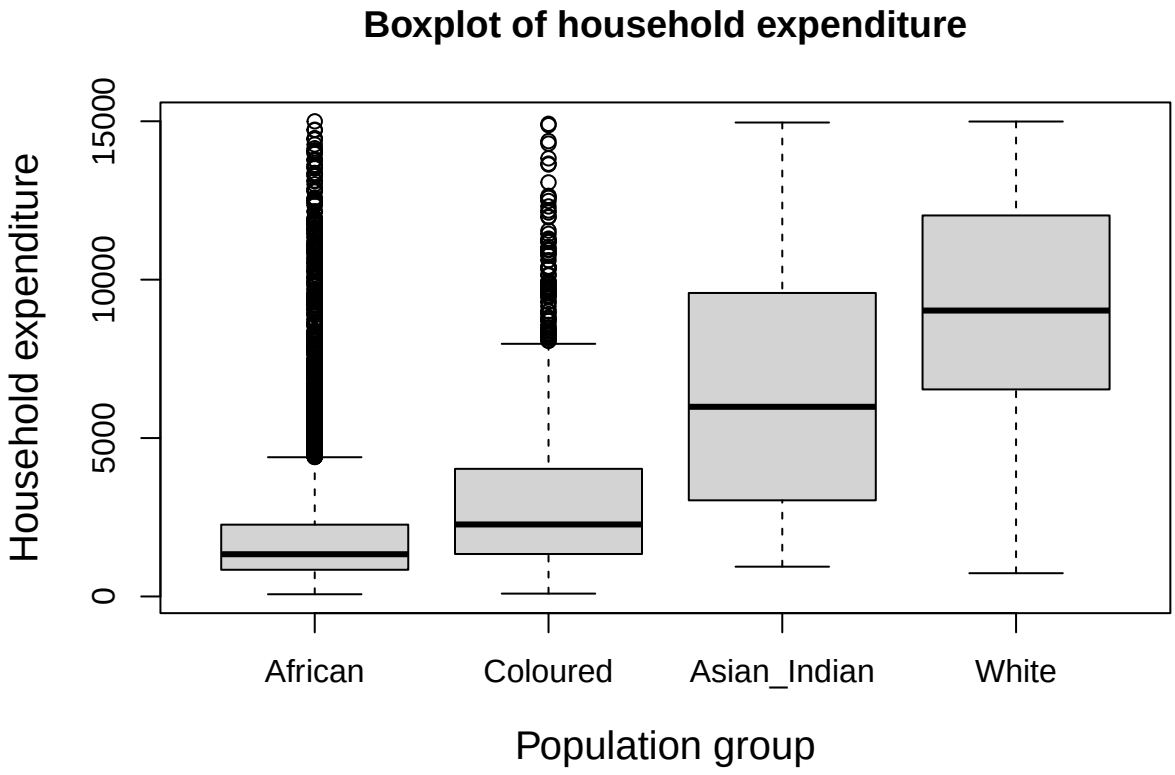
The boxplot is another useful plot to check the distribution of a variable. The `boxplot()` function can plot for individual variables or for variables by group. The boxplot for an individual variable (e.g. household expenditure) can be plotted as follows:

```
# we excluded expenditure greater than 15k for ease of viewing the plot
boxplot(inc_exp$w1_expenditure[inc_exp$w1_expenditure<15000])
```



We can also use a formula (~) to plot the boxplot of a variable over categories of another. The box plot function also allows the following argument input: `boxplot(x, data = )` where `x` is a formula and `data` is data frame from where the variables are contained. Below we plot household expenditure by population group.

```
boxplot(w1_expenditure ~ w1_best_race,
        data=inc_exp[inc_exp$w1_expenditure<15000,],
        main = "Boxplot of household expenditure",
        xlab = "Population group", ylab = "Household expenditure",cex.lab = 1.25)
```



There are many other graphics than can be plotted using base R, as well as other customisation that can be done.

Graphical Parameters

The preceding sections have introduced mostly one-line quick graph plotting commands. We can customise each of the graphs to our liking by changing a number of the default parameters. Most of the base R plotting commands accept additional graphical parameters. The command `par()` set parameters for plotting and below are some of the commonly used parameters and their effect.

- Some graphical parameters in R:
  - `cex` - font size
  - `col` - color of plotting symbols
  - `lty` - line type
  - `lwd` - line width
  - `mar` - inner margins
  - `mfrow` - splits plotting area (mult. figs. per page)
  - `oma` - outer margins
  - `pch` - plotting symbol
  - `xlim` - Min and max of X axis range
  - `ylim` - Min and max of Y axis range
- Type `par()` to see various default parameters
- Also `?par`

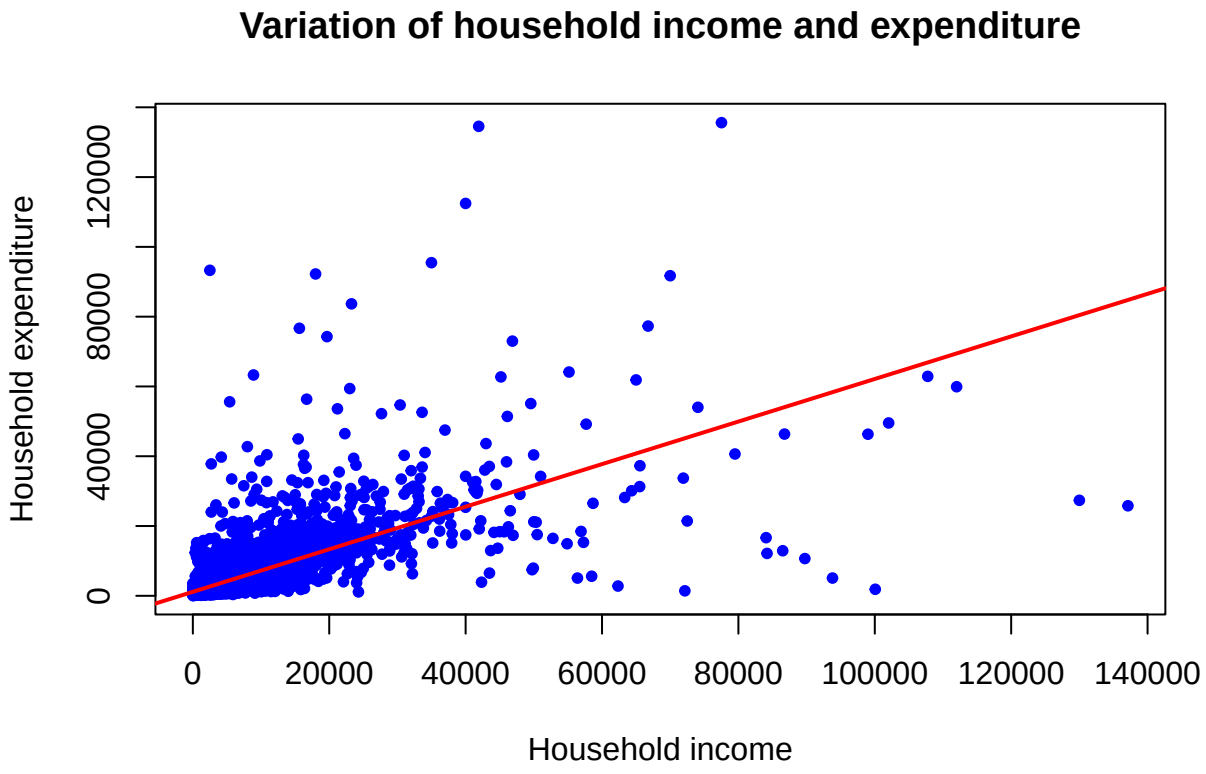
Add to the plot

- Here we learn how to add/stack other graphical elements

`abline()`

- We can add one or more straight lines using the `abline()` function
- Assume we have parameters of a straight line (`y = a + b.x`) where the slope is `0.61` and intercept is `1137`, we can add this to the previous plot as follows:

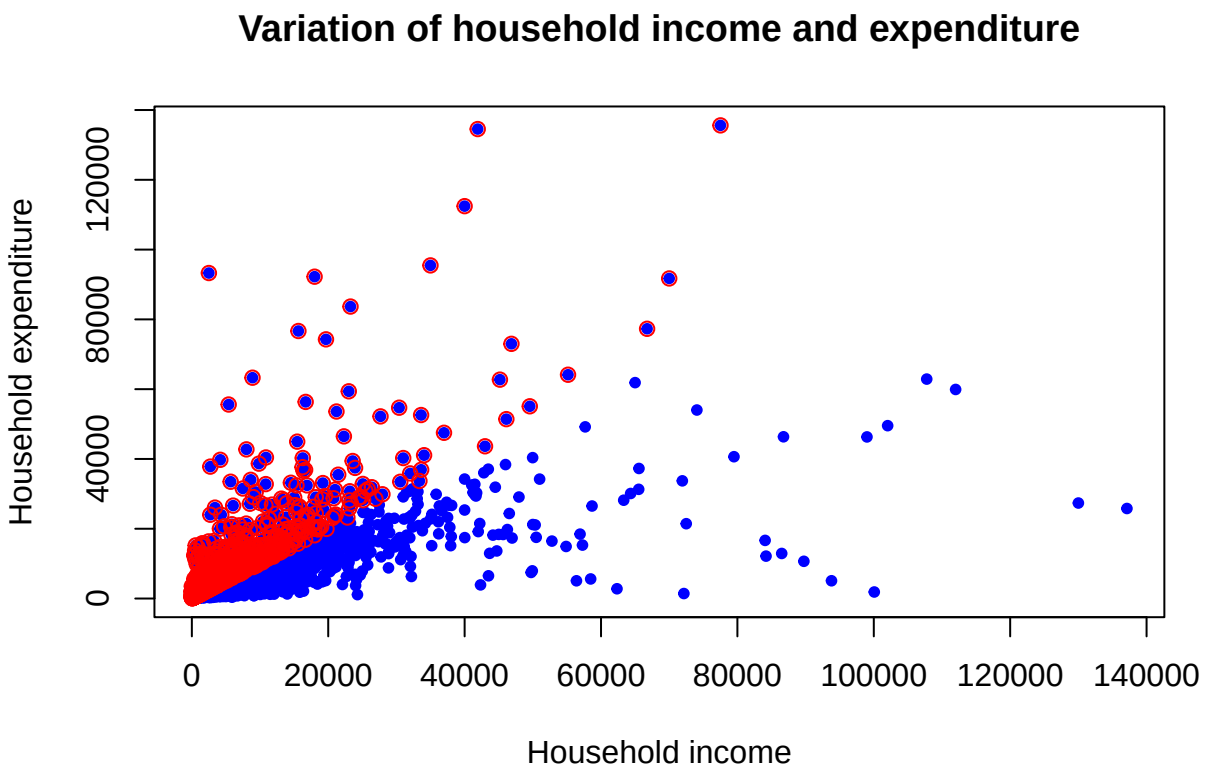
```
plot(inc_exp$w1_hhincome,inc_exp$w1_expenditure,
     xlab = "Household income", #horizontal axis label
     ylab = "Household expenditure", #vertical axis label
     main = "Variation of household income and expenditure", #plot title
     col = "blue", #colour
     pch = 20) #sympol
abline(a = 1137, b = 0.61,lwd = 2, col="red") # add abline
```



