R for Household Consumption and Expenditure Surveys

Nutrition Data Analysis

Liberty Mlambo,Lucia Sergovia de la Revilla, Thomas Codd, Gareth Osman, Kevin T

4/10/24

Table of contents

Preface

About this manual

This manual is designed for absolute beginners who are interested in using R and RStudio for nutrition analysis of household consumption and expenditure surveys(HCES). The manual is an adaptation of materials for statistical analysis of HCES developed by the Micronutrient Action Policy Support(MAPS) Project and made available to all for use, without warranty or liability. The MAPS project is funded by the Bill and Melinda Gates Foundation.

Acknowledgements

We would like to thank the following people for their contributions to this manual:

Liberty Mlambo, Lucia Sergovia de la Revilla, Thomas Codd,Gareth Osman, Kevin Tang, Tineka Blake, Edward Joy, Louise E. Ander

Who is this manual for?

The goal of this manual is to provide a comprehensive introduction to these powerful technologies and to teach you how to use them to better understand your data and collaborate with others on your project.

Throughout this manual, you will learn how to install and set up R and RStudio on your computer, as well as how to use them to perform data analysis, create visualizations, and manage your code. The manual includes step-by-step instructions, examples, and practice exercises to help you master these technologies.

Whether you are a researcher, data scientist, or statistician, this manual will provide you with the skills and knowledge you need to start using R and RStudio for HCES analysis, to better understand your data and collaborate with others on your project.

It is important to note that this manual is not a comprehensive guide to R and RStudio but rather an introduction, designed to give you the foundational knowledge to start working with

these technologies. There are many other resources available for learning more about these technologies, including online tutorials, forums, and documentation.

We hope you find this manual helpful and that it empowers you to work with these powerful tools.

1 Introduction

Welcome to the training manual on using R, RStudio, Git, and GitHub for Household Consumption and Expenditure surveys. This manual is designed for absolute beginners and aims to provide a comprehensive introduction to these powerful technologies for use in Nutrition analysis.

1.1 Software requirements

First, we will cover R, which is a powerful and versatile programming language that is widely used for data analysis, statistical modeling, and data visualization.

- It is an open-source software that can be freely downloaded and used by anyone. R is widely used in academia, industry, and government, and is becoming increasingly popular among data scientists and analysts.
- It is a great tool for those who have been using other statistics tools like Excel, SAS, SPSS and want to take their data analysis skills to the next level.

This training will provide an introduction to the basics of R and will give you the skills you need to start working with data in R..

Next, we will introduce RStudio, which is a popular integrated development environment (IDE) for R.

- RStudio provides a user-friendly interface for working with R and makes it easy to work with R packages, which are collections of pre-written R code that can be used to perform specific tasks.
- With RStudio, you will be able to write, test, and debug your R code, and easily share your work with others.

This manual will provide step-by-step instructions for installing and setting up R and RStudio on your computer. We will also go over basic concepts and commands for working with each technology, as well as provide examples of how to use them in different contexts. With this manual, you will have the skills and knowledge you need to start using these powerful technologies to better understand your data and collaborate with others on your project.

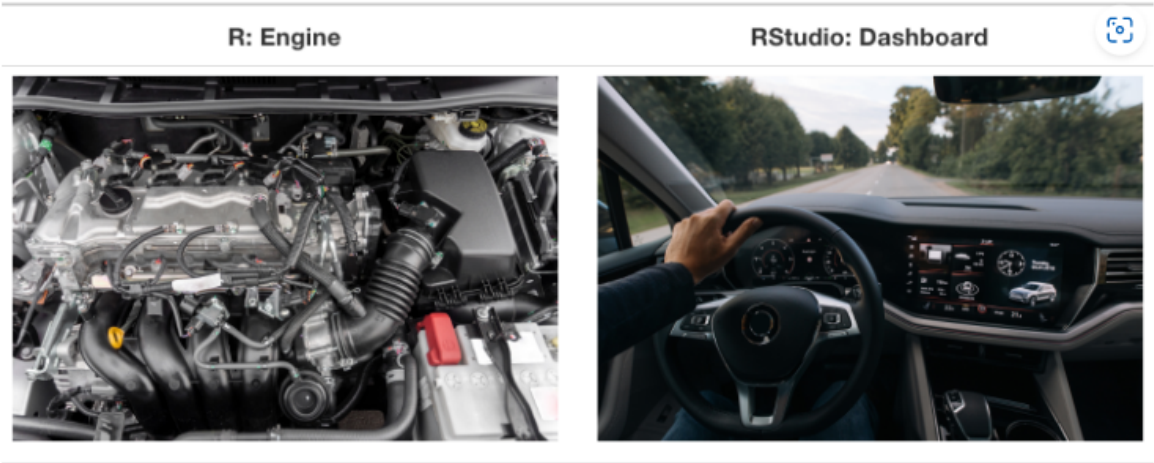
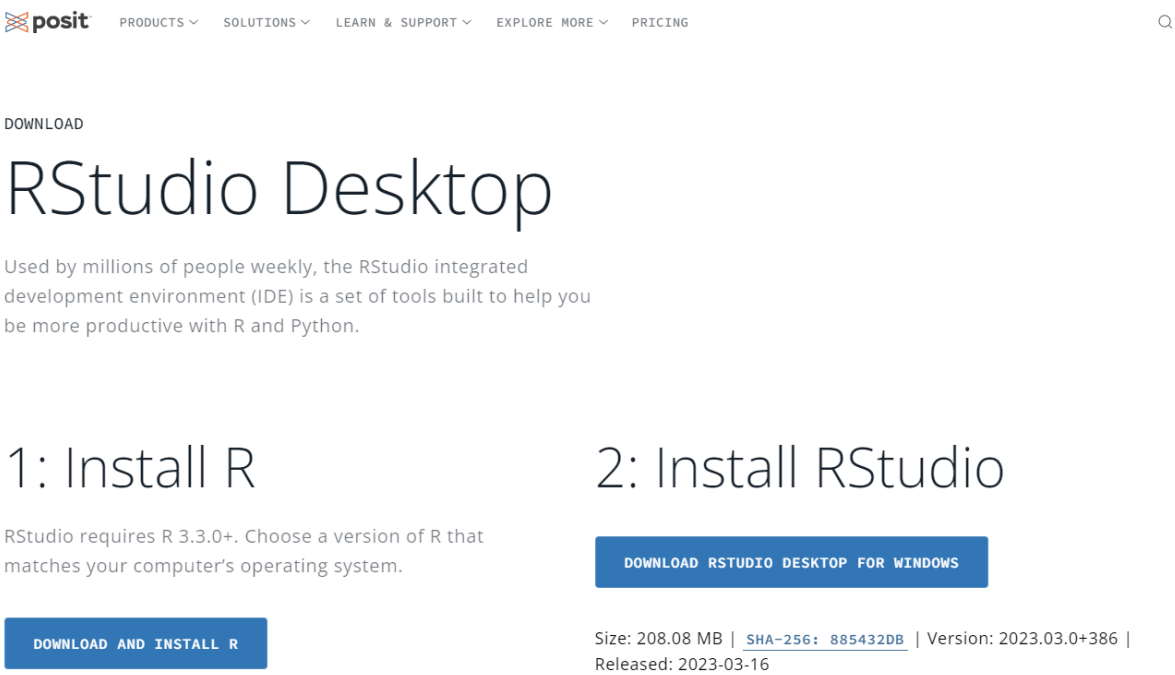
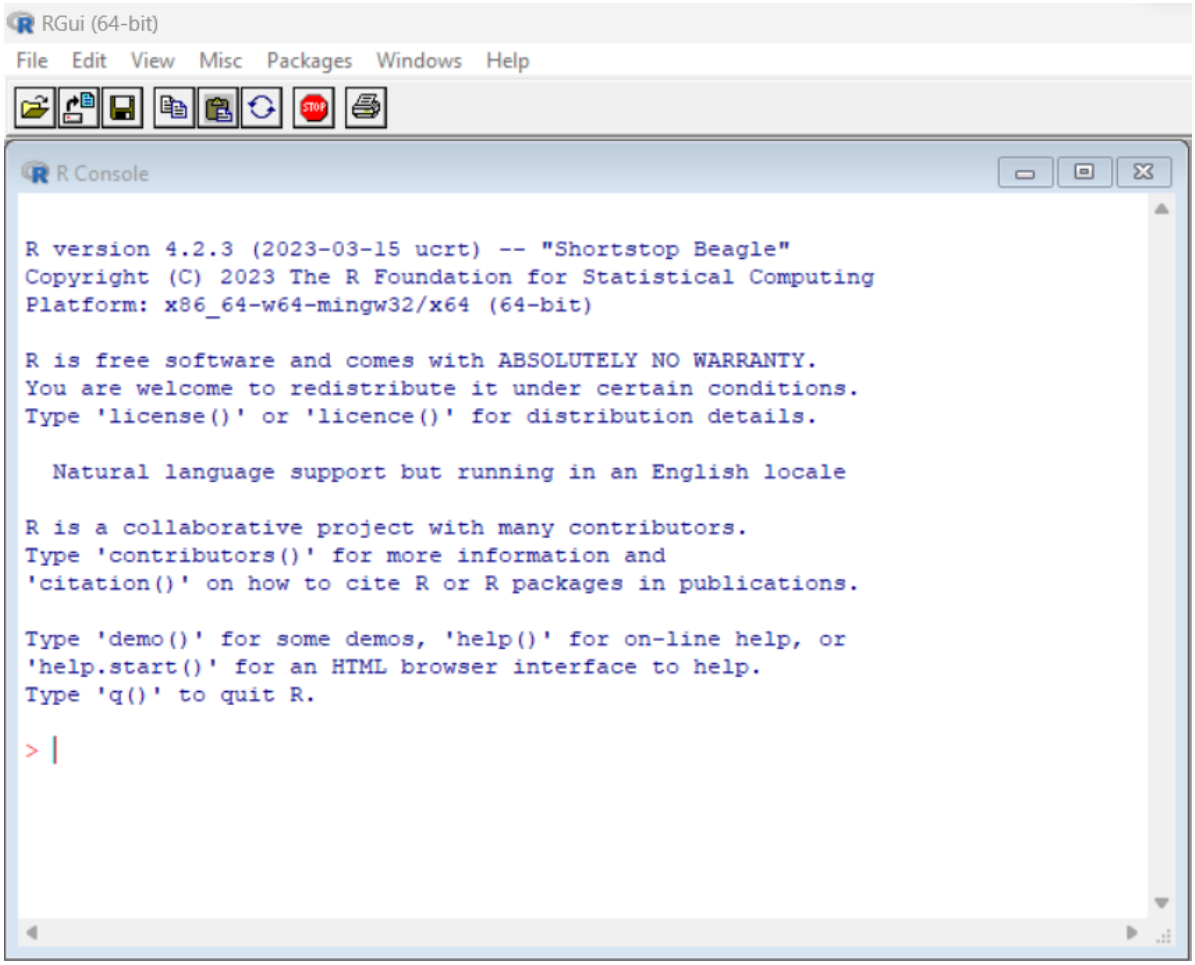


Figure 1.1: source: https://moderndive.netlify.app/1-getting-started.html

1.2 Downloading and Installing R and Rstudio



1.2.1 Downloading and Installing R



To download R, you can visit the official R website at https://cran.r-project.org/. On the website, you will see links to download the latest version of R for Windows, Mac, and Linux. Once you have downloaded the installer for your operating system, you can run the installer and follow the prompts to install R on your computer.

Downloading and installing R:

Instructions for downloading and Installing R

Step 1:

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Step 2:

Subdirectories:

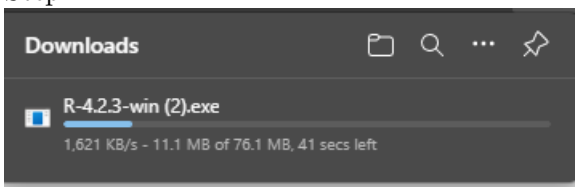
base	Binaries for base distribution. This is what you want to **install R for the first time**.
contrib	Binaries of contributed CRAN packages (for R >= 3.4.x).
old contrib	Binaries of contributed CRAN packages for outdated versions of R (for R < 3.4.x).
Rtools	Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Step 3:

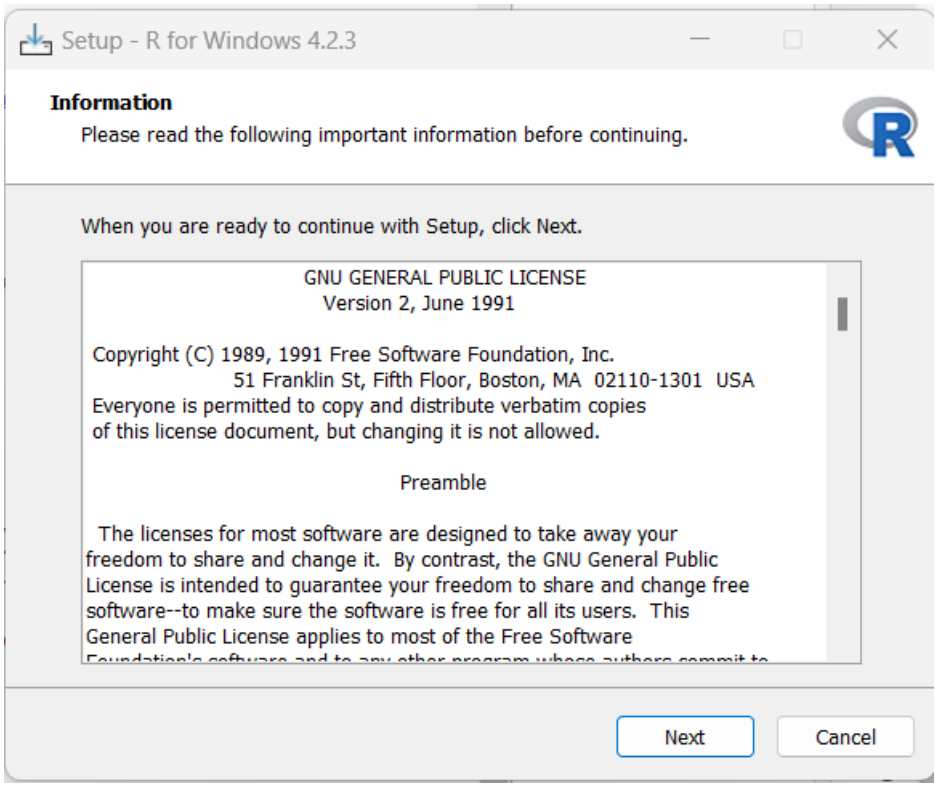
Download R-4.2.3 for Windows (77 megabytes, 64 bit)

