

# Introduction to Data Science

## VLE Materials

There are extra materials such as videos and test quizzes for this block on the VLE – You can find them here: <https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-1>

## What Is Data?

“Data is a set of values of subjects with respect to qualitative or quantitative variables.”

*Wikipedia*

---

Summarized in the form of

- Vector or matrix
- Tensor (high-order matrix)
- Image, or text

## What Is Data Science?

Many people are “doing” data science nowadays in one way or another, but many definitions exist none of which is widely acceptable. The following seems to be reasonable:

Data science is the application of computational and statistical techniques on data to address or gain insight into some problem in the real world.

Notice the key words *computational* (data science typically involves some sort of algorithmic methods written in code), *statistical* (statistical inference enables us to draw conclusions and make predictions), and *real world* (we are talking about deriving insight not into some artificial process, but into some “truth” in the real world).

We can also think of data science as the *union* of the various techniques that are required to accomplish the above. In other words, something like:

Data science = Data collection + Data (pre-)processing + Big data + Scientific hypotheses + Business Insights + Visualisation + Machine learning + Statistics + (etc)

This definition is also useful because it emphasizes that *all* these areas are crucial to obtaining the goals of data science. **In this course we will explore the programming aspect of all of the above areas.**

Another way to conceptualise data science is by thinking in terms of what it is *not*, more precisely what is it not (just); see below.

## Data Science Is Not (Just) Machine Learning

Making good machine learning based predictions can be an important part of data science, but the truly hard elements of data science involve also

- Collecting the data
- Defining the problem you are trying to solve (and frequently, re-defining it many times based upon improved understanding of the problem over time)
- Interpreting and understanding the results, and knowing what actions to take based upon this.

## Data Science Is Not (Just) Statistics

The phrases, “analyzing data computationally” or “to understand phenomena in the real world” point to the definition of statistics. But there are at least two distinctions that are worth making between data science and statistics

1. The academic field of statistics has tended more towards the theoretical aspects of data analysis than the practical aspects. See for example the article “50 Years of Data Science” in the links and resources section.
2. Historically, data science has evolved from computer science as much as it has from statistics: topics like data scraping, and data processing more generally, are core to data science, typically are steeped more in the historical context of computer science, and are unlikely to appear in many statistics courses.

## Data Science Is Not (Just) Data Nor (Just) Big Data

While it is absolutely true that a *substantial proportion* of the data science workload (if not the majority) consists of data collection and management, data science consists of all the other tasks mentioned above.

### Data vs Information

#### Data

- Raw, unorganized facts that need to be processed
- Unusable until it is organized

#### Information

- Created when data is processed, organized, and structured
- Needs to be put in an appropriate *context* in order to become useful

# Examples of Data Science Tasks

Below we provide some indicative examples

## Example 1: Email Spam

- 4601 email messages were stored and labelled as spam or not.
- The relative frequency of the 57 most common words are available. See table below for some of them.
- Aim is to design automatic spam detector that could filter out spam before clogging the users mailboxes.
- The data is summarised in the table below, taken from the Hastie et al. (2001) textbook.

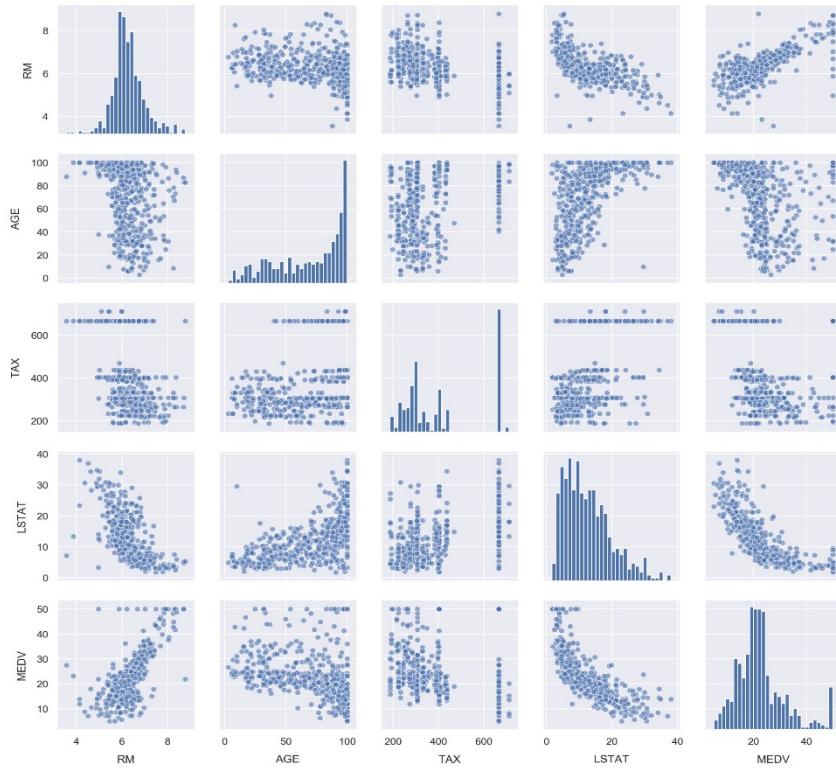
**TABLE 1.1.** *Average percentage of words or characters in an email message equal to the indicated word or character. We have chosen the words and characters showing the largest difference between spam and email.*