`points()`

- It is also possible to add points to an existing plot
- For example, we can hightlight points where expenditure is greater than income

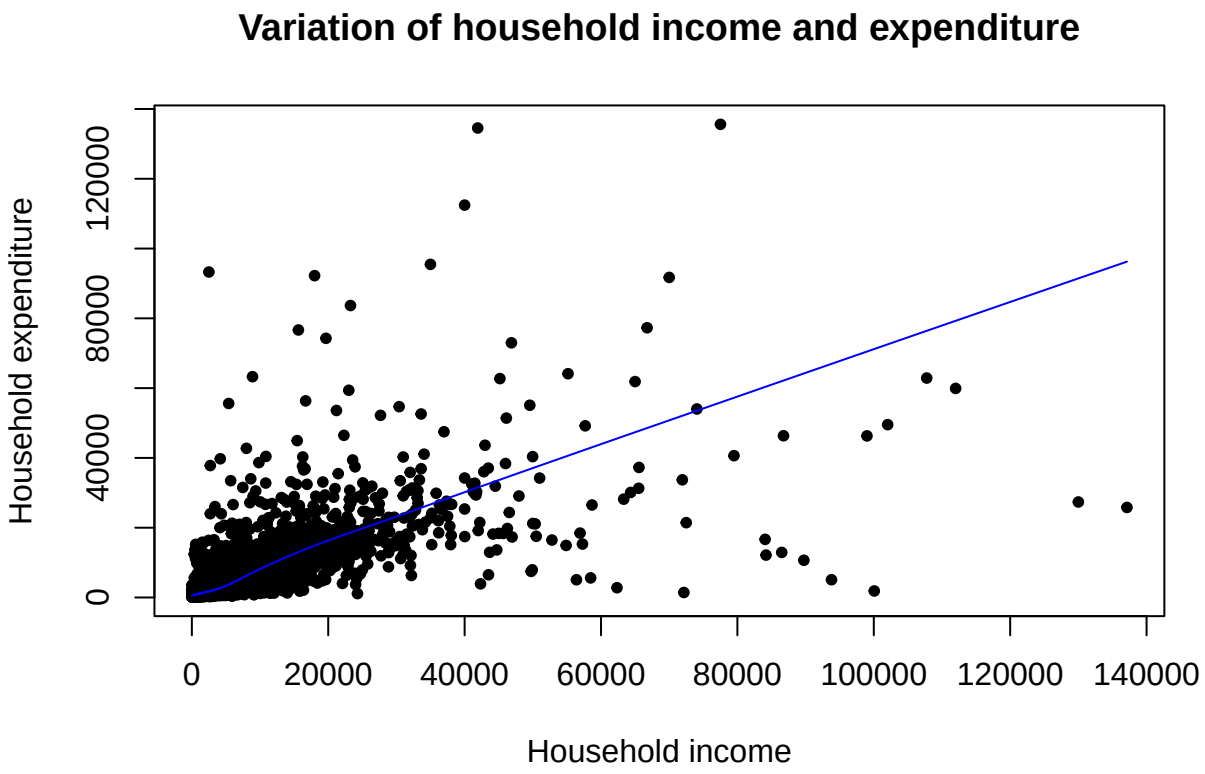
```
plot(inc_exp$w1_hhincome,inc_exp$w1_expenditure,
     xlab = "Household income", #horizontal axis label
     ylab = "Household expenditure", #vertical axis label
     main = "Variation of household income and expenditure", #plot title
     col = "blue", #colour
     pch = 20) #sympol
points(inc_exp[inc_exp$w1_expenditure>inc_exp$w1_hhincome, c("w1_hhincome", "w1_expenditure")], col = "
```



`lines()`

- `lines` is a generic function that takes coordinates given in various ways and join the corresponding points with line segments.
- In the following example, we add a line based on the LOWESS smoother fitted by the `stats` package function `loess`

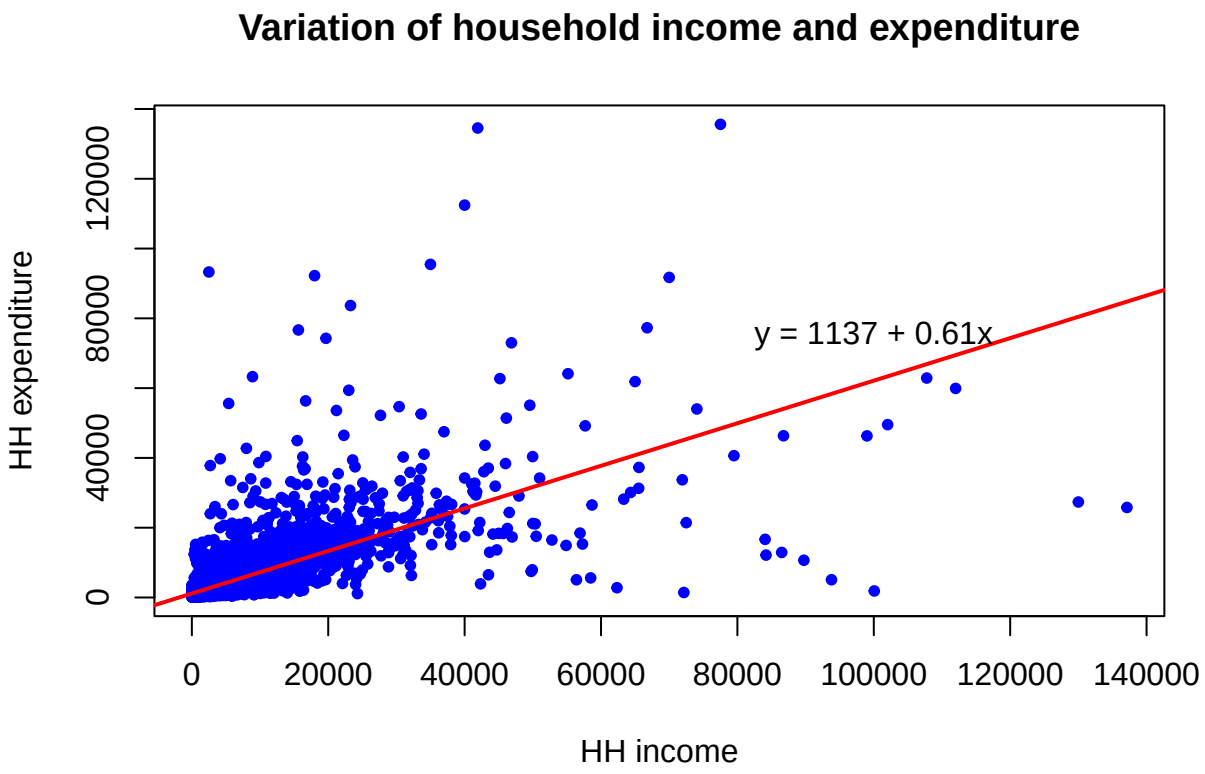
```
plot(inc_exp[c("w1_hhincome","w1_expenditure")],
     xlab = "Household income", #horizontal axis label
     ylab = "Household expenditure", #vertical axis label
     main = "Variation of household income and expenditure", #plot title
     pch = 20) #sympol
lines(stats::lowess(inc_exp[c("w1_hhincome","w1_expenditure")]), col = "blue")
```



`text()`

- `text` draws the strings given in the vector labels at the coordinates given by `x` and `y`.
- Going back to our `abline` guess above, we can add the equation, `y = 1137 + 0.61x`, to the plot

```
plot(inc_exp$w1_hhincome,inc_exp$w1_expenditure,
     xlab = "HH income", #horizontal axis label
     ylab = "HH expenditure", #vertical axis label
     main = "Variation of household income and expenditure", #plot title
     col = "blue", #colour
     pch = 20) #sympol
abline(a = 1137, b = 0.61, lwd = 2, col = "red") #add line
text(x = 100000,y = 75000, labels = "y = 1137 + 0.61x") #add equation at (x,y)
```



There are also other low level graphics functions like `segments`,

Saving plots

- You can save the plot using a graphics device such as `pdf`, `svg`, `postscript`, `png`, `jpeg`, `tiff`
- To save any of the above plots in PDF format in a file called `myplot.pdf` use the following code.

```
pdf("./figures/myplot.pdf")
boxplot(nids_df$w1_hhincome ~ nids_df$w1_best_race, main = "Boxplot of HH income",
        xlab = "Population group", ylab = "Household income",cex.lab = 1.25)
dev.off()
```

```
## pdf
##   2
```

- The `dev.off` function closes the open plotting window or device
- There are also options to set the device size (figure dimensions and resolution) specified for each device.
- Below is an example for setting the width, height and resolution for a png device

```
png(filename = "./figures/myplot.png", width = 20, height = 20, units = "cm", res = 400)
boxplot(nids_df$w1_hhincome ~ nids_df$w1_best_race, main = "Boxplot of HH income",
        xlab = "Population group", ylab = "Household income",cex.lab = 1.25)
dev.off()
```

```
## pdf
##   2
```

We have scratched the surface of base graphics and I hope it will help someone getting started and explore more complex graphs

Data visualisation - ggplot2

The ggplot2 package created by (Wickham 2016) is another member of the tidyverse family introduced in previous sections. The package is based on "The Grammer of Graphics" book (Wilkinson 2012).It uses layer elements to build a graphic. We start by re-creating some of the plots created above using the base plotting system, but using ggplot functions to familiarise with the syntax.