README on the Windows binary distribution
New features in this version

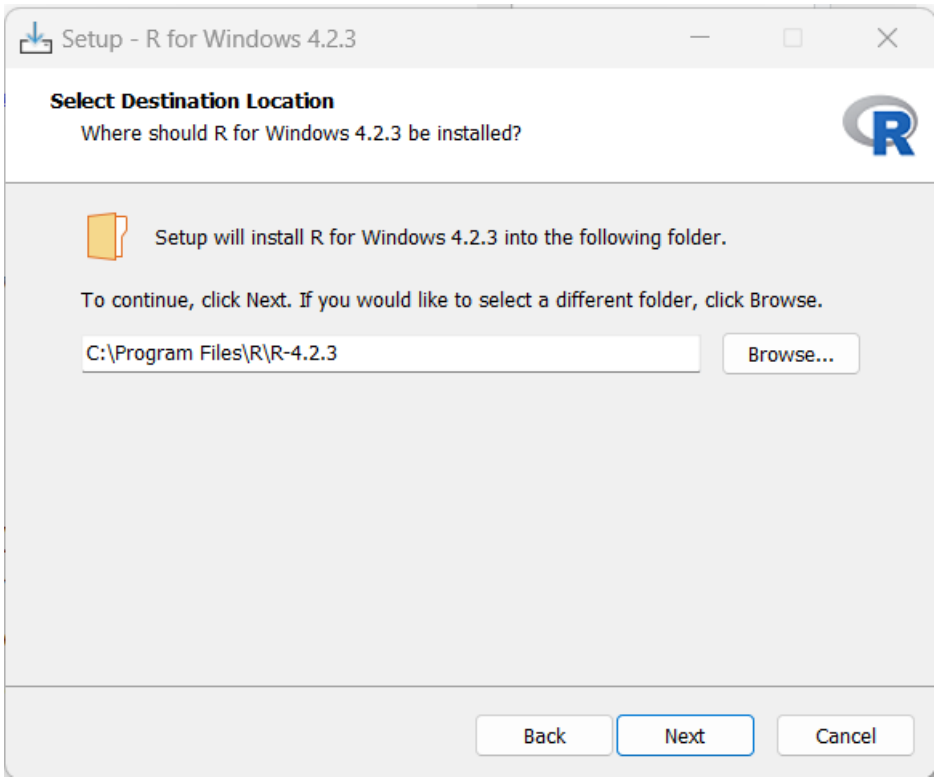
Step 4:



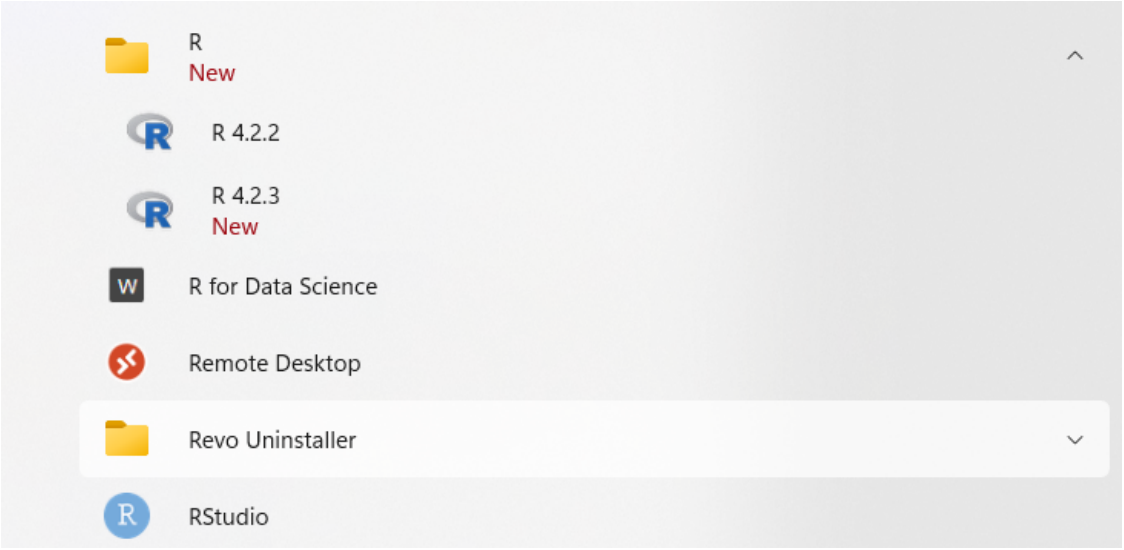
Step 5:


Setup - R for Windows 4.2.3
Information
Please read the following important information before continuing.
When you are ready to continue with Setup, click Next.
GNU GENERAL PUBLIC LICENSE
Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
Preamble
The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users. This
General Public License applies to most of the Free Software
Next
Cancel

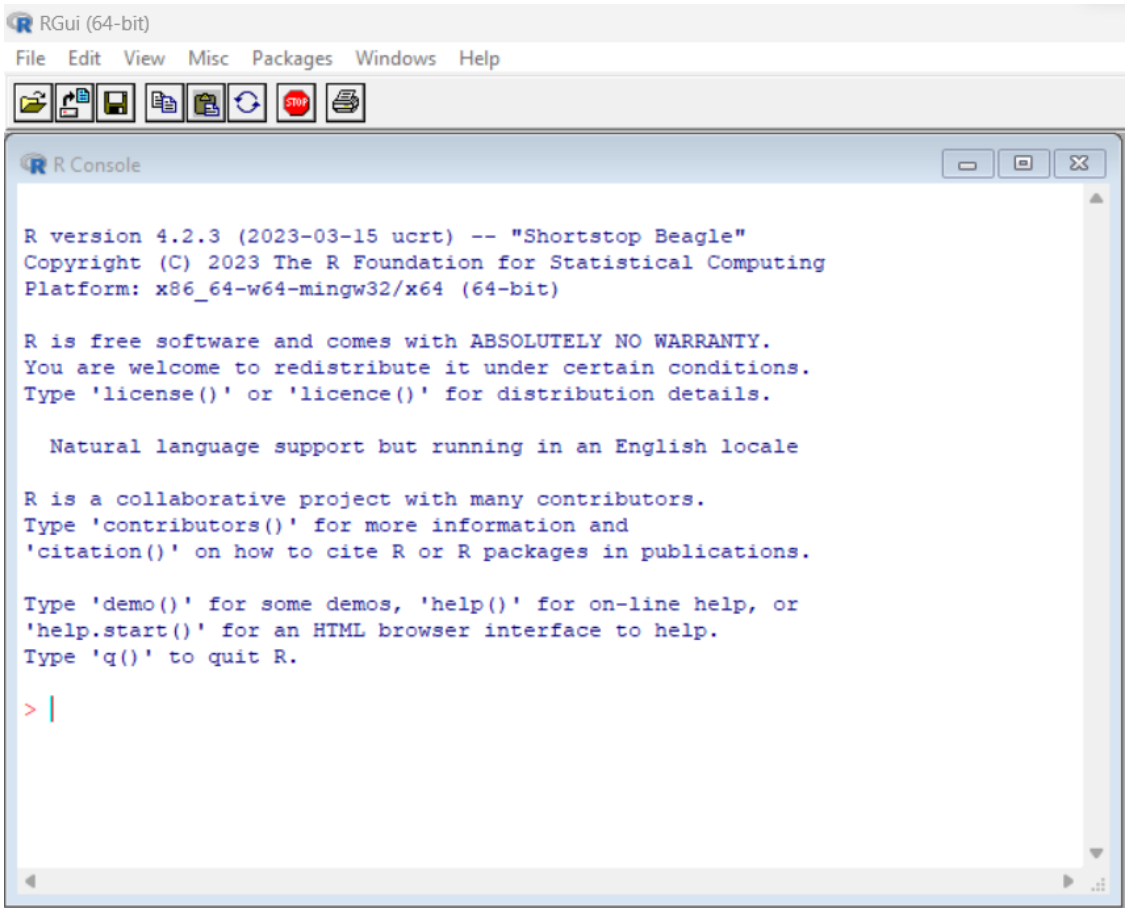
Step 6:


Setup - R for Windows 4.2.3
Select Destination Location
Where should R for Windows 4.2.3 be installed?
Setup will install R for Windows 4.2.3 into the following folder.
To continue, click Next. If you would like to select a different folder, click Browse.
C:\Program Files\R\R-4.2.3
Browse...
Back
Next
Cancel

Step 7:

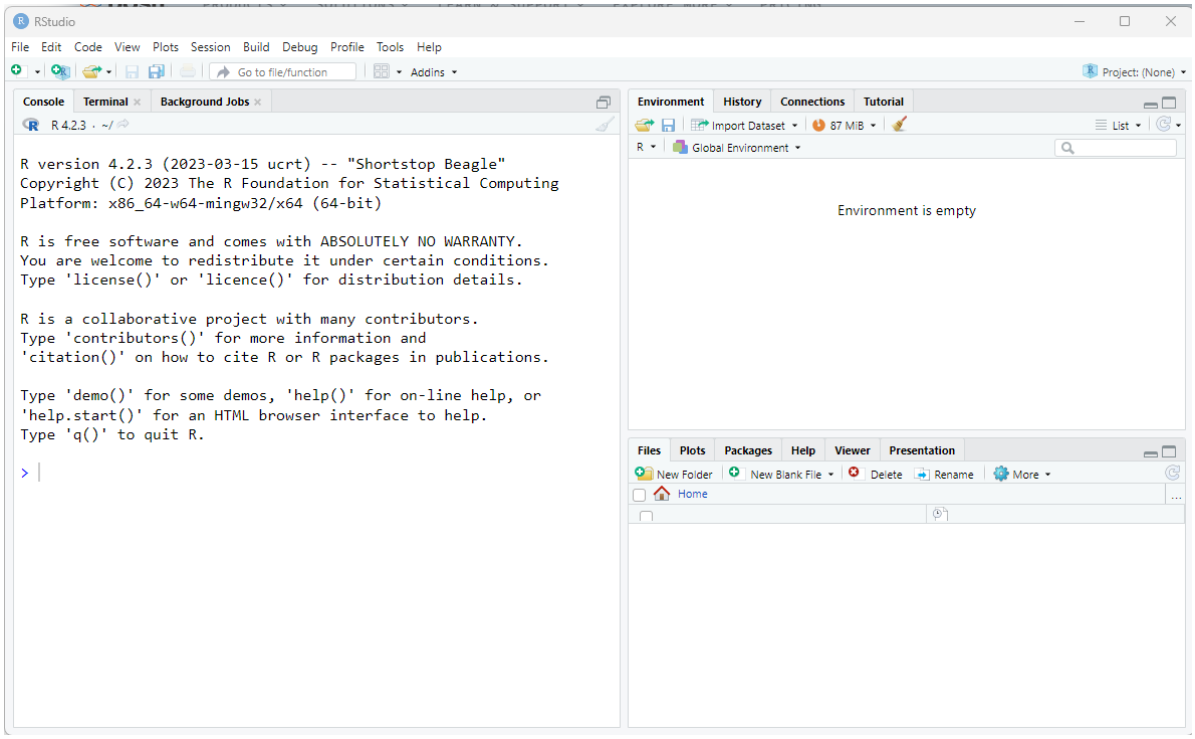

R
New
R 4.2.2
R 4.2.3
New
R for Data Science
Remote Desktop
Revo Uninstaller
RStudio

Step 8:



1.2.2 Downloading and Installing RStudio

To download RStudio, you can visit the official RStudio website at https://posit.co/download/rstudio-desktop/. On the website, you will see links to download the latest version of RStudio for Windows, Mac, and Linux. Once you have downloaded the installer for your operating system, you can run the installer and follow the prompts to install RStudio on your computer.



Instructions for downloading and Installing Rstudio

Step 1: Navigate to https://posit.co/download/rstudio-desktop/

posit PRODUCTS SOLUTIONS LEARN & SUPPORT EXPLORE MORE PRICING

DOWNLOAD

RStudio Desktop

Used by millions of people weekly, the RStudio integrated development environment (IDE) is a set of tools built to help you be more productive with R and Python.

1: Install R

RStudio requires R 3.3.0+. Choose a version of R that matches your computer's operating system.