	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

Spam email example. Table 1.1 from Hastie et al. (2001)

## Example 2: Real Estate

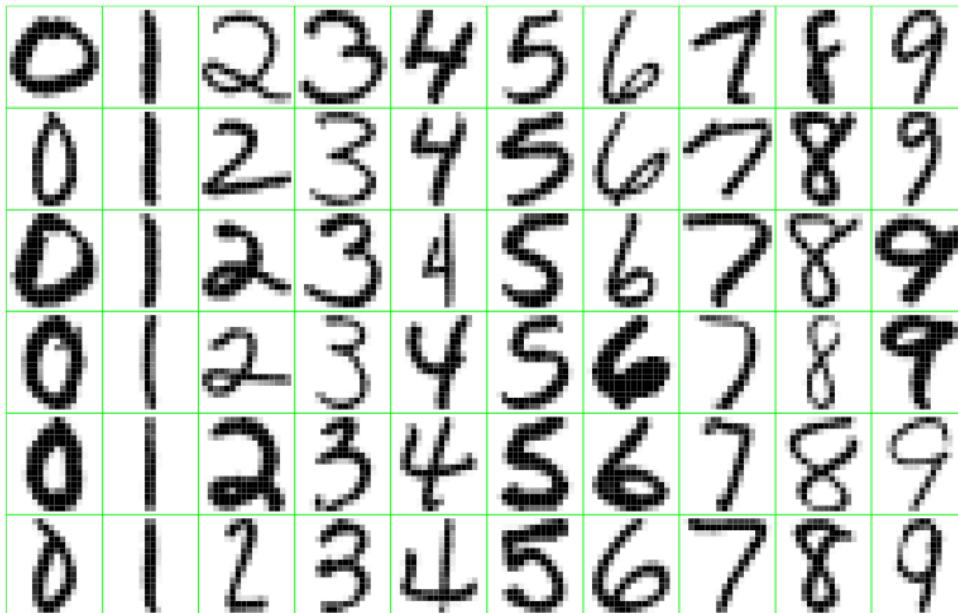
- Determine neighbourhood characteristics that drive house prices.
- The data is from the Boston Housing dataset available from the “scikit learn” Python library which can be found in the links and resources section.



Matrix scatter plot of the Boston Housing dataset variables.

## Example 3: Handwritten Digits

- Construct a machine that identifies the numbers in a handwritten ZIP code from digitised images.
- The data are summarised in the image below, taken from Hastie et al. (2001) textbook which can be found in the links and resources section.



**FIGURE 1.2.** Examples of handwritten digits from U.S. postal envelopes.

Images of digits - Figure 1.2 from Hastie et al. (2001)

## Computer Programming and Data Science

After presenting and introducing the concept of data science, we move on the question “How do we actually do data science?”. Some of the tasks we need to undertake are listed below:

- Data collection
- Data processing (wrangling)
- Data visualisation
- Train and apply algorithms from fields such as machine learning, statistics, data mining, optimisation, image processing, etc.

Clearly the above tasks require the use of *computers* and in particular programming.

**(Computer) programming:** The process of producing an *executable* computer program that performs a specific task, like the ones above. The purpose of programming is to find a *sequence of instructions* that *automate* the implementation of the task for solving a given problem.

**Programming Language:** The source code of a program is written in one or more languages that are *intelligible to humans*, rather than machine code, which is directly executed by the central processing unit. In this course we will explore the computer programming languages **R** and **Python** in the context of data science, i.e. to perform data science tasks.

# Open Source Software

- Both R and Python are open source, i.e. free computer software which the user can modify and distribute within the terms of a licence
- Collaborative development has created diverse and very powerful software ecosystems
- Modular structure permits users to build an environment exactly suited to their needs.

# Programming for Data Science Tools

As with several operations in real life, the use of suitable tools is key to success. Below we summarise the tools to be used in this course as well as others.

## Integrated Development Environments (IDEs)

These are software applications that facilitate computer programming and software development. Examples include RStudio, Spyder, Microsoft Visual Studio etc

## Version Control via Git

- `git`: A version control system
- Allows for complete history of changes, branching, staging areas, and flexible and distributed workflows
- Simplified workflow (taken from Anita Cheng's blog post that can be found in the links and resources section).