We first need to load the ggplot2 library in order to access functions from the package:

```
library(ggplot2)
```

Data

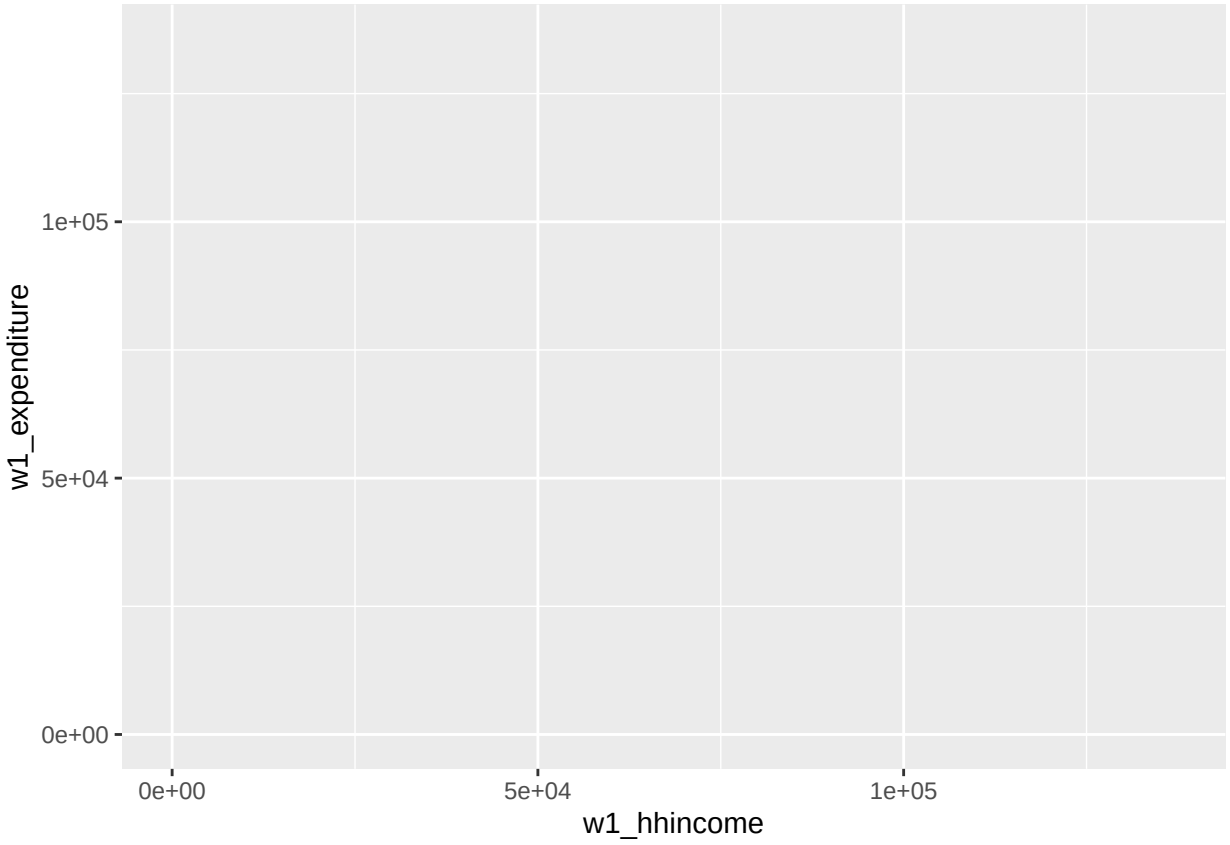
We are going to start by plotting the previous scatter plot from scratch and the first step is to identify the data set. The data set in our case is the inc_exp with household level income and expenditure. The ggplot() function creates a coordinate system or a blank canvass.

```
ggplot(data = inc_exp)
```

Aesthetics

These are visual properties of the objects in the plot - at the least your `x-variable` and/or `y-variable`. For a scatter plot, we need both `x` and `y` variables.

```
ggplot(data = inc_exp, aes(x=w1_hhincome, y=w1_expenditure))
```



They set the limits or default settings.

Example aesthetics

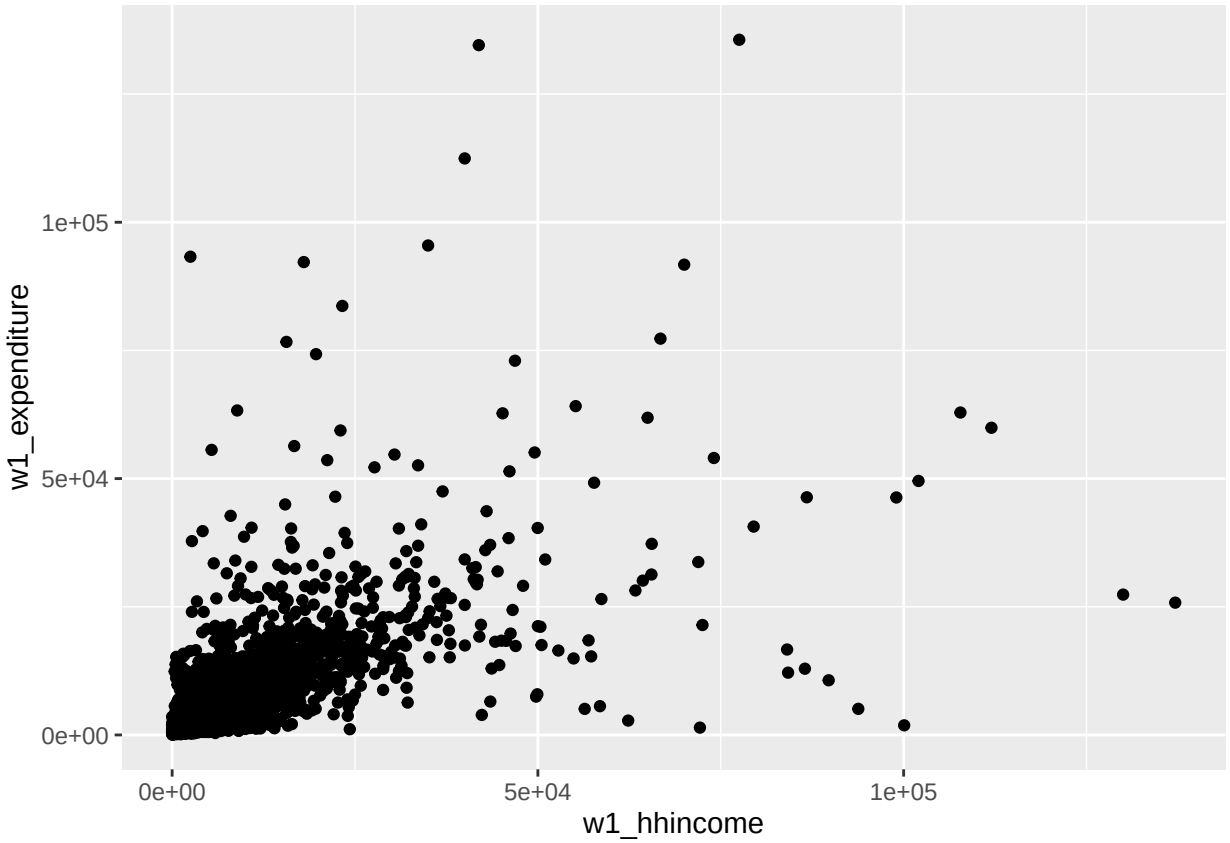
- `x`: positioning along x-axis
- `y`: positioning along y-axis
- `color`: color of objects
- `fill`: fill color of objects
- `alpha`: transparency of objects (value between 0, transparent, and 1, `opaque` - inverse of how many stacked objects it will take to be opaque)
- `linetype`: how lines should be drawn (solid, dashed, dotted, etc.)
- `shape`: shape of markers in scatter plots
- `size`: how large objects appear

Geom (add a geom layer)

- This is how the data will be visualized -`geometric objects` to show up on the plot.

- This is done using `geom_*()` where `*` is one of the several geoms that we can plot.
- When plotting with `ggplot2`, you specify different parts of the plot, and add them together using the `+` operator.
- For a scatter plot, use `geom_point()` and `+` to the plot above.

```
ggplot(data = inc_exp, aes(x=w1_hhincome, y=w1_expenditure)) +
  geom_point()
```



- The bare minimum required to create a plot in ggplot2 are: `dataset`, `X` and `Y` variables and the `geometric` shape to represent data.