DOWNLOAD AND INSTALL R

2: Install RStudio

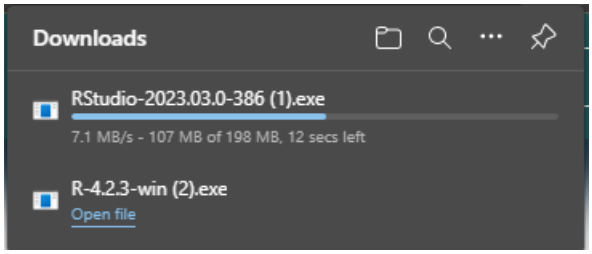
DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS

Size: 208.08 MB | SHA-256: 885432DB | Version: 2023.03.0+386 | Released: 2023-03-16

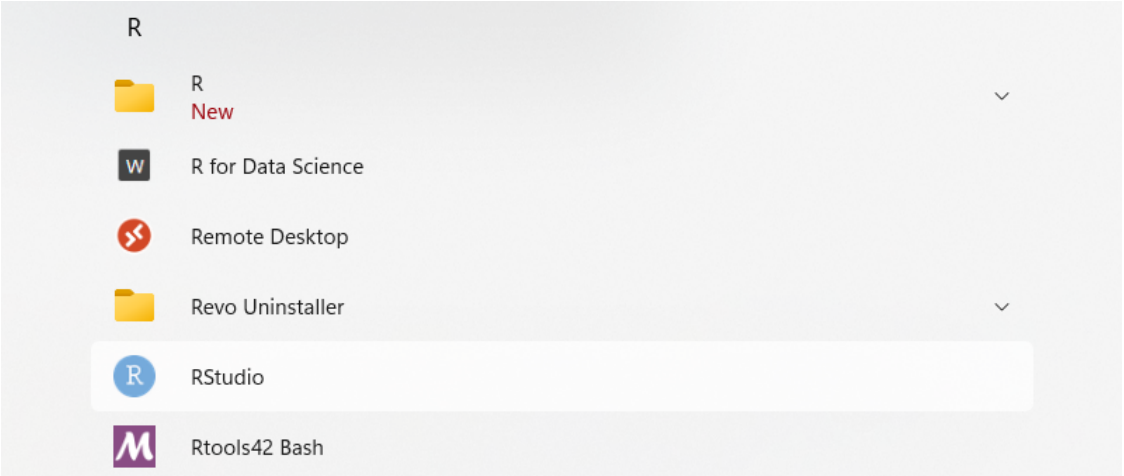
Step 2:

OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2023.03.0-386.EXE	208.08 MB	885432DB
macOS 11+	RSTUDIO-2023.03.0-386.DMG	374.55 MB	ED87B818
Ubuntu 18+/Debian 10+	RSTUDIO-2023.03.0-386-AMD64.DEB	137.78 MB	D71B670E
Ubuntu 22	RSTUDIO-2023.03.0-386-AMD64.DEB	138.28 MB	0A347709

Step 3:



Step 4:



Please note that these are general instructions for a Microsoft Windows operating system, and depending on your system setup and security settings, some steps might be slightly different. Also, you will need to make sure that you have administrative access or permission to install the software on your computer.

You can also refer to the software website instruction or online tutorials that are specific to your operating system and setup.

From here we will use the term R to refer to R and Rstudio or vice-versa.

1.3 Recommended setup while using this book

Step 1: Download the training files from the following link: https://dzvoti.github.io/r4hces/r4hces-data.zip

Step 2: Unzip the file and save it in a folder on your computer.

Step 3: Open RStudio create a new project using an existing folder. Select the folder where you saved the training files.

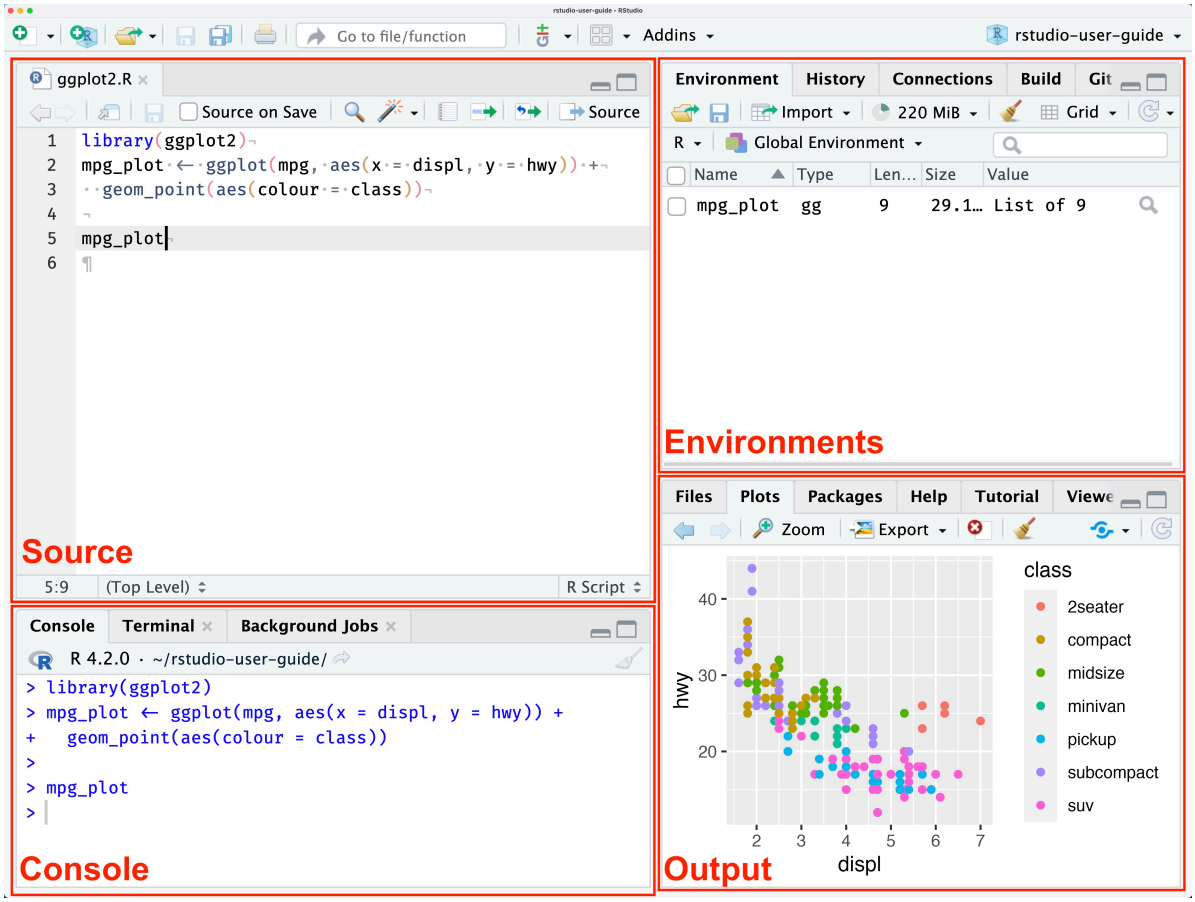


Figure 1.2: Source: https://docs.posit.co/ide/user/ide/guide/ui/ui-panes.html

2 Data Types

The principal data types in R are `numeric`, `character`, `factor` and `logical`. There are others, but these are the main ones.

- A datum of type `numeric` is a numerical value, such as food quantity value.
- A datum of type `character` is a string of characters, such as the name of food.
- A datum of type `factor` is the label for a food type or categories which we might use in hces analysis.

- A datum of type `logical` takes values `TRUE` or `FALSE`

2.1 Assignment

We can store any of these datatypes in an `object` by assigning the value to that object. For example, we can assign the value `Maize` to the object `food_name` as follows:

```
food_name <- "Maize"
```

The `<-` is the assignment operator. It assigns the value on the right to the object on the left. We can then use the object `food_name` in other commands, for example, to print the value of `food_name` we can use the `print()` function:

> `<-`
>
> If you are using RStudio, you can type `<-` by pressing the `Alt` key and `-` key at the same time.

```
print(food_name)
```

There are other assignment operators, such as `=` and `->`, but `<-` is the most common. We can also assign the value of an object to another object, for example:

```
food_name2 <- food_name
```

In this case, the value of `food_name` is assigned to `food_name2`. We can then print the value of `food_name2`:

```
print(food_name2)
```

In this book we will use the `<-` and the `=` assignment operator. We use the `<-` when we want to assign a value to an object, and the `=` when we want to assign a value to an argument in a function. This is a convention that is used by many R programmers. More on functions later.

2.2 Character data

The simplest data type in R is the character. A character is a string of characters, for example, the "Maize" name that we assigned above. The " " indicate that we want to store the string of characters between the " " in the object. If we don't use the " " then R will look for an object with that name, and if it doesn't find it, it will throw an error. For example, if we type:

```
food_name <- Maize
```

We can fix this by putting the " " around the string of characters:

```
food_name <- "Maize"
```

We ca perform operations on character data, such as concatenation, which is the joining of two or more strings of characters. We can do this using the `paste()` function. For example, we can create a new character object called `food_name3` by concatenating the values of `food_name` and `food_name2` as follows:

```
# Create character vector with value "Maize"
food_name <- "Maize"
# Create character vector with value "Meal"
food_name2 <- "Meal"
# Concatenate the values of food_name and food_name2 and assign the result to a new charac
food_name3 <- paste(food_name, food_name2)
# Print the value of food_name3
print(food_name3)
```

2.3 Numeric data

A numeric is a numerical value, such as the food quantity value. We can assign a numeric value to an object as follows:

```
food_quantity <- 0.5
```

Note that we don't need to put the " " around the numeric value. If we do, then R will treat it as a character, and not a numeric. For example, if we type:

```
food_quantity <- "0.5"
```

We can then do simple mathematical manipulations with a numeric value. For example, we can add 0.5 to the value of `food_quantity` as follows:

```
food_quantity <- 0.5
# Add 0.5 to the value of food_quantity
food_quantity <- food_quantity + 0.5
```

Exercise

1. Create a character object called `food_name` and assign it the value `"Maize"`.
2. Create another character object called `food_subname` and assign it the value `"Meal"`.
3. Concatenate the values of `food_name` and `food_subname` and assign the result to a new character object called `full_name`.
4. Create a numeric object called `food_quantity_g` and assign it the value `15`.
5. Convert the value of `food_quantity_g` to milligrams and assign the result to a new numeric object called `food_quantity_mg`.

2.3.1 Operations on numeric data

We can perform operations on numeric data, such as addition, subtraction, multiplication and division. For example, we can create a new numeric object called `food_quantity` by adding the values of `food_quantity_g` and `food_quantity_mg` as follows:

```
# Create a numeric object called food_quantity_g and assign it the value 15
food_quantity_g <- 15

# Create a numeric object called food_quantity_mg and
# calculate the value of food_quantity_g in milligrams
```

```
food_quantity_mg <- food_quantity_g * 1000
```

Just like in maths the operators in R follow operator precedence.However we can use brackets to specify the order of operations.

2.4 Logical data

Logical data takes the values `TRUE` or `FALSE`. We can assign a logical value to an object as follows:

```
is_staple <- TRUE
```

Logical values can be returned from operation e.g. testing for equality. For example, we can test whether the vales in two objects is the same as follow:

```
# Create character vectors
food_name <- "Maize"
food_name2 <- "Maize"
food_name3 <- "Rice"

# Test equality
food_name == food_name2
food_name == food_name3
```

Notice how when testing for equality we use `==`? This is because the `=` is an assignment operator and not a logical operator. We can also use the `!=` operator to test for inequality. For example, we can test whether the values in two objects are not the same as follows:

```
food_name != food_name2
food_name != food_name3
```

Other logical operators are `>`, `<`, `>=` and `<=`. Logical object can be the subject of logical functions, notably `"if .. then"`. Consider the example below:

```
# Create numeric object
age <- 18
# Test whether age is greater than 18
if(age > 18) {
    print("You are an adult")
    }else{
```

```
        print("You are not an adult")
        }
```

Testing the same example with a different value of age:

```
# Create numeric object
age <- 17
# Test whether age is greater than 18
if(age > 18) {
    print("You are an adult")
    }else{
        print("You are not an adult")
        }
```

Logical operations can be chained together using the `&` operator for `"and"` and the `|` operator for `"or"`. For example, we can test whether the values in two objects are the same and whether the value of `food_quantity_g` is greater than 10 as follows:

```
food_name == food_name2 & food_quantity_g > 10
```

2.5 Summary

Until now we have been storing only one value in an object. We can store multiple values in an object using a vector. We will look at vectors and data structures in the next section.

3 Data Structures

A data structure in R is an R object which holds one or more data objects, a data object will be a data type, such as we have encountered in section 1 (numeric, character, etc). In this script we introduce vectors, factors, matrices, data frames and lists. The examples and exercises should help you to understandbetter how R holds and manages data.

3.1 Vectors

A vector is a series of homogeneous values of a variable (e.g. Foods from an HCES survey). The easiest way to form a vector of values in R is with the `"combine"` function `c()`. An example of a vector of character values (food_names) is shown below:

```
# Create a vector of character values
food_names <-
    c("Rice",
      "Maize",
      "Beans",
      "Cassava",
      "Potatoes",
      "Sweet potatoes",
      "Wheat")

#Create a vector of numeric values
consumption <- c(0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01)

# Create a vector of logical values
is_staple <- c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)

# Create a vector of mixed values
mixture <- c(5.2, TRUE, "CA")
```

Excercise

Use print to see the values of the vectors above e.g `print(food_names)` What happens if you try to print the vector `mixture`?

Tip

We can count the number of items in a vector with the `length()` function:

```
length(food_names)
```

Each item in a vector can be referenced by its index (i.e. its position in the sequence of values), and we can pull out a particular item using the square brackets after the vector name. For example, the `3rd` item in `food_names` can be accessed like this

```
food_names[3]
```

3.2 data frames vs tibbles

In R, data frames and tibbles are two common data structures used to store tabular data. While they are similar in many ways, there are some important differences to keep in mind.