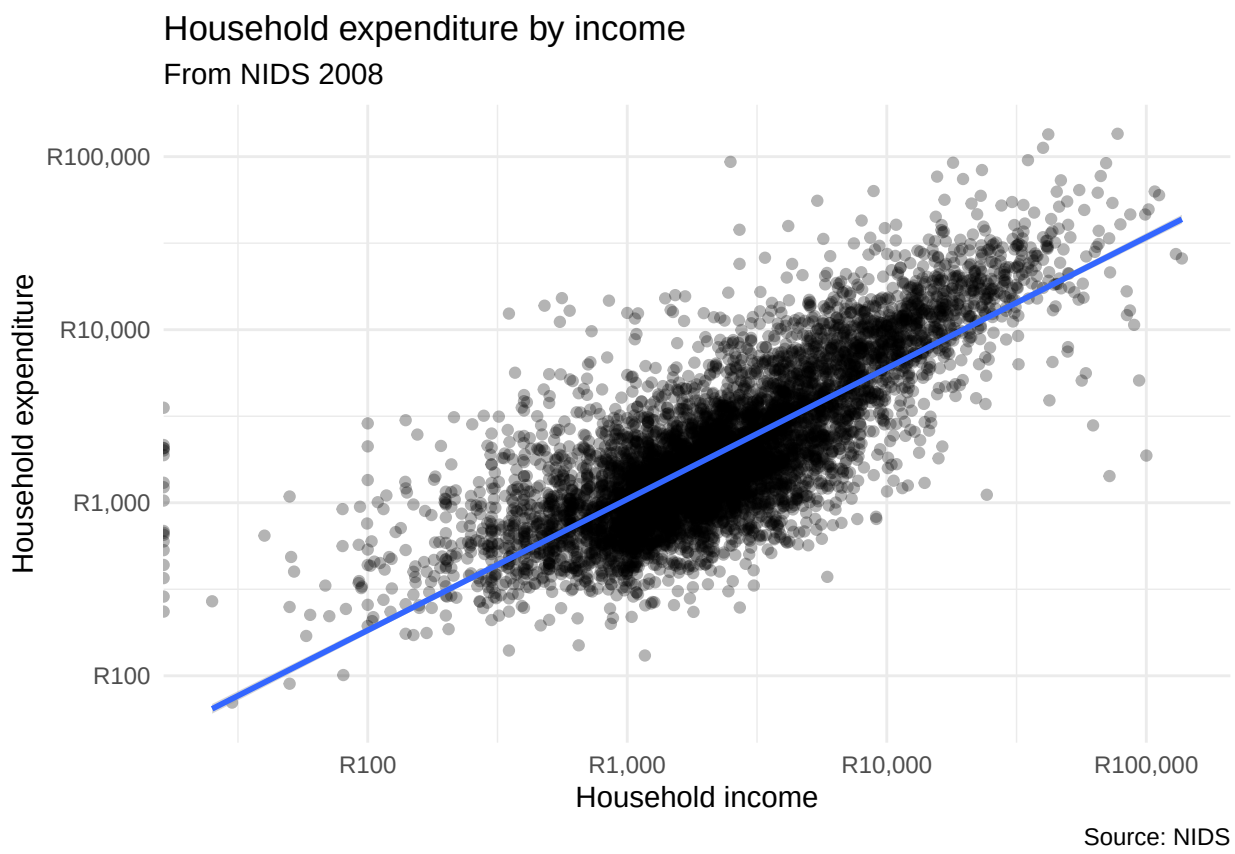
Example geometric shapes

- `ggplot2` supports a number of different types of geoms that include:
  - `geom_point` for drawing individual points (e.g., a scatter plot)
  - `geom_line` for drawing lines (e.g., for a line charts)
  - `geom_smooth` for drawing smoothed lines (e.g., for simple trends or approximations)
  - `geom_bar` for drawing bars (e.g., for bar charts)
  - `geom_histogram` for drawing binned values (e.g. a histogram)
  - `geom_polygon` for drawing arbitrary shapes
  - `geom_boxplot`: boxes-and-whiskers
  - `geom_errorbar`: T-shaped error bars

  - `geom_ribbon`: bands spanning y-values across a range of x-values

- Each of these geometries will leverage the aesthetic mappings supplied although the specific visual properties that the data will map to will vary.
- More customisation can be done to plot the variables on the log scale, add units, add title amd axis labels, and change the background theme.

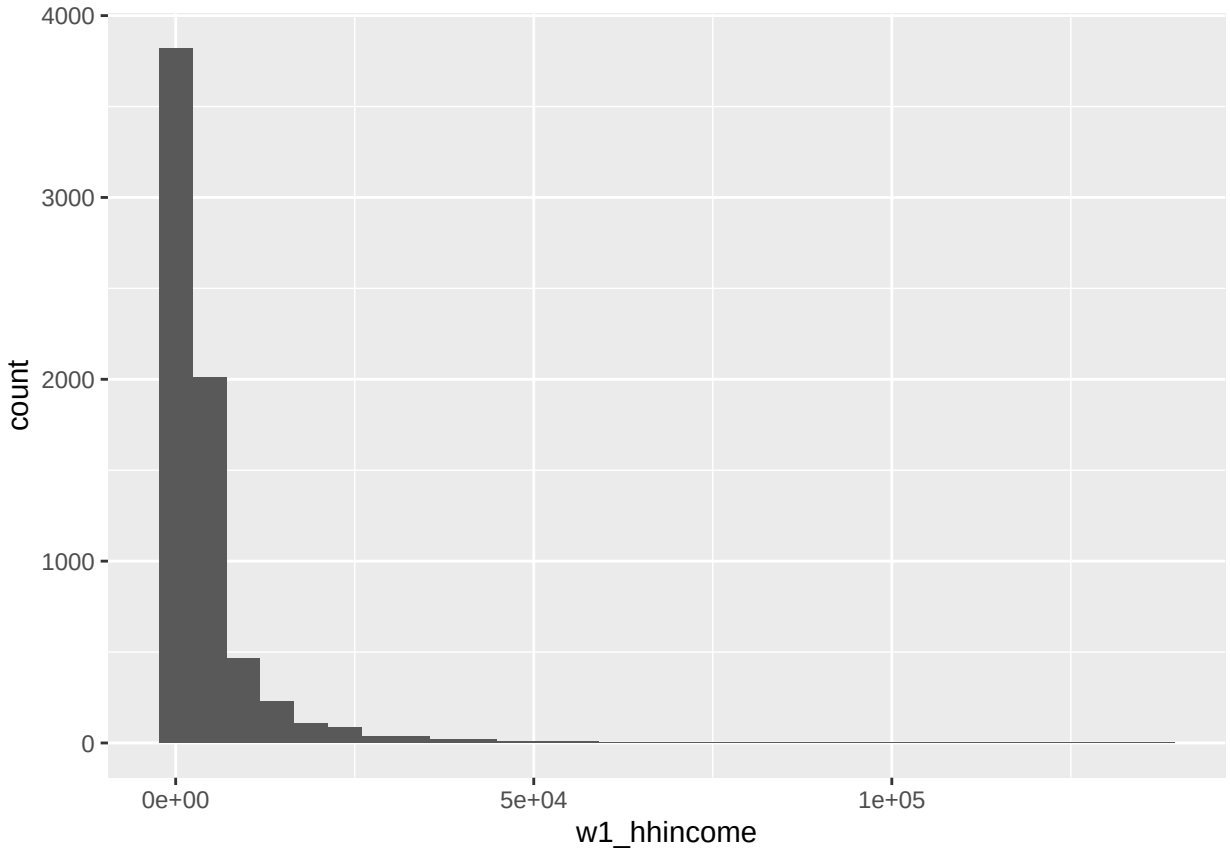
```
p <- ggplot(data = inc_exp, aes(x = w1_hhincome, y = w1_expenditure)) +
  geom_point(alpha=0.3) +  # transparency
  geom_smooth(method="lm") + # smooth distribution - linear model
  scale_x_log10(labels = scales::dollar_format(suffix = "", prefix = "R")) + #log scale
  scale_y_log10(labels = scales::dollar_format(suffix = "", prefix = "R")) + #log scale
  labs(title = "Household expenditure by income", #title
       subtitle = "From NIDS 2008", # subtitle
       x = "Household income", # x-axis title
       y = "Household expenditure", # y-axis title
       caption="Source: NIDS") + # caption
  theme_minimal() # change theme
p
```



Histogram

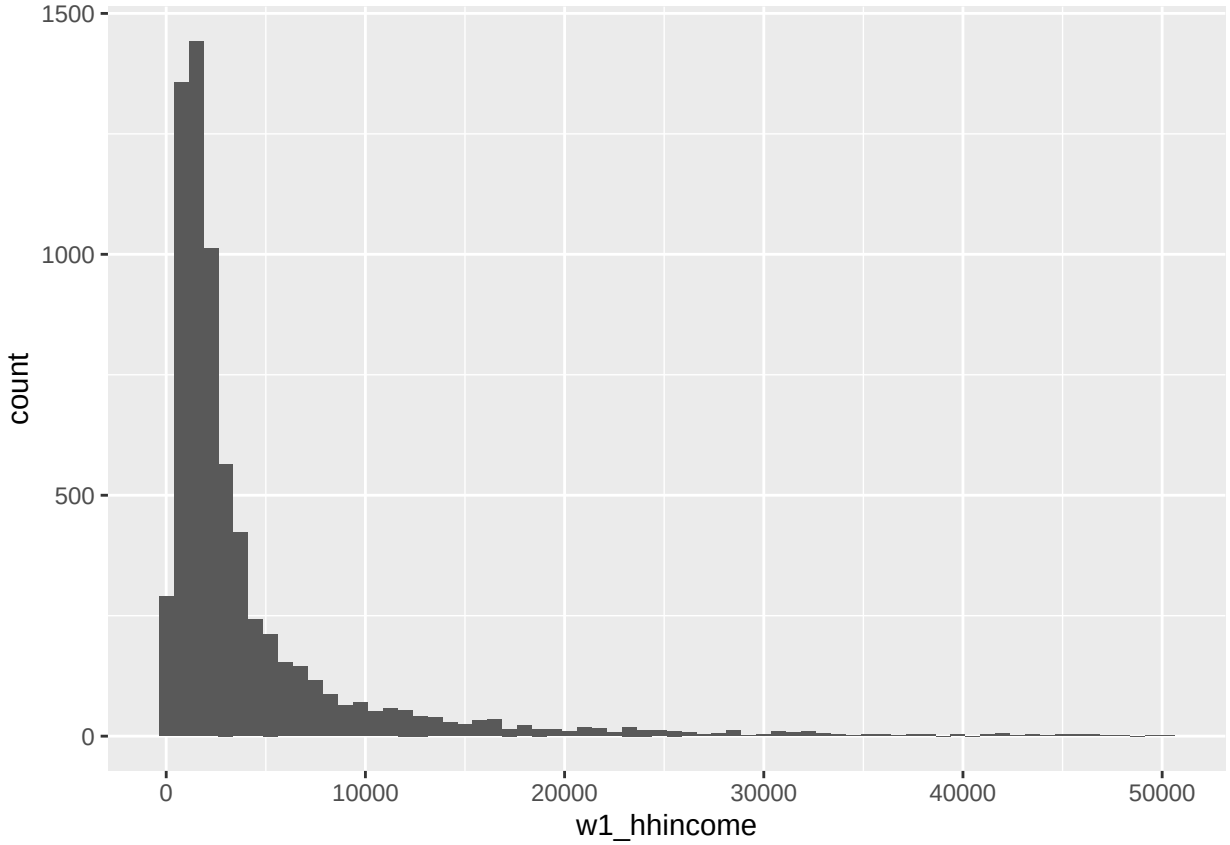
The `geom` for a histogram is `geom_histogram`. We again plot the histogram for household income:

```
ggplot(inc_exp, aes(w1_hhincome))+
  geom_histogram()
```