3.2.1 Data Frames

Data frames are a built-in R data structure that is used to store tabular data. They are similar to matrices, but with the added ability to store columns of different data types. Data frames are created using the `data.frame()` function, and can be manipulated using a variety of built-in R functions.

3.2.2 Tibbles

Tibbles are a newer data structure that were introduced as part of the `tidyverse` package. They are similar to data frames, but with some important differences. Tibbles are created using the `tibble()` function, and can also be manipulated using a variety of built-in tidyverse functions.

One of the main differences between data frames and tibbles is how they handle column names. In a data frame, column names are stored as a character vector, and can be accessed using the

$ operator. In a tibble, column names are stored as a special type of object called a `quosure`, which allows for more flexible and consistent handling of column names.

Another difference between data frames and tibbles is how they handle subsetting. In a data frame, subsetting using the `[ ]` operator can sometimes lead to unexpected results, especially when subsetting a single column. In a tibble, subsetting is more consistent and predictable, and is done using the `[[ ]]` operator or with user friendly `dplyr` function e.g. `filter`, `select`.

Overall, while data frames and tibbles are similar in many ways, tibbles offer some important advantages over data frames, especially when working with the tidyverse package.

Let us make a data frame using the `data.frame()` function. We will use the vectors we created above as the columns of the data frame. Note that the vectors must be of the same length, otherwise the data frame will be filled with `NA` values to make up the difference.

```
# Create a data frame
food_df <-
    data.frame(
        food_names = c(
            "Rice",
            "Maize",
            "Beans",
            "Cassava",
            "Potatoes",
            "Maize",
            "Wheat"
        ),
        consumption = c(0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01),
        is_staple = c(TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE),
        stringsAsFactors = TRUE
    )

# Print the data frame
print(food_df)
```

Let us make a tibble using the `tibble()` function. We will use the vectors we created above as the columns of the tibble. Note that the vectors must be of the same length, otherwise the tibble will be filled with `NA` values to make up the difference.

```
# Create a tibble
food_tb <- tibble::tibble(
    food_names = c(
        "Rice",
```

```
        "Maize",
        "Beans",
        "Cassava",
        "Potatoes",
        "Maize",
        "Wheat"
    ),
    consumption = c(0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01),
    is_staple = c(TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
)

# Print the tibble
print(food_tb)
```

Excercise

Use the `class()` function to check the class of the `food_df` and `food_tb` objects. 1. What did you notice? 2. What is the difference between the two objects? Guess the data structure of each object. 3. Did you notice how the vector names were used as column names in the data frame and tibble?

3.3 Factors

Note that a `factor` is actually a vector, but with an associated list of `levels`, always presented in alpha-numeric order. These are used by `R` functions such as `lm()` which does linear modelling, such as the analysis of variance. We shall see how factors can be used in the later section on data frames.

Let us create a factor from a vector of character values. We can do this using the `factor()` function. The first argument is the vector of character values, and the second is the list of levels. If we don't specify the levels, `R` will use the unique values in the vector, in alphabetical order.

3.3.1 Coercing a vector to a factor

Example of converting the `food_names` vector to a factor:

```
# Create a factor without providing the levels argument
food_names_factor_1 <- factor(food_names)
# Print the factor
```

```r
print(food_names_factor_1)

# Create a factor from a vector of character values
food_names_factor_2 <-
    factor(
        food_names,
        levels = c(
            "Rice",
            "Maize",
            "Beans",
            "Cassava",
            "Potatoes",
            "Sweet potatoes",
            "Wheat"
        )
    )

# Print the factor
print(food_names_factor_2)
```

Excercise

1. What is the difference between the two factors?
2. Create a factor from the `is_staple` vector. What are the levels?
3. Create a factor from the `consumption` vector. What are the levels?

3.3.2 Coercing a vector to a factor in a data frame

Example of converting the `food_names` vector to a factor in a data frame:

```r
library(dplyr)
# Use the food_tb data frame created above and convert the food_names column to a factor
food_tb <- mutate(food_tb, food_names = factor(food_names))

# Print the data frame
print(food_tb)
```

3.4 Summary

There are other data structures in R, e.g. Matrix and lists but these are the most common. We will now look at some of the operations we can perform on vectors and data frames in the future sections.

But first,we introduced the `dplyr` package above. This is a `package` which provides a set of functions for manipulating data frames. We will use it extensively in this book. We can use the `mutate()` function to add a new column to a data frame. In this case we are adding a new column called `food_names` which is a factor version of the `food_names` column in the data frame.This means we introduced a new `function` `mutate()` and a new `package` `dplyr`.

In the next section we define what are `packages` and `functions`.

4 Packages and Functions

4.1 Functions

Functions are a set of instructions that can be called by name. They are useful for automating repetitive tasks and for encapsulating complex tasks.

Functions are defined using the `function()` function. An example of a function is the `dataframe` function which we used above to create a dataframe.

A function takes in one or more arguments and returns a value. The arguments are specified in the function definition and the value is returned using the `return()` function.

To see the arguments of a function, use the `?` before the function name e.g. `?dataframe`. To see the code of a function, just type the function name without the parentheses.

For example, to see the code of the `dataframe` function, type `dataframe` without the parentheses. Other examples are `head` , `str` and `summary` functions.

4.1.1 Creating a function

The basic syntax for defining a function is as follows:

```
# Define a function
function_name <- function(arg1, arg2, ...) {
    # Function body
    # ...
    # Return value
    return(return_value)
}
```

For example, let us create a function called `add` that takes in two arguments and returns the sum of the two arguments.

```
# Define a function
add <- function(x, y) {
    # Return the sum of the two arguments
```

```r
    return(x + y)
}

# Call the function
add(2, 3)
```

```
[1] 5
```

4.2 Packages

A package is a collection of functions, data, and documentation that extends the functionality of R. There are thousands of packages available for R.

To use a package, you first need to install it using the `install.packages()` function. Once installed, you can load the package using the `library()` function.

For example, to install the `dplyr` package, you would type `install.packages("dplyr")`.

To load the `dplyr` package, you would type `library(dplyr)`. To see the functions in a package, type `help(package = "package_name")` e.g. `help(package = "dplyr")`.

To see the code of a function in a package, type `package_name::function_name` e.g. `dplyr::mutate`. You can also use the `?` before the function name e.g. `?dplyr::mutate`.

4.2.1 Package sources

There are three main sources of packages for R:

- CRAN - The Comprehensive R Archive Network: https://cran.r-project.org/. This is the main source of packages for R. It contains over 15,000 packages. To install a package from CRAN, you can use the `install.packages()` function.
- GitHub: Most developers store their packages on GitHub. To install a package from GitHub, you can use the `install_github()` function from the `devtools` package. e.g. `devtools::install_github("dzvoti/hcesNutR")`.

> Tip
>
> Notice how we used package_name::function_name to call the `install_github()` function.

4.2.2 Loading packages

Once installed a package need to be 'loaded' for its function to be available in R. This is done using the `library()` function.

For example, to load the `dplyr` package, you would type `require(dplyr)`. The difference between the two functions is that `library()` will throw an error if the package is not installed, while `require()` will throw a warning. You can also use the `::` operator to call a function from a package without loading the package.

Also,to call the `mutate()` function from the `dplyr` package without loading the package, you would type `dplyr::mutate()`. This is useful when you want to use a function from a package without loading the package.

4.2.3 Removing packages

To remove a package, you can use the `remove.packages()` function. For example, to remove the `dplyr` package, you would type `remove.packages("dplyr")`.

4.2.4 Updating packages

To update a package, you can use the `update.packages()` function. For example, to update the `dplyr` package, you would type `update.packages("dplyr")`.

4.2.5 Listing installed packages

To list all installed packages, you can use the `installed.packages()` function. For example, to list all installed packages, you would type `installed.packages()`.

4.2.6 Recomended packages

There are thousands of packages available for R. However, there are some packages that are recommended for beginners. These include:

Table 4.1: Recommended Packages {.striped .hover}

Package	Description
`tidyverse`	A collection of packages designed for data science. It includes the: `dplyr`, `ggplot2`, `tidyr`, `readr`, `purrr`, `tibble`, `stringr`, `forcats` and `haven` packages.

Package	Description
`here`	A package for managing file paths.

Now that we know what packages and functions are, let us look at some of the functions we can use to manipulate vectors and dataframes in the next section on Data Import,Wrangling and Export.

5 Data I/O and Wrangling

Up to this point we were creating data to manipulate in R. In this section we will learn how to import data into R, manipulate it and export it. We will use the `readr` and `haven` packages to import data into R, the `dplyr` package to manipulate data and the `readr` package to export data from R. We will also use the `here` package to manage file paths.

5.1 Data Input/Import

There are many ways to import data into R. In this section we will look at how to import data from a CSV file, an Excel file and Stata file. In the sample data folder there are `*.csv` files and stata files `*.dta`. Foe example to import the health data from a survey stored in `hh_mod_a_filt_vMAPS.dta` stored in the `mwi-ihs5-sample-data` folder within our working directory we run:

```
# Load the haven package
library(haven)

# Import roster data
ihs5_roster <- read_dta(here::here("data","mwi-ihs5-sample-data", "hh_mod_a_filt_vMAPS.dta

# Preview the data
head(ihs5_roster )
```

To read a csv file we use the `read_csv()` function from the `readr` package. For example, to import the `IHS5_UNIT_CONVERSION_FACTORS_vMAPS.csv` file stored in the `mwi-ihs5-sample-data` folder within our working directory we run:

```
# Load the readr package
library(readr)

# Import unit conversion factors data
ihs5_unit_conversion_factors <- read_csv(here::here("data","mwi-ihs5-sample-data", "IHS5_U

# Preview the data
```

```r
head(ihs5_unit_conversion_factors)
```

Excercise

1. Import your own excel file into R.

Tip

Notice how all import operations are done within the `here::here()` function. This is because we are using the `here` package to manage file paths. The `here::here()` function returns the path to the file relative to the working directory. This is useful when you want to share your code with others, as they can run the code without having to change the file paths.
It is very import that file names and directories are typed as they are. R is sensitive to capital letters and spaces. For example, if you type `IHS5_UNIT_CONVERSION_FACTORS_VMAPS.csv` instead of `IHS5_UNIT_CONVERSION_FACTORS_vMAPS.csv`, R will throw an error. To get around this in RStudio use the `tab` key to autocomplete file names and directories.

After importing files they are usually stored in memory as dataframes/tibbles. We can check the class of an object using the `class()` function. For example, to check the class of the `ihs5_roster` object, we would type `class(ihs5_roster)`. We can also check the structure of an object using the `str()` function. For example, to check the structure of the `ihs5_roster` object, we would type `str(ihs5_roster)`. We want to make sure that the data is imported correctly before we start manipulating it.

5.2 Data Wrangling

The `dplyr` package from the `tidyverse` package is our data wrangling tool of choice. It provides a set of functions for manipulating dataframes e.g. renaming columns, conditional removal of rows, creation of other columns and so on. We will load and manipulate the `consumption` module of our hypothetical Malawi IHS5 survey data. The data is stored in the `mwi-ihs5-sample-data` folder within our working directory and is called `HH_MOD_G1_vMAPS.dta`. We will use the `here` package to manage file paths.

```r
library(dplyr) # data manipulation
library(haven) # data import
library(here) # file paths
```

5.2.1 Import the data

```
# Import the data
ihs5_consumption <- read_dta(here::here("data","mwi-ihs5-sample-data", "HH_MOD_G1_vMAPS.dt
```

Excercise

1. Check if the data improrted correctly
2. Check the structure of the data
3. How many observations and variables are there?

5.2.2 Subsetting data

5.2.3 Subsetting data frames

There are a number of functions that can be used to extract subsets of R objects in tidyverse syntax. The most important are the following from the `dplyr` package:

- filter() allows you to select a subset of rows in a data frame.
- select() allows you to select a subset of columns in a data frame.
- arrange() allows you to reorder the rows of a data frame.
- mutate() allows you to create new columns from existing columns.
- summarise() allows you to collapse many values down to a single summary.
- pull() allows you to extract a single column from a data frame as a vector.

5.2.4 Subsetting columns

This data that we loaded is a randomly generated immitation of the Malawi Intergrated Household Survey 2018-2019 described here. This data contains responses on `total consumption` as well as disaggregation of the sources of these foods.In this book we will process only the 'total consumption.

Remember we said that our data is loaded in memory? Seeing that the `ihs5_consumption` data contains columns we do not need let us subset it. The `select` function in dplyr is very useful for this. For example to keep only the columns with household identifiers and food names, units and quantity of consumption we keep the following columns in our data; "case_id", "HHID", "hh_g01", "hh_g01_oth", "hh_g02", "hh_g03a", "hh_g03b", "hh_g03b_label", "hh_g03b_oth", "hh_g03c", "hh_g03c_1".

```r
# Subset the data
ihs5_consumption_subset <-
    select(
        ihs5_consumption,
        case_id,
        HHID,
        hh_g01,
        hh_g01_oth,
        hh_g02,
        hh_g03a,
        hh_g03b,
        hh_g03b_label,
        hh_g03b_oth,
        hh_g03c,
        hh_g03c_1
    )
```

The syntax for most tidyverse functions is `function (data,columns)`. Notice that we stored the subsetting operation in a new object called `ihs5_consumption_subset`? This is generally frowned upon unless we intend to use the original dataset for separate operations. Storing the subset in a new object will use up more memory to store the 2 objects. We can overwrite the original object by typing:

```r
ihs5_consumption <- select(
    ihs5_consumption,
    case_id,
    HHID,
    hh_g01,
    hh_g01_oth,
    hh_g02,
    hh_g03a,
    hh_g03b,
    hh_g03b_label,
    hh_g03b_oth,
    hh_g03c,
    hh_g03c_1
)
```

Tip

Instead of typing the column names, we can use the `:` operator to select a range of columns. For example, to select all the columns between `case_id` and `hh_g03c_1` we would type:

```
# Subset the data
ihs5_consumption_subset <-
    select(
        ihs5_consumption,
        case_id:hh_g03c_1
    )
```

Next let us give the columns more meaningful names. We can do this using the `rename` function. For example, to rename the `hh_g01` column to `consumedYN` and `hh_g02` to `food_item`, we would type:

```
# Rename the columns
ihs5_consumption <-
    rename(ihs5_consumption,
           consumedYN = hh_g01,
           food_item = hh_g02)
```

Notice how our operations only affect the specific columns we specify? This is because the `select` and `rename` functions are smart and intiutive.

Excercise

1. Rename the remaining columns to:

old_name	new_name
hh_g01_oth	food_item_other
hh_g03a	consumption_quantity
hh_g03b	consumption_unit
hh_g03b_label	consumption_unit_label
hh_g03b_oth	consumption_unit_oth
hh_g03c	consumption_subunit_1
hh_g03c_1	consumption_subunit_2

Tip

Solution:

```
# Reload the data to start from scratch
ihs5_consumption <-
    read_dta(here::here("data",
                        "mwi-ihs5-sample-data",
                        "HH_MOD_G1_vMAPS.dta"))

# Rename the columns
ihs5_consumption <-
    rename(
        ihs5_consumption,
        consumedYN = hh_g01,
        food_item = hh_g02,
        food_item_other = hh_g01_oth,
        consumption_quantity = hh_g03a,
        consumption_unit = hh_g03b,
        consumption_unit_label = hh_g03b_label,
        consumption_unit_oth = hh_g03b_oth,
        consumption_subunit_1 = hh_g03c,
        consumption_subunit_2 = hh_g03c_1
    )
```

5.2.5 Subsetting rows

We can also subset rows using the `filter` function. For example, to keep only the rows where `consumedYN` is equal to `1`, we would type:

```
# Subset the data
ihs5_consumption <- filter(ihs5_consumption, consumedYN == 1)
```

Notice how we are using the logical operator `==` to test each row whether the value of `consumedYN` is equal to `1`? This is called a conditional statement as we discussed in the previous sections.

5.3 Chaining operations using the pipe operator

We can chain operations using the pipe operator `%>%` or `|>`. This is useful when we want to perform multiple operations on a dataset. For example, to read, subset the data and rename the columns in one operation, we would type:

```
# Read, subset and rename the data
ihs5_consumption <-
    read_dta(here::here("data",
                        "mwi-ihs5-sample-data",
                        "HH_MOD_G1_vMAPS.dta")) |>
    select(
        case_id,
        HHID,
        hh_g01,
        hh_g01_oth,
        hh_g02,
        hh_g03a,
        hh_g03b,
        hh_g03b_label,
        hh_g03b_oth,
        hh_g03c,
        hh_g03c_1
    ) %>%
    rename(
        consumedYN = hh_g01,
        food_item = hh_g02,
        food_item_other = hh_g01_oth,
        consumption_quantity = hh_g03a,
        consumption_unit = hh_g03b,
        consumption_unit_label = hh_g03b_label,
        consumption_unit_oth = hh_g03b_oth,
        consumption_subunit_1 = hh_g03c,
        consumption_subunit_2 = hh_g03c_1
    )
```

We deliberately used both the pipe operators `%>%` and `|>` to show that they are the same. The %>% is the most popular of the `tidyverse` pipes from the `magrittr` package.

Recent versions (circa 2020) intoduced the native `R` pipe `|>`. The pipe operator is useful when we want to perform multiple operations on a dataset without storing the intermediate results in memory.

In the above example we only stored the final result in memory. This is useful when we are working with large datasets and want to save memory.

> Pipes
>
> In Rstudio you can type the pipe operator by typing `Ctlr + shift + m`. You can also change whether the pipe operator is `%>%` or `|>` in the `Tools > Global Options > Code > Editing` menu by changing the `Use native pipe operator |> (requires R 4.1+)` option.

> Warning
>
> When chaining operations we do not need to specify the data argument in the subsequent functions. This is because the output of the previous function is passed to the next function. If we want to specify the data argument, we can use the `.` symbol. For example, to specify the data argument in the `rename` function, we would type:

5.3.1 Change the data type of a column

The `mutate` function is used to create new columns from existing columns. It is also used to change the data type of a column. For example, to change the data type of the `consumption_quantity` column to numeric, we would type:

```
ihs5_consumption <- ihs5_consumption |>
    mutate(food_item_code = as.character(food_item))
```

5.3.2 Create a new column

As we mentioned earlier, the `mutate` function is used to create new columns from existing columns. For example, to create a new column with hh_members (randomly generated) we would type:

```
ihs5_consumption <- ihs5_consumption |>
    mutate(hh_members = sample(1:10, nrow(ihs5_consumption), replace = TRUE))
```

Here we are using the `sample` function to generate random numbers between 1 and 10. The `nrow` function returns the number of rows in the `ihs5_consumption` data. The `replace = TRUE` argument tells the `sample` function to sample with replacement. This means that the same number can be sampled more than once. If we want to sample without replacement we would type `replace = FALSE`.

We used the `sample` function a lot during the generation of the sample data used in this book. You can see more on this in the data generation section.

5.3.3 Vectorised operations

The `mutate` function is also useful for vectorised operations. For example, to create a new column with the consumption per person we would type:

```
ihs5_consumption <- ihs5_consumption |>
    mutate(consumption_per_person = consumption_quantity / hh_members)
```

> Excercise
>
> Suppose this data is from a 7 day recall survey. Create a new column with the consumption per person per day.

5.3.4 Enriching data

We can enrich our data by joining different files using the `join` function. The most common joins are `left_join`, `right_join`, `inner_join` and `full_join`.

The `left_join` function joins two dataframes by keeping all the rows in the first dataframe and matching the rows in the second dataframe.

Most joining operations in hces data analysis are `left_join` operations as we want to keep all the rows in the primary data we are processing and `enrich` it with matched rows in the other data. For example, to join the `ihs5_consumption` data with the `ihs5_household_identifies` contained in `hh_mod_a_filt_vMAPS.dta` data we would type:

```
# Import the data
ihs5_household_identifiers <-
    read_dta(here::here("data",
                        "mwi-ihs5-sample-data",
                        "hh_mod_a_filt_vMAPS.dta"))

# Join the data
ihs5_consumption_j1 <- ihs5_consumption |>
    left_join(ihs5_household_identifiers, by = "HHID")
```

The result is an enriched dataset with rows from the `ihs5_household_identifiers` data that match the `HHID` column in the `ihs5_consumption` data. The `by` argument tells the `left_join` function which column to use to match the rows. If the column names are

the same in both dataframes, we do not need to specify the `by` argument. For example, to join the `ihs5_consumption` data with the `ihs5_household_identifies` contained in `hh_mod_a_filt_vMAPS.dta` data we would type:

```
# Import the data
ihs5_household_identifiers <-
    read_dta(here::here("data",
                        "mwi-ihs5-sample-data",
                        "hh_mod_a_filt_vMAPS.dta"))

# Join the data
ihs5_consumption <- ihs5_consumption |>
    left_join(ihs5_household_identifiers)
```

Excercise

1. Compare the results of the two joins.
2. What is the difference?

5.3.5 Grouping and Summarising Data

We can group data using the `group_by` function. Grouping data is useful when we want to summarise data. In `dplyr` the summaries are created from the `groups` in the data. For eample to summarise the `consumption_per_person` by `food_item` we would type:

```
# Summarise the data
ihs5_consumption_summary <- ihs5_consumption |>
    group_by(food_item) |>
    summarise(consumption_per_person = mean(consumption_per_person, na.rm = TRUE))
```

Here we are using the `mean` function to calculate the mean of the `consumption_per_person` column. The `na.rm = TRUE` argument tells the `mean` function to ignore missing values.

We can also compute multiple summaries at once. For example, to compute the mean and standard deviation of the `consumption_per_person` column we would type:

```
# Summarise the data
ihs5_consumption_summary <- ihs5_consumption |>
    group_by(food_item) |>
    summarise(
        consumption_per_person_mean = mean(consumption_per_person, na.rm = TRUE),
```

```
        consumption_per_person_sd = sd(consumption_per_person, na.rm = TRUE)
    )
```

To compute summaries across multiple groups we can use the `group_by` function with multiple arguments. For example, to compute the mean and standard deviation of the `consumption_per_person` column by `food_item` and `region` we would type:

```
# Summarise the data
ihs5_consumption_summary <- ihs5_consumption |>
    group_by(food_item, region) |>
    summarise(
        consumption_per_person_mean = mean(consumption_per_person, na.rm = TRUE),
        consumption_per_person_sd = sd(consumption_per_person, na.rm = TRUE)
    )
```

In the next section we will learn how to use plots to visualise our data. A basic example of a plot is a bar chart. For example we can visualise the consumption per person by food item using a bar chart. To do this we will use the `ggplot2` package from the `tidyverse` package like so:

```
# Load the ggplot2 package
library(ggplot2)

#  Plot the data
ihs5_consumption |>
    # Add plot aesthetics
    ggplot(aes(x = region, y = consumption_per_person, group = region)) +
    # Add plot type
    geom_boxplot()
```

Here we plotted a `boxplot` of the `consumption_per_person` by `region`.

5.3.6 Data Output/Export

We can export data from R using the `write_csv()` function from the `readr` package. For example, to export the `ihs5_consumption` data to a csv file called `ihs5_consumption.csv` stored in our working directory we run:

```
# Export the data
write_csv(ihs5_consumption, here::here("data",
                                       "ihs5_consumption.csv"))
```

We recommend exporting files to csv as this allows interoperability between various software. If you prefer exporting your data to excel, you can use the `write_xlsx()` function from the `writexl` package. For example, to export the `ihs5_consumption` data to an excel file called `ihs5_consumption.xlsx` stored in our working directory we run:

```
# Export the data
writexl::write_xlsx(ihs5_consumption, here::here("ihs5_consumption.xlsx"))
```

To export the data to a stata file, we can use the `write_dta()` function from the `haven` package. For example, to export the `ihs5_consumption` data to a stata file called `ihs5_consumption.dta` stored in our working directory we run:

```
# Export the data
write_dta(ihs5_consumption, here::here("ihs5_consumption.dta"))
```

6 Data Visualisation

7 hcesNutR Package

The goal of the hcesNutR project is to create a repository of functions and data that will help with the analysis of the Household Consumption Expenditure Survey (HCES) data. A good source of HCES data is the world bank microdata repository.

The package contain functions that will help with the analysis of HCES data. The package also contains the sample data used in this book i.e. r4hces-data/mwi-ihs5-sample-data We will use this sample data to demonstrate the use of the functions in the package. The package is still under development and will be updated regularly.

7.1 Reporting bugs

Please report any bugs or issues here.

7.2 Installation

You can install the development version of hcesNutR from GitHub with:

```
# install.packages("devtools")
devtools::install_github("dzvoti/hcesNutR")
```

As we discussed in previous chapters you need to load the package in your R session before you can use it. You can load the package by running the following code in your R console.

```
library(hcesNutR)
```

7.3 Functions in the package

You can view the functions in the package by running the following code in your R console.

```
ls("package:hcesNutR")
```

> **Tip**
>
> You can read the functions and their description on the project website at: dzvoti.github.io/hcesNutR/reference/index.html

7.4 Sample data

The data used in this example is randomly generated to mimic the structure of the Fifth Integrated Household Survey 2019-2020 an HCES of Malawi. The variables and structure of this data is found here

> **Tip**
>
> All functions in this package take a dataframe/tibble as input data. This is by design to allow flexibility on input data. The example used here is for use on stata files with `.dta` but the functions should work with `.csv` files as well.

7.4.1 Import and explore the sample data

Import the sample data from the `r4hces-data/mwi-ihs5-sample-data` folder. Use the `read_dta` function from the `haven` package to import it.

```
# Import the data using the haven package from the tidyverse
sample_hces <-
  haven::read_dta(here::here("data",
                             "mwi-ihs5-sample-data",
                             "HH_MOD_G1_vMAPS.dta"))
```

7.4.2 Trim the data

In this example we will use `hcesNutR` functions to demonstrate processing of `total` consumption data. The `total` consumption data is the data that contains the total consumption of each food item by each household.

The other consumption columns contain values for consumption from sources i.e. gifted, purchased, ownProduced. The workflow for processing the "other" consumption data is the same as demonstrated below.

```r
# Trim the data to total consumption
sample_hces <- sample_hces |>
  dplyr::select(case_id:HHID,
                hh_g01:hh_g03c_1)
```

7.5 hcesnutR Workflow

7.5.1 Column Naming Conventions and Renaming

The `sample_hces` data is in stata format which contains data with short column name codes that have associated "question" labels that explain the contents of the data. To make the column names more interpretable, the package provides the `rename_hces` function, which can be used to rename the column codes to standard hces names used downstream.

The `rename_hces` function uses column names from the `standard_name_mappings_pairs` dataset within the package. Alternatively, a user can create their own name pairs or manually rename their columns to the `standard` names.

It is important to note that all downstream functions in the `hcesNutR` package work with standard names and will not work with the short column names. Therefore, it is recommended to use the `rename_hces()` function to ensure that the column names are consistent with the package's naming conventions.