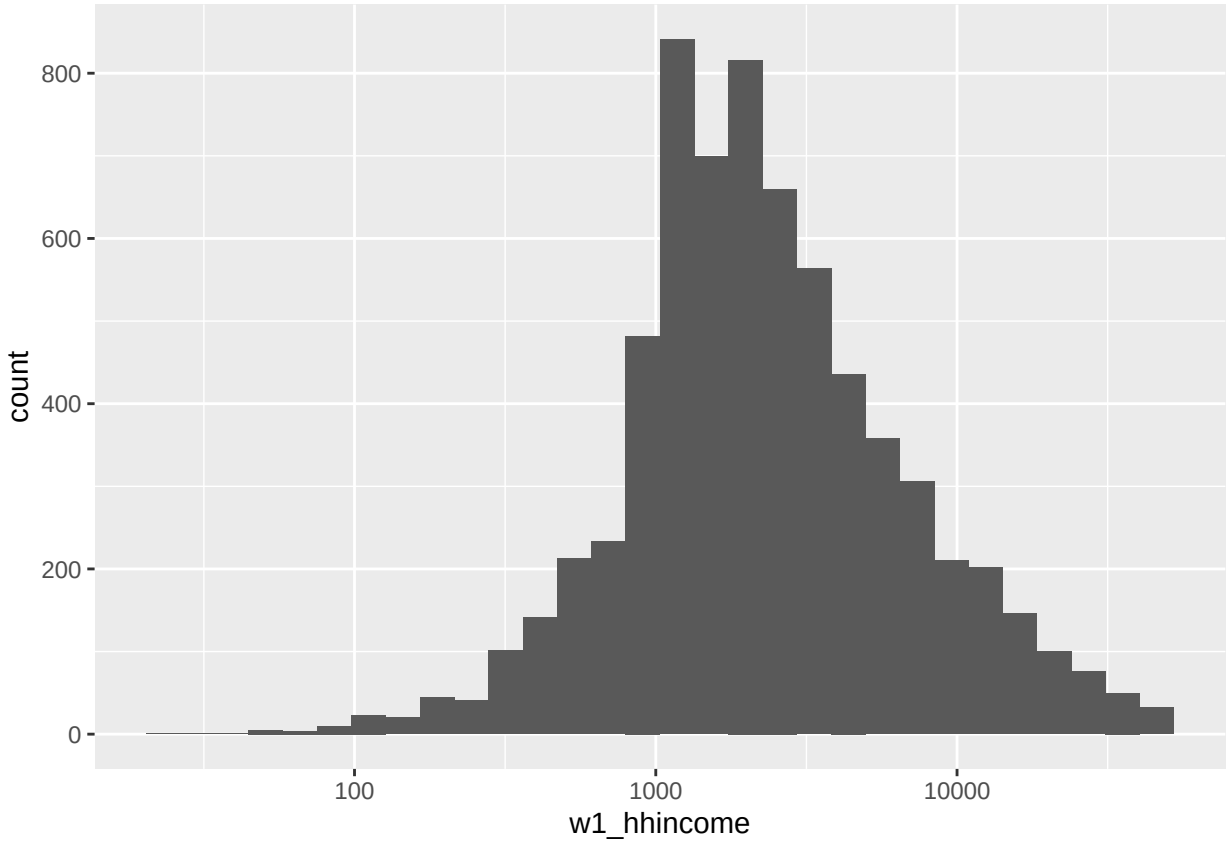
We can improve this by dropping some outlier observations

```
ggplot(inc_exp %>% filter(w1_hhincome<50000), aes(w1_hhincome))+
  geom_histogram(binwidth = 750)
```



Or plot on a log scale

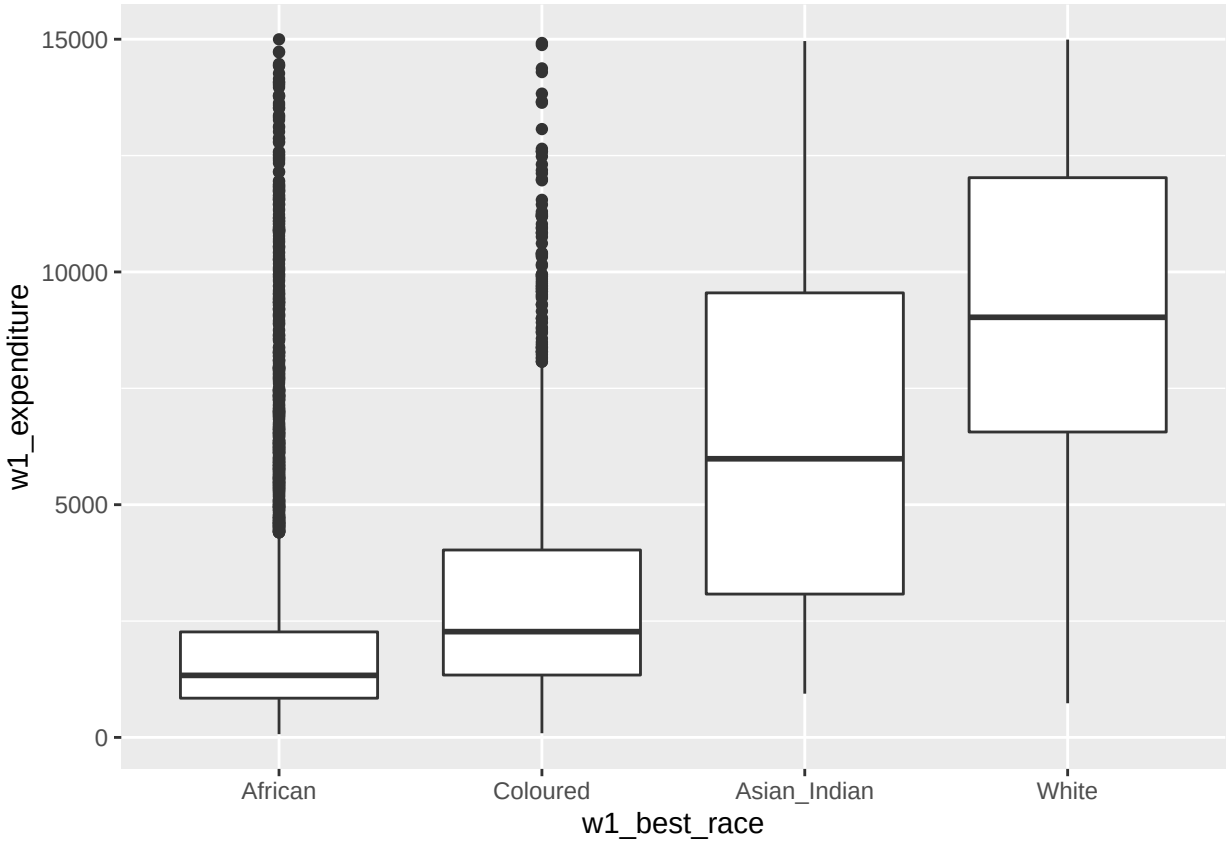
```
ggplot(inc_exp %>% filter(w1_hhincome<50000), aes(w1_hhincome))+
  geom_histogram()+
  scale_x_log10()
```



Boxplot

The `geom` for a boxplot is `geom_boxplot`. We again plot the boxplot for household expenditure by population group of the household head. This population group variable is in the `nids_df`.

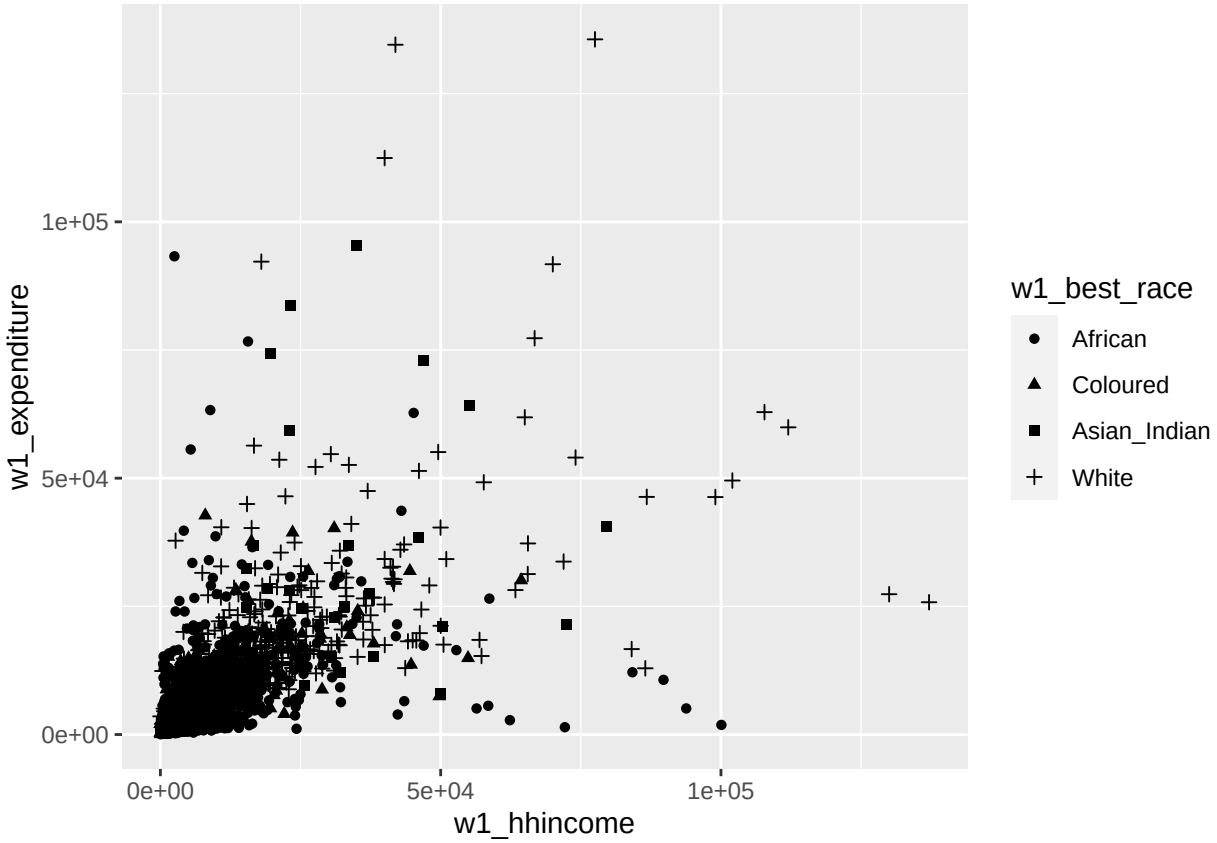
```
inc_exp %>% filter(w1_expenditure<15000) %>%
  ggplot(aes(x = w1_best_race, y = w1_expenditure))+
  geom_boxplot()
```



Aesthetics - shapes

We plot household expenditure against household income with different shapes by population group

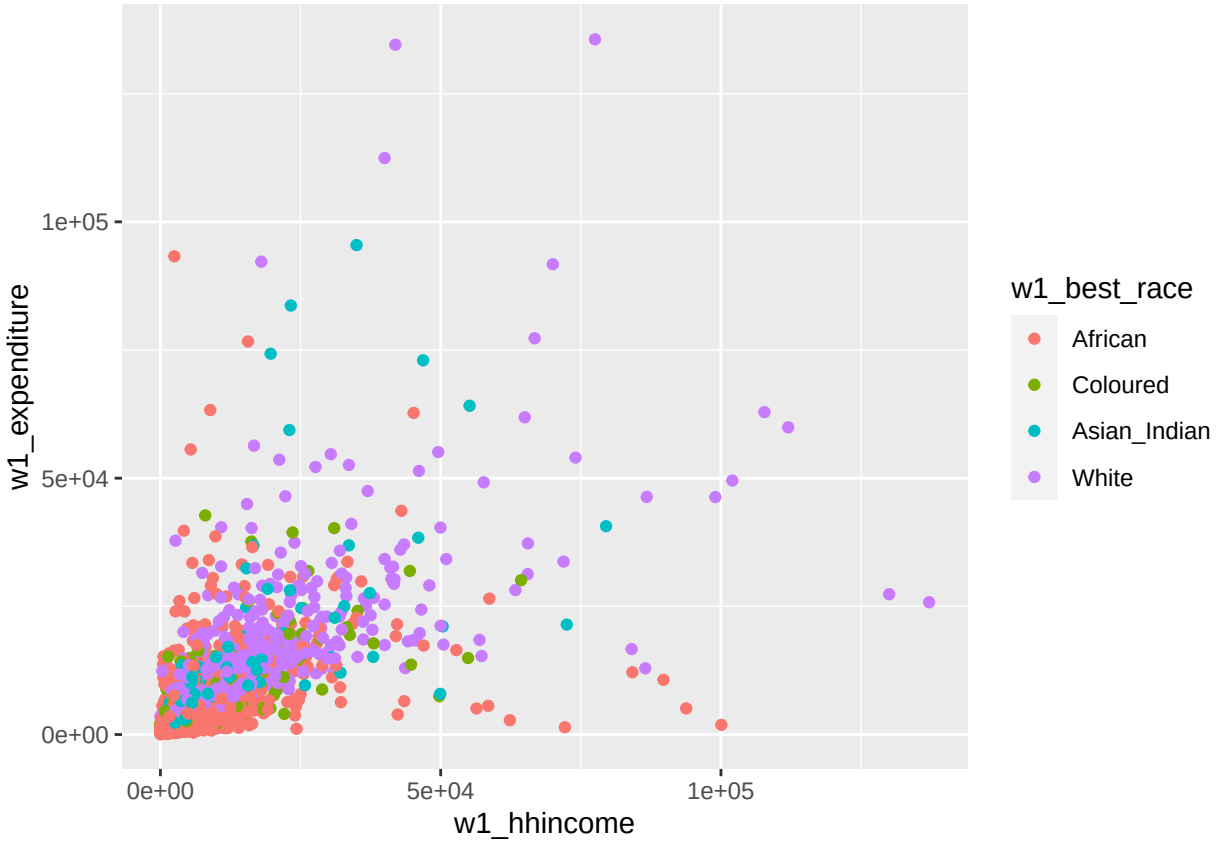
```
ggplot(data = inc_exp, aes(x=w1_hhincome, y=w1_expenditure, shape = w1_best_race)) +
  geom_point()
```



Aesthetics - color

We plot household expenditure against household income with different colour by race

```
ggplot(data = inc_exp, aes(x=w1_hhincome, y=w1_expenditure, color = w1_best_race )) +
  geom_point()
```



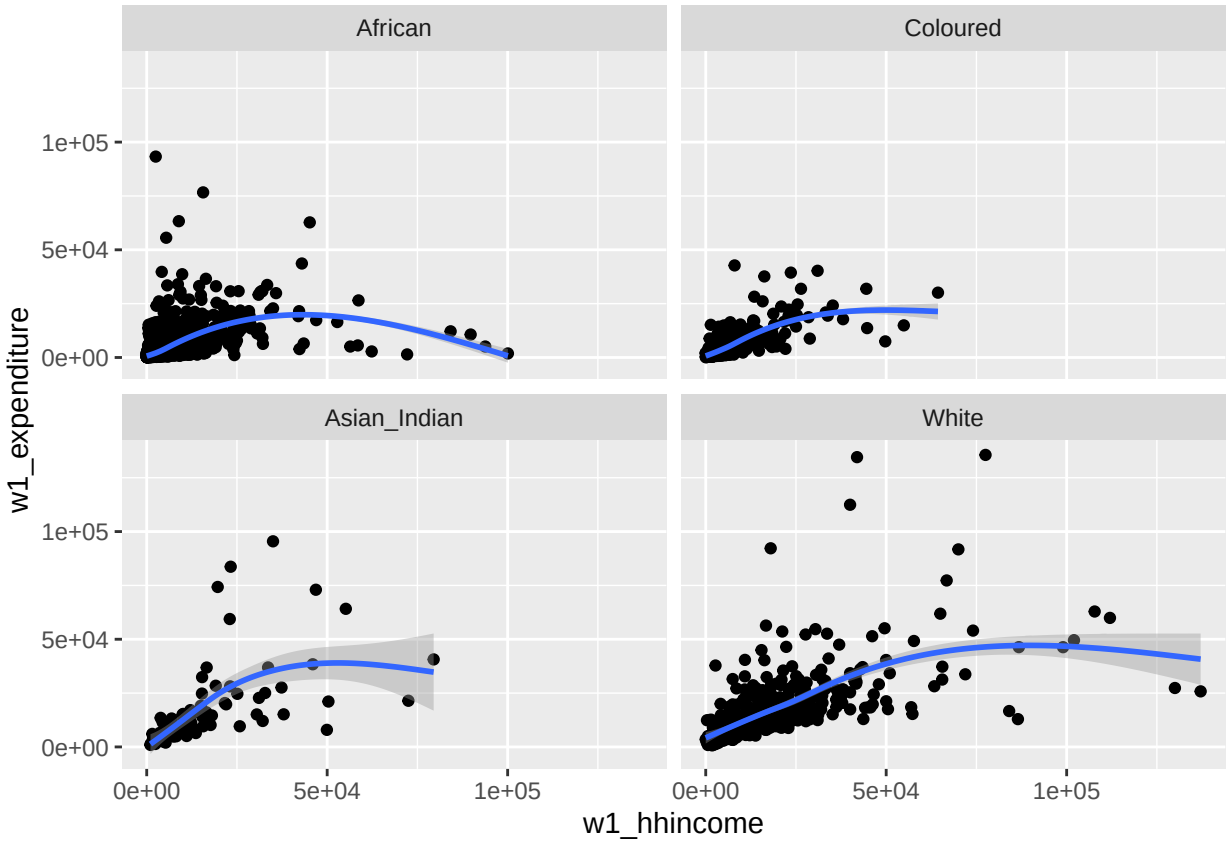
This also applies to shapes

Facet

- Split data into subsets and plot each subset on a different panel
- The faceting functions are:
  - `facet_wrap` - wraps a 1d sequence of panels into 2d and
  - `facet_grid` - forms a matrix of panels defined by row and column faceting variables.

facet_wrap

```
ggplot(inc_exp, aes(x = w1_hhincome, y=w1_expenditure)) +
  geom_point() +
  geom_smooth() +
  facet_wrap(~w1_best_race)
```



Any other graphics can similarly be plotted.