For more information on how to use the `rename_hces` function, please refer to the function's documentation: `rename_hces`.

```r
# Rename the variables
sample_hces <- hcesNutR::rename_hces(sample_hces,
                                     country_name = "MWI",
                                     survey_name = "IHS5")
```

7.5.2 Remove unconsumed food items

HCES surveys administer a standard questionaire to each household where they are asked to conform whether they consumed the food items on their standard list. If a household did not consume a food item, the value of the 'consYN' is set to a constant. The `remove_unconsumed` function removes all food items that were not consumed by the household. The function takes in a data frame and the name of the column that contains the consumption information. The function also takes in the value that indicates that the food item was consumed.

```r
# Remove unconsumed food items
sample_hces <- hcesNutR::remove_unconsumed(sample_hces,
                                           consCol = "consYN",
                                           consVal = 1)
```

7.5.3 Create two columns from each dbl+lbl column

The `create_dta_labels` function creates two columns from each dbl+lbl (double plus label) column. The first column contains the numeric values and the second column contains the labels. The function takes in a data frame and finds all columns that contains the double plus label column. The function returns a data frame with the new columns.

```r
# Split dbl+lbl columns
sample_hces <- hcesNutR::create_dta_labels(sample_hces)
```

7.5.4 Concatenate columns

Some HCES data surveys split consumed food items or their consumption units into multiple columns. The `concatenate_columns` function cleans the data by combining the split columns into one column. The function can exclude values from contatenation by specifying the whole or part of values to be excluded.

7.5.4.1 Concatenate food item names

```r
# Merge food item names
sample_hces <-
  hcesNutR::concatenate_columns(sample_hces,
                                c("item_code_name",
                                  "item_oth"),
                                "SPECIFY",
                                "item_code_name")
```

7.5.4.2 Concatenate food item units

```
# Merge consumption unit names. For units it is essential to remove parentesis as they are
sample_hces <-
  hcesNutR::concatenate_columns(
    sample_hces,
    c(
      "cons_unit_name",
      "cons_unit_oth",
      "cons_unit_size_name",
      "hh_g03c_1_name"
    ),
    "SPECIFY",
    "cons_unit_name",
    TRUE
  )
```

> **Tip**
>
> Use the `select` and `rename` functions from the dplyr package to subset the columns containing food item name , food item code, food unit name and food unit code. This is to ensure that the names are meaningful and consistent with the package's naming conventions.

```
sample_hces <- sample_hces |>
  dplyr::select(
    case_id,
    hhid,
    item_code_name,
    item_code_code,
    cons_unit_name,
    cons_unitA,
    cons_quant
  ) |>
  dplyr::rename(food_name = item_code_name,
                food_code = item_code_code,
                cons_unit_code = cons_unitA)
```

7.5.5 Match survey food items to standard food items

The `match_food_names` function is useful for standardising survey food names. This is feasible due to an internal dataset of standard food item names matched with their corresponding survey food names for supported surveys. Alternatively users can use their own food matching names by passing a csv to the function. See hcesNutR::food_list for csv structure.

```
sample_hces <-
  match_food_names_v2(
    sample_hces,
    country = "MWI",
    survey = "IHS5",
    food_name_col = "food_name",
    food_code_col = "food_code",
    overwrite = FALSE
  )
```

7.5.6 Match survey consumption units to standard consumption units

The `match_food_units_v2` function is useful for standardising survey consumption units. This is feasible due to an internal dataset of standard consumption units matched with their corresponding survey consumption units for supported surveys. Alternatively users can download our template from `hcesNutR::unit_names_n_codes_df` and modify it to use their own consumption unit matching names.

```
sample_hces <-
  match_food_units_v2(
    sample_hces,
    country = "MWI",
    survey = "IHS5",
    unit_name_col = "cons_unit_name",
    unit_code_col = "cons_unit_code",
    matches_csv = NULL,
    overwrite = FALSE
  )
```

7.5.7 Add regions and districts to the data

Identify the HCES module that contains `household identifiers`. In some cases this will already be present in the HCES data and should be skipped. From the `household identifiers`

select the ones that are required and add to the data. In this example we will add the region and district identifiers to the data from the `hh_mod_a_filt.dta` file.

```
# Import household identifiers from the hh_mod_a_filt.dta file
household_identifiers <-
  haven::read_dta(here::here("data",
                             "mwi-ihs5-sample-data",
                             "hh_mod_a_filt_vMAPS.dta")) |>
  # subset the identifiers and keep only the ones needed.
  dplyr::select(case_id,
                HHID,
                region) |>
  dplyr::rename(hhid = HHID)

# Add the identifiers to the data
sample_hces <-
  dplyr::left_join(sample_hces,
                   household_identifiers,
                   by = c("hhid", "case_id"))
```

7.5.8 Create a `measure_id` column

The `create_measure_id` function creates a measure id column that is used to identify the consumption measure of each food item. The function takes in a data frame and the name of the column that contains the consumption information. The function also takes in the value that indicates that the food item was consumed.

The `measure_id` is a unique identifier that allows us to join the consumption data with the food conversion factors data.

```
# Create measure id column
sample_hces <-
  create_measure_id(
    sample_hces,
    country = "MWI",
    survey = "IHS5",
    cols = c("region",
             "matched_cons_unit_code",
             "matched_food_code"),
    include_ISOs = FALSE
  )
```

7.5.9 Import food conversion factors.

The available data comes with a 'food_conversion fcators file which has conversion fcators that link the food names and units to their corresponding

```
# Import food conversion factors file
IHS5_conv_fct <-
  readr::read_csv(
    here::here(
      "data",
      "mwi-ihs5-sample-data",
      "IHS5_UNIT_CONVERSION_FACTORS_vMAPS.csv"
    )
  )
```

We need to check if the conversion factors file contain all the expected conversion factors for the hces data being processed. The `check_conv_fct` function checks if the conversion factors file contains all the expected conversion factors for the hces data being processed. T

> Warning
>
> Remember this data was randomly generated so it is expected that the weights will not be realistic. Also not all food items have conversion factors so the weight of those food items will be `NA`.

```
# Check conversion factors
check_conv_fct(hces_df = sample_hces,
               conv_fct_df = IHS5_conv_fct)
```

7.5.10 Calculate weight of food items in kilograms.

The `apply_wght_conv_fct` function will take the `hces_df` and `conv_fct_df` and calculate the weight of each food item in kilograms.

> Warning
>
> Remember this data was randomly generated so it is expected that the weights will not be realistic. Also not all food items have conversion factors so the weight of those food items will be `NA`.

```r
sample_hces <-
  apply_wght_conv_fct(
    hces_df = sample_hces,
    conv_fct_df = IHS5_conv_fct,
    factor_col = "factor",
    measure_id_col = "measure_id",
    wt_kg_col = "wt_kg",
    cons_qnty_col = "cons_quant",
    allowDuplicates = TRUE
  )
```

7.5.11 Calculate AFE/AME and add to the data

> Assumptions
>
> The ame/afe factors are calculated using the following assumptions: - Merge HH demographic data with AME/AFE factors - Men's weight: 65kg (assumption) - Women's weight: 55kg (from DHS) - PAL: 1.6X the BMR

7.5.11.1 Import data required

In order to calculate the AFE and AME metrics we require the following data: - Household roster with the sex and age of each individual `HH_MOD_B_vMAPS.dta` - Household health `HH_MOD_D_vMAPS.dta` - AFE and AME factors `IHS5_AME_FACTORS_vMAPS.csv` and `IHS5_AME_SPEC_vMAPS.csv`

```r
# Import data of the roster and health modules of the IHS5 survey
ihs5_roster <-
  haven::read_dta(here::here("data",
                             "mwi-ihs5-sample-data",
                             "HH_MOD_B_vMAPS.dta"))
ihs5_health <-
  haven::read_dta(here::here("data",
                             "mwi-ihs5-sample-data",
                             "HH_MOD_D_vMAPS.dta"))

# Import data of the AME/AFE factors and specifications
ame_factors <-
  read.csv(here::here("data",
```

```
                        "mwi-ihs5-sample-data",
                        "IHS5_AME_FACTORS_vMAPS.csv")) |>
  janitor::clean_names()

ame_spec_factors <-
  read.csv(here::here("data",
                        "mwi-ihs5-sample-data",
                        "IHS5_AME_SPEC_vMAPS.csv")) |>
  janitor::clean_names() |>
  # Rename the population column to cat and select the relevant columns
  dplyr::rename(cat = population) |>
  dplyr::select(cat, ame_spec, afe_spec)
```

7.5.11.2 Extra energy requirements for pregnancy

```
# Extra energy requirements for pregnancy and Illness
pregnantPersons <- ihs5_health |>
  dplyr::filter(hh_d05a == 28 |
                  hh_d05b == 28) |>
  # NOTE: 28 is the code for pregnancy in this survey
  dplyr::mutate(ame_preg = 0.11, afe_preg = 0.14) |>
  dplyr::select(HHID, ame_preg, afe_preg)
```

7.5.11.3 Process HH roster data

```
# Process the roster data and rename variables to be more intuitive
aMFe_summaries <- ihs5_roster |>
  # Rename the variables to be more intuitive
  dplyr::rename(sex = hh_b03, age_y = hh_b05a, age_m = hh_b05b) |>
  dplyr::mutate(age_m_total = (age_y * 12 + age_m)) |>
  # Add the AME/AFE factors to the roster data
  dplyr::left_join(ame_factors, by = c("age_y" = "age")) |>
  dplyr::mutate(
    ame_base = dplyr::case_when(sex == 1 ~ ame_m, sex == 2 ~ ame_f),
    afe_base = dplyr::case_when(sex == 1 ~ afe_m, sex == 2 ~ afe_f),
    age_u1_cat = dplyr::case_when(
      # NOTE: Round here will ensure that decimals are not omited in the calculation.
      round(age_m_total) %in% 0:5 ~ "0-5 months",
      round(age_m_total) %in% 6:8 ~ "6-8 months",
```

```
    round(age_m_total) %in% 9:11 ~ "9-11 months"
  )
) |>
# Add the AME/AFE factors for the specific age categories
dplyr::left_join(ame_spec_factors, by = c("age_u1_cat" = "cat")) |>
# Dietary requirements for children under 1 year old
dplyr::mutate(
  ame_lac = dplyr::case_when(age_y < 2 ~ 0.19),
  afe_lac = dplyr::case_when(age_y < 2 ~ 0.24)
) |>
dplyr::rowwise() |>
# TODO: Will it not be better to have the pregnancy values added at the same time here?
dplyr::mutate(ame = sum(c(ame_base, ame_spec, ame_lac), na.rm = TRUE),
              afe = sum(c(afe_base, afe_spec, afe_lac), na.rm = TRUE)) |>
# Calculate number of individuals in the households
dplyr::group_by(HHID) |>
dplyr::summarize(
  hh_persons = dplyr::n(),
  hh_ame = sum(ame),
  hh_afe = sum(afe)
) |>
# Merge with the pregnancy and illness data
dplyr::left_join(pregnantPersons, by = "HHID") |>
dplyr::rowwise() |>
dplyr::mutate(hh_ame = sum(c(hh_ame, ame_preg), na.rm = T),
              hh_afe = sum(c(hh_afe, afe_preg), na.rm = T)) |>
dplyr::ungroup() |>
# Fix single household factors
dplyr::mutate(
  hh_ame = dplyr::if_else(hh_persons == 1, 1, hh_ame),
  hh_afe = dplyr::if_else(hh_persons == 1, 1, hh_afe)
) |>
dplyr::select(HHID, hh_persons, hh_ame, hh_afe) |>
dplyr::rename(hhid = HHID)
```

7.5.11.4 Enrich Consumption Data with AFE/AME

We will use the `left_join` function from `dplyr` to join the consumption data with the `aMFe_summaries` data.

The `left_join` function will join the `aMFe_summaries` data to the `sample_hces` data by matching the `hhid` column in both data sets.

The `left_join` function will add the `hh_persons`, `hh_ame` and `hh_afe` columns to the `sample_hces` data.

The `hh_persons` column contains the number of people in each household. The `hh_ame` and `hh_afe` columns contain the AME and AFE factors for each household.

```
sample_hces <- sample_hces |>
  dplyr::left_join(aMFe_summaries)
```

Now we have a "clean" data set that we can use for analysis.

7.6 Summary

This chapter demonstrated the use of the `hcesNutR` package to process HCES data. The package contains functions that will help with the analysis of HCES data.

The package also contains the sample data used in this book i.e. r4hces-data/mwi-ihs5-sample-data We used this sample data to demonstrate the use of the functions in the package.

The package is still under development and will be updated regularly.Please report any bugs or issues here.

7.7 Future work

- Add more functions to the package
- Support more surveys (NGA Living Standards Survey 2018-2019)
- Add more internal data to the package

8 Food Composition Table & Databases: Standardisation

8.1 Introduction

8.1.1 Selecting food composition data

When selecting the food composition table or database (FCT) that will be used, it is good to reflect on the following questions:

1. Relevancy for the study/context (e.g., is that FCT/FCBD geographically and culturally close to our survey scope?).
2. FCT availability & missing values (e.g., are relevant foods and nutrients reported?).
3. Data quality and reporting (e.g., what are the method of analysis and metadata available?).

8.1.2 Objective

This document provide, together with the template document, the steps and description for cleaning and standardising FCTs from diverse sources. More details about the cleaned data that can be found in the repository is documented in this folder (documentation).

For easy navigation and use of this script it is recommended to use Rstudio. In RStudio please click the "Show Document Outline" button to the right of the source button, at the top right of this window. This will allow for easier navigation of the script.

8.2 Environment Prep

First we need to check what packages are installed. If you have run this template before in this RStudio project and are sure these packages are already installed, you can comment out (put a hash at the start of) line 20, and skip it.

```r
# Run this to clean the environment
rm(list = ls())

# Loading libraries

library(readxl) # reading and writing excel files
library(stringr) # character string handling
library(dplyr) # cleaning data
library(here) # file management
```

8.3 Obtaining the raw (FCT) file

8.3.1 Data License Check

Before using any dataset, we recommend to check licensing conditions & record the data source, you can use the README template.