Saving plots to files

- use `ggsave()`.
- The last plot displayed is saved by default, but we can also save a plot stored to an R object.
- `ggsave` also guesses the type of graphics device from the extension. Available devices include `eps/ps`, `tex` (pictex), `pdf`, `jpeg`, `tiff`, `png`, `bmp`, `svg` and `wmf`. The size of the image can be specified as well.
- To save `p` to the `figures` subfolder, type the folowing:

```
#save last displayed plot as pdf
ggsave(plot = p, "./figures/incVexp.pdf")
#save plot stored in "p" as png and resize
ggsave(plot = p, "./figures/incVexp.png", width=10, height=10, units="in")
#?ggsave
```

Spatial data

Spatial data represents the location, size and shape of an object on the Earth.

Types of spatial data

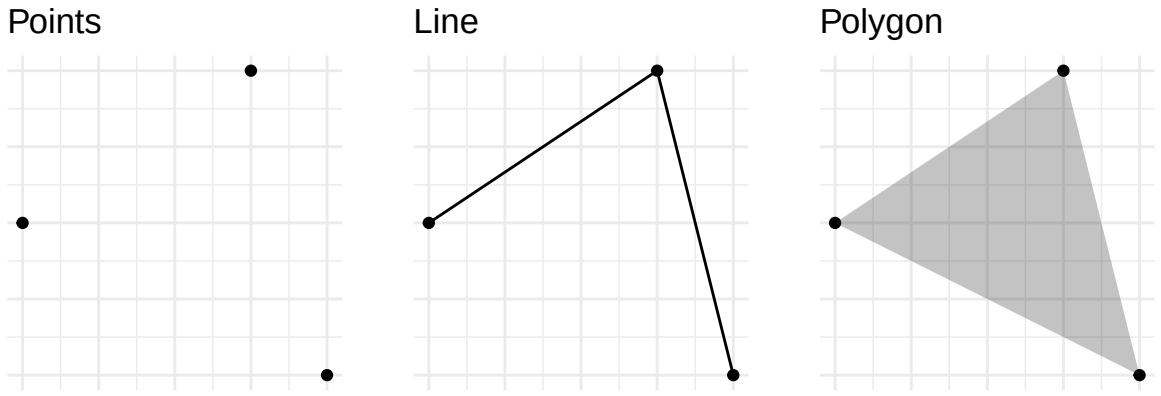
There are two fundamental models for representing geospatial data, i.e. **vector** and **raster**. **Vector data** is based on coordinates (e.g. longitude and latitude) of points on the earth's surface, which can be connected to make lines and polygons. **Rasta data** is based on pixels or gridded cells to represent surface features.

This short introduction introduces tools for handling **vector data** in R. Examples of vector data are:

Point(s) - discrete data points (e.g. house, point of interest, e.t.c.)

Line(s) - connected points (e.g. roads, electricity grid, etc)

Polygon(s) - closed boundaries (administrative city, lake, forest, etc)



Common vector spatial data formats

The following are some of the common formats in which spatial data is stored:

.shp (shapefile - at least 3 files - **.shp**, **.shx**, **.dbf** plus **.prj**, **.shp.xml**)

.geojson (web-based mapping)

.csv (general format that can be opened on most platforms)

Tools used to process spatial data

- ArcGIS (Esri)
- QGIS (Quantum GIS)

- GRASS GIS
- GeoDa
- ~~R~~

Why R for spatial data analysis

- R is **free and open source** which makes it cheaper to get started (how about other FOSS for GIS)
- The R command-line interface facilitates the **automation** of repetitive task (some of the GUI software share an API with python)
- The R command-line interface enables a **transparent** and **reproducible** workflow
- The command-line interface is also **flexible** to customize and extent the analysis to one's liking.
- R offers a large ecosystem of libraries/packages to perform various operations in the data analysis cycle (data access, data cleaning, data analysis, visualisation and report generation). Therefore, there is usually no need to do your analysis in one software and bring results into a GIS software.

Check the **Introduction** in Lovelace, Nowosad, and Muenchow (2019) for more on good reasons to use R for spatial data analysis.

Intergration with other GIS software

- It is possible to integrate R with other mature GIS software (most of these are GUI interfaces) and have access to functionality that is not yet available in R. Check Lovelace, Nowosad, and Muenchow (2019) (also freely available online on https://geocompr.robinlovelace.net/) on wrapper packages for QGIS, GRASS, SAGA and more, and the R – ArcGIS Community for packages to connect with ArcGIS.

The `sf` library

Simple features is a set of standards for geospatial data. The `sf` library (Pebesma 2018) is an implementation of simple features in R.

An `sf` object in R is based on the `data.frame` or tibble: it has multiple columns with different variables (often called attributes), as well as a geometry column, containing the spatial vector geometry. Each row represents one feature. It also contains information about the coordinate reference system used for the geometries. The `sf` package is compatible with `tidyverse` packages like `dplyr`, `ggplot`, e.t.c.

The `sf` package offers many methods and functions to operate on spatial data. Most `sf` function names, particularly those related to spatial data processing and analysis, start with `st_`, for *spatial type*. Typical application of the `sf` functions includes *reading, manipulating and writing of sets of features, with attributes and geometries*.

To access `sf` functions, we first need load the `sf` package into our workspace:

```
# load packages
library(sf)
```

- We import shapefiles using the `st_read` function or its alias, `read_sf`
- The layer specification of the `layer` argument varies by driver

- We import the SA province shapefile as follows:

```
sa_prov<-st_read(dsn="./PR_SA_2011", layer="PR_SA_2011") #no extension
```

```
## Reading layer `PR_SA_2011' from data source
##   `C:\Users\Alison\Desktop\Takwanisa stuff\r-courses\rprimer\PR_SA_2011'
##   using driver `ESRI Shapefile'
## Simple feature collection with 9 features and 7 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 16.45191 ymin: -34.83417 xmax: 32.94498 ymax: -22.12503
## Geodetic CRS:  WGS 84
```

```
#sa_prov<-st_read(dsn="./shapefiles/PR_SA_2011/PR_SA_2011.shp") #extension
```

```
class(sa_prov)
```

```
## [1] "sf"         "data.frame"
```

- `sa_prov` is of class `sf` and `data.frame`.

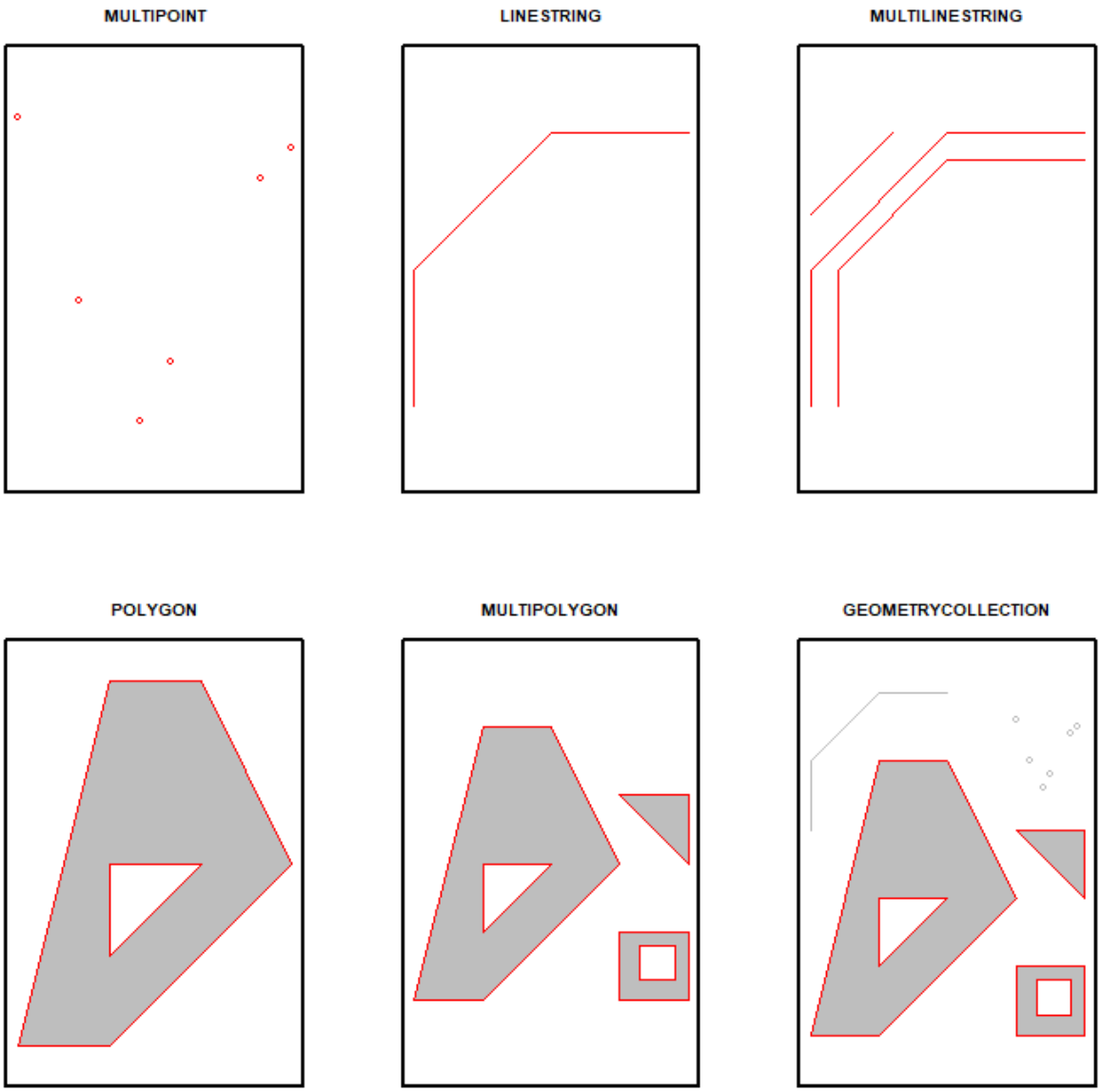
```
sa_prov
```

```
## Simple feature collection with 9 features and 7 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 16.45191 ymin: -34.83417 xmax: 32.94498 ymax: -22.12503
## Geodetic CRS:  WGS 84
##   PR_MDB_C PR_CODE PR_CODE_st       PR_NAME ALBERS_ARE SHAPE_Leng SHAPE_Area
## 1       EC       2          2  Eastern Cape  168965.97   28.20638  16.154667
## 2       FS       4          4    Free State  129825.16   24.24960  11.979428
## 3       GT       7          7       Gauteng   18178.31   10.96706   1.640382
## 4      KZN       5          5 KwaZulu-Natal   94361.32   19.02478   8.715379
## 5      LIM       9          9       Limpopo  125753.97   21.85956  11.141413
## 6       MP       8          8    Mpumalanga   76494.69   23.65078   6.889972
## 7       NW       6          6    North West  104881.68   26.09149   9.477825
## 8       NC       3          3 Northern Cape  372889.41   48.00840  34.691554
## 9       WC       1          1  Western Cape  129462.19   38.08598  12.494680
##                         geometry
## 1 MULTIPOLYGON (((29.02187 -3...
## 2 MULTIPOLYGON (((27.97704 -2...
## 3 MULTIPOLYGON (((28.75833 -2...
## 4 MULTIPOLYGON (((30.51695 -3...
## 5 MULTIPOLYGON (((29.65523 -2...
## 6 MULTIPOLYGON (((31.80952 -2...
## 7 MULTIPOLYGON (((26.4189 -24...
## 8 MULTIPOLYGON (((20.04457 -2...
## 9 MULTIPOLYGON (((18.15928 -3...
```