8.3.2 Data Download

If the data is publicly available online, usually you only need to run the code below to obtain the raw files. Remember you only need to do it the first time! Then, the data will be stored in the folder of your choice (see below).

For instance, many raw files can be found provided by the FAO here, in various formats.

Once the link to the data is found, check what file type it is, and paste the direct file link to replace the fill-in value below.

8.3.3 File names conventions

We advise to use the ISO code (2 digits) (see ISO 3166 2-alpha code for further information) of the country or the region of the FCT scope, plus the two last digits of the year of publication to name, both the folder which will contain the data and the scripts related to the FCT. For instance, Western Africa FCT, 2019 will be coded as WA19. This will help with the interoperability, reusability and findability of the data. Also, to streamline the work in the future. That name convention will be used also as the identifier of the FCT.

Note that you need to create the folders to store the FCT.

```
f <- "https://www.fao.org/fileadmin/user_upload/faoweb/2020/WAFCT_2019.xlsx"

 download.file(f,
             destfile = here::here("data", # data folder
                                   'WA19', #FCT folder
                                   #FCT file
                                   "WAFCT_2019.xlsx"),
             method="wininet", #use "curl" for OS X / Linux, "wininet" for Windows
             mode="wb")
```

If using an RStudio project, and you put the .R file and the data file in the same folder as the RStudio project or within a subfolder, files and folders are much easier to navegate as your project/here::here location automatically moves to the main project folder.

8.3.3.1 Using here::here()

8.3.3.1.1 A brief introduction to here::here()

If you are using an RStudio project but used a different download method, or already have the file you want to process on your computer, or are using base R we can still use the here::here function, however we will have to find the file first. The best practice is to put the file in the same folder as this script, or in a folder within the project. If this is done, then use `here::here()` to find your current working directory, and then navigate to the file folder. More information about the here package can be found here.

```
# Run this script to see where is your directory
here::here()
```

8.3.3.1.2 Using here::here()

In order to navigate there, you have to include each subfolder between the `here::here` location and the file itself (so the 'data' folder, the 'FCT' folder and the FCT file).

Find your file in your project, and then direct here::here to it.

```
# This identifies the file and file path, and saves it as a variable
FCT_file_location <- here::here('data','WA19', "WAFCT_2019.xlsx")
```

8.4 Importing the data (loading the data)

8.4.0.1 Using the download code above

First, we must find the file on your system that we want to import. If using RStudio: If you used the download method above Section 1.2 then we will see the same location as specified there to specify the file. Simply copy the contents of the here::here brackets and use it to fill the here::here brackets in the line of code below.

8.4.1 Importing Files

FCT files come in many different forms - the most common being ".xlsx" files and ".csv" files. Methods to import both of these file types will be covered - please navigate to the relevant subsection.

During import, a identifier for the FCT is created and added to the table. Please replace 'WA19' from the next code chunck with the FCT id., comprised of the countries ISO 3166 2-alpha code, and the year the FCT was produced (e.g. for the Western Africa FCT from 2019, the reference would be 'MWA9'). This should be the same as the folder name explained in (section 1.3)[link-to-section].

```
# This is an example of the name
FCT_id <- 'WA19' # Change two first letter for your ISO 2 code & the two digits for the la
```

8.4.1.1 Importing .xlsx files

For the excel-type of files, first, you need to check what information is provided and which of the sheet is providing the FC data.

```
# Checking the sheets

readxl::excel_sheets(FCT_file_location)

data.df <- readxl::read_excel(FCT_file_location, #The file location, as
                              sheet = 5  # Change to the excel sheet where
                              ) %>%
  mutate(source_fct = FCT_id)  #Creates the source_fct column and fills with
```

8.4.1.2 Importing .csv files

```
data.df <- read.csv2(FCT_file_location, #The file location, as identified in section 2.1
                     sep = ",") %>%  # Replace w/ other symbol if needed
  mutate(source_fct = FCT_id) #Creates the source_fct column and fills with a id for this
```

Once imported, it is important to check the data.frame created from the csv, by using `head(data.df)` or clicking on its entry in the Environment panel of RStudio (This second option is not advised with very large files, however, as it can be slow).

If the data shown by doing this has all its columns combined, with a symbol in-between, then that symbol (e.g. ';') is the separator for that csv. Replace comma in the `sep = ","` line from the code block above with the new symbol, and run the entire block again.

8.4.1.3 Visually checking the data

```
# Checking the dataframe
head(data.df)
```

8.4.1.4 Checking the loaded data

Question

How many rows & columns have the data?

You can use the function `dim()` to answer to check the number of rows and column.

```
dim(data.df) # rows & columns
```

Answer

Other useful functions to evaluate the structure of the data are:

```
# Structure (variable names, class, etc.)
str(data.df)

# Checking the last rows and columns
tail(data.df)
```

For opening the dataframe in a tab, you can use `View(data.df)`.

Note: if the dataset is very very big, may crash the R session.

After checking that the correct FCT file have loaded the, then proceed. If not, find the correct file and import it instead.

8.5 Cleaning (tidying) and standardising the data

8.5.1 Formatting FCT into a tabular format

8.5.1.1 Trimming dataframe rows

Running this will trim down the table to only include the row numbers between x and y - replace x and y with your desired values. If you wanted to include multiple row ranges, that is also possible - use comments to differentiate between different row ranges and individual rows. e.g. if you wanted to include rows a:b, row c, row e, and rows g:x, then the code would be `slice(a:b, c, e, g:x)`.

```
data.df %>% slice(1:5) %>% knitr::kable()
```

8.5.1.2 Trimming dataframe columns

If you only wish to include certain columns/nutrients, then you might wish to remove the unnecessary columns to make the dataframe easier to read and manage. This can be done through 2 methods; either by selecting the names of the columns you want to keep, or by selecting the names of the columns you want to remove.

8.5.1.3 Keep specified columns only

This method requires creating a list of column names you want to keep - for #example the line below would select the columns 'Energy_kcal', 'Fatg', 'Protein_g', but nothing else. If you wish to trim the columns this way, replace the items in the first line with the column names you want to keep, then run the code block below.

```
# Storing the variables you want to keep
columns_to_keep <- c('Scientific name', 'Energy\r\n(kJ)')

# Selecting the variables
data.df %>% select(columns_to_keep) %>%
```

```
  head(5) %>%
  knitr::kable()
```

8.5.1.4 Remove specified columns, keep all others

Sometimes it is easier to list the columns you want to remove, rather than the ones you want to keep. The code block below identifies the columns to be removed ('VitB12_mcg' and 'Calcium_mg' in the example), and then removes them. If you wish to trim the columns this way, replace the items in the first line with the column names you want to remove, then run the code block below.

This works in a similar way to the codeblock in section 3.3.1, however by putting an exclamation mark (!) before the list of columns, it inverts the selection - instead of instructing R to keep only the listed columns (as with the codeblock above), it instructs R to keep all columns but the listed ones.

```
# Selecting the variables that you don't want to keep
columns_to_remove <- c('Food name in French', 'Sum of proximate components\r\n(g)')

data.df %>% select(!columns_to_remove) %>%
  head(5) %>%
  knitr::kable()
```

8.5.2 Creating food groups variable and tidying

Some food composition tables reported food groups that were placed as the first row of each category, however that it is not a data structure that can be used, as we need one column per variable. Hence, the food group names are extracted from the rows, and are allocated as a new attribute of each food (e.g., fish and fishery products to catfish). The food groups are stored in a new column (`food_group`).

This process requires multiple steps, each covered in their own subsections below: Extracting food group names, Creating the variable, and checking changes in the structure.

8.5.2.1 Extracting food group names

```
#Creates a list of the food groups using their unique row structure in the table to identi

fgroup <- data.df %>%
```

```
  filter(is.na(`Food name in English`), !is.na(Code)) %>%
  pull(Code) %>%
  stringr::str_split_fixed( '/', n = 2) %>%
  as_tibble() %>%
  pull(V1)

group.id <-  unique(str_extract(data.df$Code, "^[:digit:]{2}\\_"))[-1]
```

9 Creating the food_group variable in the FCT

```
# Removes any rows without a food description entry (the food group name rows, and a row t

data.df <- data.df %>% #Identifies the food group number from the fdc_id, and applies the
  mutate(food_group = ifelse(grepl("01_", Code), fgroup[1],
                       ifelse(grepl("02_", Code), fgroup[2],
                       ifelse(grepl("03_", Code), fgroup[3],
                       ifelse(grepl("04_", Code), fgroup[4],
                       ifelse(grepl("05_", Code), fgroup[5],
                       ifelse(grepl("06_", Code), fgroup[6],
                       ifelse(grepl("07_", Code), fgroup[7],
                       ifelse(grepl("08_", Code), fgroup[8],
                       ifelse(grepl("09_", Code), fgroup[9],
                       ifelse(grepl("10_", Code), fgroup[10],
                       ifelse(grepl("11_", Code), fgroup[11],
                       ifelse(grepl("12_", Code), fgroup[12],
                       ifelse(grepl("13_", Code), fgroup[13],
                       ifelse(grepl("14_", Code), fgroup[14],                          'NA'
```

9.0.1 Diving combined variables into two (or more) columns

In some cases, to minimise the number of missing values, two similar food components are combined into one column/ variable. For example, when Beta-carotene and Beta-Carotene Eq. were reported in the same column and identified using brackets ([]) around the component values (`CARTEB or [CARTEBEQ]`). However, because we are aiming to have one variable per column, we need to divide the combined variables into two (or more) columns, as such the Beta-carote variable was separated into two independent columns (`CARTEBmg`, `CARTBEQmg`). Note that when we are separating these two food components into the new variable, there are several things that we need to check: we need to check whether they are present in the original FCT, otherwise we may be overwriting values.

1) The variable is not present, then create a new column.
2) The variable is present: only overwrite if the value is missing in the original column.

3) Do not remove the brackets from the original variable. As we will add information (metadata) about the quality of the value after removing the brackets.

9.0.2 Renaming variable names: Food components definition and re-naming

Each FCT has its own variable names, including for similar food components. Some FCTs included information related to the FAO/ INFOODS food component identifiers (Tagnames)

while other did not. In order to merge all data from various FCTs we need to harmonise the names of all the variables and particularly the food components of interest. To do so, we evaluated and renamed them with the most appropriate Tagname. Other variables (e.g., food id, food name or food description) were renamed to a common variable name, for instance, `fdc_id`, `food_desc`.

> Question
>
> Are the food component variable names using Tagnames & units, i.e., [Variable][unit] (NAmg)?

If yes, use that information to rename food components

```
# Checking current names
names(data.df)
```

If not, do they provide Tagnames information?

```
# Checking for Tagnames
data.df %>%
  head(5) %>%
  knitr::kable()
```

In this case we can use the Tagname information on the two first rows to rename our variables.

```
# Automatic renaming

for( i in 8:62){ #Loops through each column between column 8 and 64
  first_row <- toString(data.df[1, i]) #Takes the first row for that column and assigns it
  second_row <- toString(data.df[2, i]) #Takes the second row for that column and assigns
  split_string <- str_split(first_row, "\\(") #Splits the first row around "(", assigning
  units_int <- gsub("\\*|\\(|\\)", "", split_string[[1]][length(split_string[[1]])]) #Sepa
  colnames(data.df)[i] <- paste0(second_row, units_int) #The column name is replaced with
```

```
}

# Checking new names

names(data.df)
```

If information is not provided the manual renaming of the food components would be necessary and hence, the identification of each food component to its Tagname.

9.0.2.0.1 The identification of food components

Information on the food components and their description should be sought in for FCT. We advise to use the (Tagnames). Some minor changes in the Tagnames are introduced to be compliant with R conventions. E.g., removing spaces in variable names, changing symbols to characters (e.g., µg to mcg), or standardising the name formatting from using underscores and/or parenthesis to using only underscores. Also, changing dashes (-) to underscores (_). Note that within the Tagnames, the dash is used to denote that the method for obtaining that (component) value is unknown. This is important for the quality assessment of the data.

Also, we also assumed that all the variables labelled as "standardised" were combined or recalculated variables.