- the `sf` document describe the three classes used to represent simple features which are:

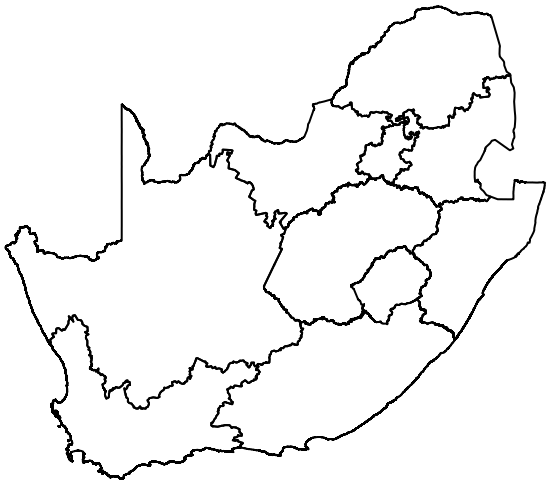
  - `sf`, the table (`data.frame`) with feature attributes and feature geometries, which contains
  - `sfc`, the list-column with the geometries for each feature (record), which is composed of
  - `sfg`, the feature geometry of an individual simple feature.

- `sf` has its own geometry types



- see also the `sf` website
- `sf` has its own plot function.
- We can use the `st_geometry` function to get the geometry from an `sf` object and plot:

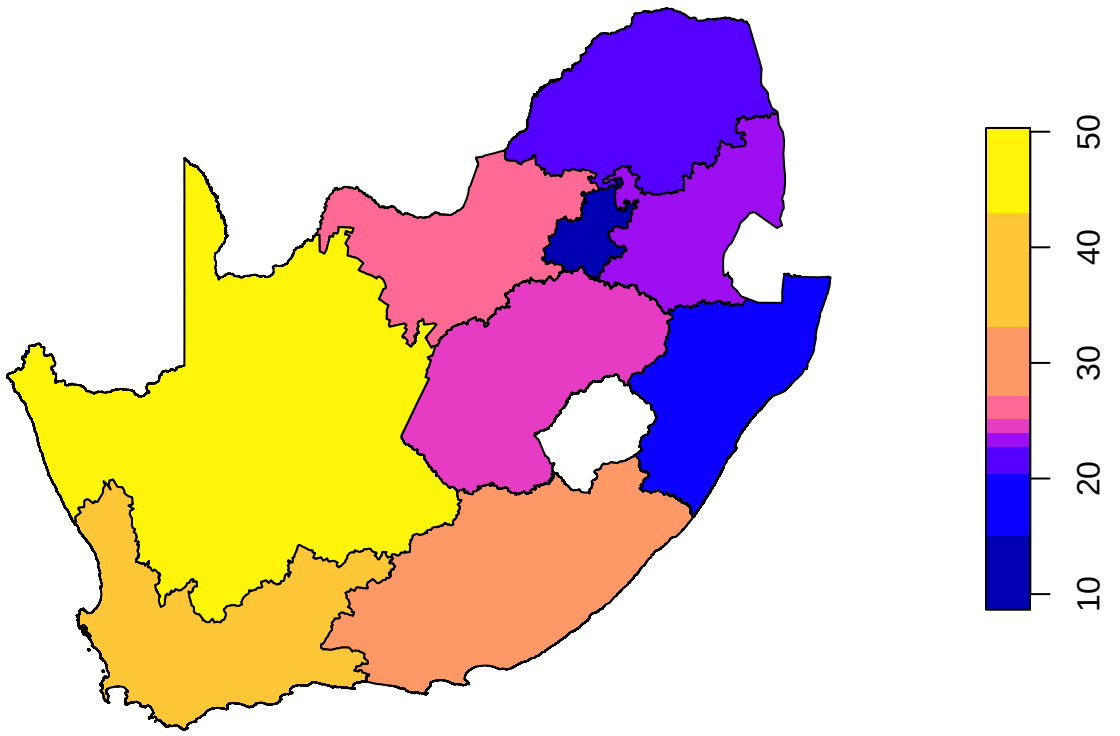
```
# pull just the geometry
sa_prov %>% st_geometry() %>% plot()
```



```
#plot(st_geometry(sa_prov))
```

Or you can plot a specific attribute, e.g., shapefile length:

```
plot(sa_prov["SHAPE_Leng"])
```



The plotting above was just a teaser, more advanced features will be covered tomorrow, and we end here.

Thank you

```
sessionInfo()
```

```
## R version 4.1.1 (2021-08-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19044)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_South Africa.1252  LC_CTYPE=English_South Africa.1252
## [3] LC_MONETARY=English_South Africa.1252 LC_NUMERIC=C
## [5] LC_TIME=English_South Africa.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] sf_1.0-8        patchwork_1.1.1 forcats_0.5.1   stringr_1.4.0
##  [5] purrr_0.3.4     tidyr_1.2.0     tibble_3.1.8    ggplot2_3.3.6
##  [9] tidyverse_1.3.2 dplyr_1.0.9     readr_2.1.2
##
## loaded via a namespace (and not attached):
```

```
##  [1] httr_1.4.3          bit64_4.0.5         vroom_1.5.7
##  [4] jsonlite_1.8.0      splines_4.1.1       modelr_0.1.8
##  [7] assertthat_0.2.1    highr_0.9           googlesheets4_1.0.0
## [10] cellranger_1.1.0    yaml_2.3.5          pillar_1.8.0
## [13] backports_1.4.1     lattice_0.20-44     glue_1.6.2
## [16] digest_0.6.29       rvest_1.0.2         colorspace_2.0-3
## [19] htmltools_0.5.3     Matrix_1.3-4        pkgconfig_2.0.3
## [22] broom_1.0.0         haven_2.5.0         scales_1.2.0
## [25] tzdb_0.3.0          proxy_0.4-27        googledrive_2.0.0
## [28] mgcv_1.8-36         generics_0.1.3      farver_2.1.1
## [31] ellipsis_0.3.2      withr_2.5.0         cli_3.3.0
## [34] magrittr_2.0.3      crayon_1.5.1        readxl_1.4.0
## [37] evaluate_0.15       fs_1.5.2            fansi_1.0.3
## [40] nlme_3.1-152        class_7.3-19        xml2_1.3.3
## [43] textshaping_0.3.6   tools_4.1.1         hms_1.1.1
## [46] gargle_1.2.0        lifecycle_1.0.1     munsell_0.5.0
## [49] reprex_2.0.1        e1071_1.7-11        compiler_4.1.1
## [52] systemfonts_1.0.4   rlang_1.0.4         units_0.8-0
## [55] classInt_0.4-7      grid_4.1.1          rstudioapi_0.13
## [58] labeling_0.4.2      rmarkdown_2.14      gtable_0.3.0
## [61] DBI_1.1.3           R6_2.5.1            lubridate_1.8.0
## [64] knitr_1.39          fastmap_1.1.0       bit_4.0.4
## [67] utf8_1.2.2          ragg_1.2.2          KernSmooth_2.23-20
## [70] stringi_1.7.8       parallel_4.1.1      Rcpp_1.0.9
## [73] vctrs_0.4.1         dbplyr_2.2.1        tidyselect_1.1.2
## [76] xfun_0.31
```

Further reading

1. Geocomputation with R (Lovelace, Nowosad, and Muenchow 2019): An excellent, free online resource on working with geospatial raster and vector data in R.

2. Spatial Data Science (Pebesma and Bivand 2022): Still work in progress but an excellent online book on working with spatial data in R.

References

Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2019. *Geocomputation with R*. CRC Press.

Pebesma, Edzer. 2018. "Simple Features for R: Standardized Support for Spatial Vector Data." *The R Journal* 10 (1): 439–46. https://doi.org/10.32614/RJ-2018-009.

Pebesma, Edzer, and Roger Bivand. 2022. *Spatial Data Science*.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer.

Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. " O'Reilly Media, Inc.".

Wilkinson, Leland. 2012. "The Grammar of Graphics." In *Handbook of Computational Statistics*, 375–414. Springer.