Table 2. provides a list of all the most relevant food components and their Tagnames.

```
read.csv(here::here("data",           "fct_variables_standards.csv")) %>% knitr::kable()
```

Some other variables can be manually renamed for instance, food code/id, food name, etc.). Change the names in quotes (" ") to those in your dataset (if needed), remove/ add as needed.

```
data.df <- data.df %>%
dplyr::rename(
  fdc_id = "Code", # Food id/code
food_desc = "Food name in English", # Food name/description
food_descFR = "Food name in French",# Food name/description
scientific_name = "Scientific name", # Scientific name
Edible_factor_in_FCT = "Edible portion coefficient 1 (from as purchased to as described)",
Edible_factor_in_FCT2 = "Edible portion coefficient 2 (from as described to as eaten)", #
nutrient_data_source = "BiblioID/Source") # Reference for NVs reported

names(data.df)[1:5]
```

Are the variable names = column names? If not, more formatting is needed (back to previous step)

9.0.3 Standardisation of values

To perform mathematical operations, characters needed to be converted into numeric operator. This includes three steps:

9.0.3.1 Removing brackets or other special characters.

As, described above in section, special character ("[]", "()", "*")were used to denote "low quality values" and/or alternative (determination) methods. We kept record of those values in metadata files for those cases where the numeric value was extracted and the special character removed. In addition, the variable `comments` was created in a way of keeping that information as metadata for other users.

```
#Creating a dataset w/ the values that were of low quality [] trace or normal

var_nut <- data.df %>% select(Edible_factor_in_FCT:VITCmg) %>% colnames() #selecting nutri

#dataset w/ metadata info that will be removed from the dataset for use
metadata <- data.df %>% mutate_at(var_nut,  ~case_when(
  str_detect(. , '\\[.*?\\]') ~ "low_quality", #Looking for things in square brackets to m
  str_detect(. , 'tr') ~ "trace", #Looking for things marked as "tr" and labels them as tr
  TRUE ~ "normal_value")) #Else it marks the entry as a normal value
```

The following chunk is related to formatting the FCT section, as here we are creating new varibles to separate those Tagnames and/or similar food components into their individual columns with their Tagnames.

```
#Extracting variables calculated with different (lower quality) method
#and reported as using [] and removing them from the original variable

data.df <- data.df %>%
#Creating calculated values from the lower quality method and removing the original values
  mutate(FATCEg = str_extract(FATg, '(?<=\\[).*?(?=\\])'),
   #e.g. this creates the FIBCg value from the FIBTGg value
         FIBCg =  str_extract(FIBTGg, '(?<=\\[).*?(?=\\])'),
         CARTBmcg = ifelse(is.na(CARTBmcg), str_extract(CARTBEQmcg, '(?<=\\[).*?(?=\\])'),
         TOCPHAmg = ifelse(is.na(TOCPHAmg),str_extract(VITEmg, '(?<=\\[).*?(?=\\])'), TOCP
         NIAmg = ifelse(is.na(NIAmg), str_extract(NIAEQmg, '(?<=\\[).*?(?=\\])'), NIAmg),
         FOLSUMmcg = str_extract(FOLmcg, '(?<=\\[).*?(?=\\])'),
         PHYTCPPD_PHYTCPPImg = str_extract(PHYTCPPmg, '(?<=\\[).*?(?=\\])'))
```

9.0.3.2 Changing characters into numeric**

For instance, values that were reported to be trace ("tr") or below the detection limit ("<LOD") were converted to zero (0). These changes are also recorded in the `comments` variable as part of the metadata available.

```
#The following f(x) removes [] and changing tr w/ 0

no_brackets_tr <- function(i){
  case_when(
    str_detect(i, 'tr|[tr]') ~ "0",
    str_detect(i, '\\[.*?\\]')  ~ str_extract(i, '(?<=\\[).*?(?=\\])'),
    TRUE ~ i)
}

data.df <- data.df %>%
  mutate_at(var_nut, no_brackets_tr) #This applies the above function
```

9.0.3.3 Extracting information

Some food component information, for instance alcohol content, could be reported within the food description instead of in a independent variable. Hence that information needs to be extracted and a new variable generated.

9.0.3.4 Converting into numeric

```
# Converting to numeric

wafct <- wafct %>% mutate_at(vars(`Edible_factor_in_FCT`:`PHYTCPPD_PHYTCPPImg`), as.numeri
```

9.0.4 Standardising unit of measurement

To standardise and merge the different FCTs, food components need to be reported in the same units. For example, some nutrients needed to be convert from mg/100g to g/100g, or from percentage (100%) to a fraction (1). For all the unit conversion we followed the FAO/INFOODS Guidelines for Converting Units, Denominators and Expression (FAO/INFOODS, 2012b), and the suggested standard reporting units.

Eg. Converting alcohol from weight in volume (w/v) to weight in mass (w/m) (Eq.1a) or percentage of alcohol (v/v) into weight mass (Eq.1b) (See INFOODS Guidelines - page 12).

Eq.1a ALC (g/100mL) (w/v) / density (g/mL) = ALC (g/100 EP)

Eq.1b ALC (%) (v/v) * 0.789 (g/mL) / density (g/mL) = ALC (g/100 EP)

Eg. amino acids (AA) reported per g in 100g of PROT to mg in 100g of EP:

Eq.2.1 AA mg/100g EP = AA mg/g prot * prot g/100g EP /100

Eq.2.2 AA mg/100g EP = AA g/ 100g prot * prot g/100g EP /100 * 1000/100

Eq.2.3 AA mg/100g EP = AA g/100g prot * prot g/100g EP *10

\Eq.3: Edible portion = Edible portion (%)/100

9.0.4.0.1 Data quality and reporting (e.g., method of analysis, good metadata)

General quality checks are: the level of detail in the food description, the methods used for nutrient values compilation, and the documentation and degree of detail.

Other quality checks that can be performed are: calculating sum of proximate and re-calculating the values of: Carbohydrates available by difference, energy, etc. These is covered in the visualisation and QC section.

9.0.5 Saving the output

We are saving the standardised FCT into the data folder, for use in the future.

```
# Data Output

write.csv(data.df, file = here::here("data", paste0(FCT_id, "_FCT_FAO_Tags.csv")),
          row.names = FALSE) #Saves the newly-created data table to the Output folder

#Run this to clean the environment
rm(list = ls())
```

9.1 Further readings

1. Charrondiere, U.R., Stadlmayr, B., Grande, F., Vincent, A., Oseredczuk, M., Sivakumaran, S., Puwastien, P., Judprasong, K., Haytowitz, D., Gnagnarella, P. 2023. FAO/INFOODS Evaluation framework to assess the quality of published food composition tables and databases - User guide. Rome, FAO. https://doi.org/10.4060/cc5371en

10

11 Appendix A: Sample Data

11.1 Introduction

The sample data used in this book was generated from the Malawi Intergrated Household Survey Fifth Edition 2018-2019 downloaded from here.

The data was generated randomly using the following code:

11.2 Define functions used

11.2.1 Create case_id generation

```
generate_case_ids <- function(n) {
    start_id <- 201011000001
    end_id <- start_id + n-1
    case_ids <- as.character(seq(start_id, end_id, by = 1))
    return(case_ids)
}
```

11.2.2 Create HHID generation function

```
generate_HHIDs <- function(n) {
  hhids <- sapply(1:n, function(x) {
    paste(sample(c(0:9, letters[1:6]), 32, replace = TRUE), collapse = "")
  })
  return(hhids)
}
```

11.3 Set seed and number of households to generate

```
# Set seed
set.seed(123)
# Set number of households to generate
households <- 100
```

11.4 Load Original data and extract food and unit lists

```
# Import Malawi IHS5 HCES consumption module data
original_data <-
  haven::read_dta(here::here("data-ignore", "IHS5", "HH_MOD_G1.dta"))

# Extract "standard" food list from the original data
food_list <-
  original_data |>
  dplyr::select(hh_g02) |>
  dplyr::distinct()

# Extract "non-standard" food lists from the original data
other_food_list_codes <-
  original_data |>
  dplyr::distinct(hh_g02, hh_g01_oth) |>
  dplyr::filter(hh_g01_oth != "") |>
  dplyr::distinct(hh_g02) |>
  dplyr::arrange()
other_food_list_options <-
  original_data |>
  dplyr::distinct(hh_g02, hh_g01_oth) |>
  dplyr::filter(hh_g01_oth != "")

# Extract Food unit lists from the original data
food_unit_lists <-
  original_data |>
  dplyr::distinct(hh_g03b, hh_g03b_label, hh_g03b_oth, hh_g03c, hh_g03c_1)

# Extract the length of Number of foods from the food list
n_foods <- length(food_list$hh_g02)
```

11.5 Data creation

11.5.1 Create HHIDs

```
# Creeate case_ids
case_id <- generate_case_ids(households)
# Generate HHIDs
hhids <- generate_HHIDs(households)
```

11.5.2 Create data

```
sample_data <- tibble::tibble(
  case_id = rep(case_id, each = n_foods),
  HHID = rep(hhids, each = n_foods),
      hh_g00_1 = 2,
    hh_g00_2 = 2,
  food_list |> dplyr::slice(rep(1:dplyr::n(), households)),
      hh_g01 = sample(
      original_data$hh_g01,
      # replace = T,
      size = households * 142
    )
  ) |>

# Add "other food items"

  dplyr::rowwise() |>
  dplyr::mutate(
    hh_g01_oth = dplyr::case_when(
      hh_g02 == 414 &
        hh_g01 == 1 ~ sample(
          dplyr::filter(other_food_list_options,hh_g02 == 414) |> dplyr::pull(hh_g01_oth),
          1
        ),
      hh_g02 == 515 &
        hh_g01 == 1  ~ sample(
          dplyr::filter(other_food_list_options,hh_g02 == 515) |> dplyr::pull(hh_g01_oth),
          1
        ),
      hh_g02 == 117 &
```

```
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 117) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 830 &
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 830) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 310 &
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 310) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 412 &
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 412) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 610 &
      hh_g01 == 1  ~ sample(
       dplyr::filter( other_food_list_options,hh_g02 == 610) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 916 &
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 916) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 209 &
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 209) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 709 &
      hh_g01 == 1  ~ sample(
        dplyr::filter(other_food_list_options,hh_g02 == 709) |> dplyr::pull(hh_g01_oth),
        1
      ),
    hh_g02 == 818 &
      hh_g01 == 1  ~ sample(
```

```
          dplyr::filter(other_food_list_options,hh_g02 == 818) |> dplyr::pull(hh_g01_oth)
         1
        ),
      hh_g02 == 804 &
        hh_g01 == 1  ~ sample(dplyr::filter(other_food_list_options,hh_g02 == 804) |> dply
         1
        ),
      TRUE ~ ""
    )
  ) |>
  dplyr::mutate(hh_g03a = dplyr::case_when(hh_g01 == 1 ~ sample(c(1:10, 0.5:10), 1),
                                           TRUE ~ NA)) |>
  dplyr::rowwise() |>
  dplyr::mutate(unit_key = dplyr::case_when(hh_g01 == 1 ~ sample(1:214, 1), TRUE ~
                                              NA)) |>
  dplyr::mutate(
    hh_g03b = food_unit_lists$hh_g03b[unit_key],
    hh_g03b_label = food_unit_lists$hh_g03b_label[unit_key],
    hh_g03b_oth = food_unit_lists$hh_g03b_oth[unit_key],
    hh_g03c = food_unit_lists$hh_g03c[unit_key],
    hh_g03c_1 = food_unit_lists$hh_g03c_1[unit_key]
  ) |>
  dplyr::select(
    -unit_key,
    "case_id",
    "HHID",
    "hh_g00_1",
    "hh_g00_2",
    "hh_g01",
    "hh_g01_oth",
    "hh_g02",
    "hh_g03a",
    "hh_g03b",
    "hh_g03b_label",
    "hh_g03b_oth",
    "hh_g03c",
    "hh_g03c_1"
  )
```

```
# Add the rest of the columns
sample_data <- original_data |> dplyr::filter(is.na(case_id)) |>
```

```
dplyr::bind_rows(sample_data)
```

```
# Attach stata column labels
for (i in names(sample_data)){
  attr(sample_data[[i]], "label") <- attr(original_data[[i]], "label")
}
```

```
# Export sample data as stata file
haven::write_dta(sample_data,here::here("data","sample_data","MWI-IHSV","HH_MOD_G1_vMAPS.d
```

11.5.3 Create `hh_mod_a_filt.dta` file

```
sample_data |>
dplyr::select(case_id,HHID) |>
dplyr::distinct() |>
dplyr::rowwise() |>
dplyr::mutate(region = sample(1:3,1)) |>
haven::write_dta(here::here("data","sample_data","MWI-IHSV","hh_mod_a_filt_vMAPS.dta"))
```

11.5.4 Create `hh_roster.dta`

```
# Import original roster from IHS5
ihs5_roster <- haven::read_dta(here::here("data-ignore", "IHS5", "HH_MOD_B.dta"))

# create a dataframe with the case_ids and HHIDs of our sample data
sample_roster <- sample_data |> dplyr::distinct(case_id,HHID)

# replicate each row a random number of times between 1 and 10 to simulate household membe
n <- sample(1:10, nrow(sample_roster), replace = TRUE)
sample_roster <- sample_roster[rep(seq_len(nrow(sample_roster)), times = n), ]

# Create other variables
sample_roster <- sample_roster |>
dplyr::rowwise() |>
dplyr::mutate(hh_b03 = sample(ihs5_roster$hh_b03,1),
hh_b05a = sample(ihs5_roster$hh_b05a,1),
hh_b05b = dplyr::case_when(hh_b05a < 5~sample(1:11,1),TRUE~NA))
```

```
# Add the other blank columns from the original dataset
sample_roster <- ihs5_roster |>
dplyr::filter(case_id == "") |>
dplyr::bind_rows(sample_roster)

# Attach stata column labels
for (i in names(sample_roster)){
  attr(sample_roster[[i]], "label") <- attr(ihs5_roster[[i]], "label")
}

# writeout the sample_ihs5_roster
haven::write_dta(sample_roster,here::here("data","sample_data","MWI-IHSV","HH_MOD_B_vMAPS.
```

11.5.5 Create sample "HH_MOD_D.dta"

```
# import original data
original_health <- haven::read_dta(here::here("data-ignore", "IHS5", "HH_MOD_D.dta"))

# Use the sample_roster to create a sample_health dataset
sample_health <- sample_roster |>
dplyr::select(case_id,HHID) |>
dplyr::rowwise()|>
dplyr::mutate(hh_d05a = sample(c(original_health$hh_d05a),1),
hh_d05b = sample(original_health$hh_d05b,1))

# Add the other blank columns from the original dataset
sample_health <- original_health |>
dplyr::filter(case_id == "") |>
dplyr::bind_rows(sample_health)

# Attach stata column labels
for (i in names(sample_health)){
  attr(sample_health[[i]], "label") <- attr(sample_health[[i]], "label")
}

# writeout the sample_ihs5_roster
haven::write_dta(sample_health,here::here("data","sample_data","MWI-IHSV","HH_MOD_D_vMAPS.
```

References