## GitHub

- **GitHub** code hosting platform for version control and collaboration
- Based on Git
  - Version control system for tracking changes in computer files and coordinating work on those files among multiple people
  - Created in 2005 by Linus Torvalds
- Largest host of source code in the world

## Markdown (and Other Markup Languages)

- Idea of a “markup” language: HTML, XML, LaTeX
- “Markdown”
  - Created by John Gruber as a simple way for non-programming types to write in an easy-to-read format that could be converted directly into HTML
  - No opening or closing tags
  - Plain text, and can be read when not rendered
- This document was written with markdown

## Useful Links and Resources

- [50 years of data science: Article by David Donoho](#)
- [The joy of Stats: Video by Hans Rosling](#)
- [R web page](#)

- [RStudio web page](#)
- [Python](#)
- [Anaconda web page](#)
- [GitHub web page](#)
- [Git for non-developers - Blog by Anita Cheng](#)

## References

Hastie, T., Tibshirani, R., & Friedman, J. (2001). *The elements of statistical learning*. Springer.

# Source-Code Editors and IDEs for R and Python



## Source-Code Editors

Programming is the process of instructing a computer about how to perform a task. The predominant way of doing so nowadays is by creating a set of instructions in one or more programming languages, like R and Python.

That set of instructions forms the source code for the task at hand, and is typically provided in the form of text, using special, language-specific syntax. Then, depending on the implementation of the language, another computer program, called a [compiler](#), converts the set of instructions into a lower-level language that the computer and operating system understand, producing an executable program that the user can use to perform the task. Another implementation of programming languages allows to execute the instructions directly through a computer program called an [interpreter](#).

So, for writing programs we can use any text editor as a source code editor. For example, you can open your favourite text editor (e.g. Notepad in Windows,TextEdit on macOS, or a plain text editor in Linux) and type in it the following:

```
az <- LETTERS  
az[c(8, 5, 12, 12, 15)]
```

Congratulations! You have authored an R script that first creates a variable `az` and assigns to it the vector with all upper-case letters of the English alphabet, and then subsets that vector to keep only the 8th, 5th, 12th, 12th and 15th letters (in that order). If we pass the instructions in the R script to R (and we will see how we can do so later), then we will get the output "`H`" "`E`" "`L`" "`L`" "`O`".

Of course, a standard text editor may miss some features that are extremely helpful when authoring programs, such as [autocomplete](#), [brace matching](#), [automatic code indentation](#), [syntax highlighting](#), automatic syntax checks while code is being written, etc.

As a result, a wide-range of source-code editors has been developed that provide a range of such features in an attempt to simplify and speed up working with source code in one or even multiple languages at a time. Source-code editors come either as part of integrated development environments (IDEs; more on this later), or as stand-alone programs.

Below, we list the most popular source-code editors that are not formally part of an IDE amongst the participants of the [2019 StackOverflow Analysis Survey](#), who program with at least one of R and Python (in decreasing order of popularity).

<b>Editor</b>	<b>Platform</b>	<b>Open-Source</b>
<a href="#">Visual Studio Code</a>	Windows, macOS, Linux	Yes
<a href="#">Notepad++</a>	Windows	Yes
<a href="#">Vim</a>	Windows, macOS, Linux	Yes
<a href="#">Sublime Text</a>	Windows, macOS, Linux	No, Shareware
<a href="#">Atom</a>	Windows, macOS, Linux	Yes
<a href="#">Jupyter</a>	Windows, macOS, Linux	Yes
<a href="#">Emacs</a>	Windows, macOS, Linux	Yes
<a href="#">TextMate</a>	macOS	Yes

All the editors above provide rich code development features for both R and Python.

## IDEs

As stated in [Wikipedia](#) “*an integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.*”

An IDE typically provides a source-code editor, automation tools for writing programs and compiling source code, a debugger, and utilities and ready processes to test a program. Also, the boundaries between an IDE and some of the editors we listed above are nowadays not very clear, as some editors have plugins that can make them extremely powerful IDEs (e.g. Notepad++, Vim, Atom and Emacs).

Below is a list of the most popular IDEs for R and Python

<b>IDE</b>	<b>Platform</b>	<b>Open-Source</b>	<b>Support for R</b>	<b>Support for Python</b>
<a href="#">Spyder</a>	Windows, macOS, Linux	Yes	No	Yes
<a href="#">RStudio</a>	Windows, macOS, Linux	Yes	Yes	Yes (in recent versions)
<a href="#">Eclipse</a>	Windows, macOS, Linux	Yes	Yes (via plugin)	Yes (via plugin)
<a href="#">Microsoft Visual Studio</a>	Windows, macOS	No	Yes (via plugin)	Yes (via plugin)

# Useful Links and Resources

- [Wikipedia's page on programming languages](#): An authoritative overview of programming languages and their history
- [Wikipedia's page on source-code editors](#)
- [Wikipedia's page on IDEs](#). See also [Wikipedia's comparison of IDEs by language](#)

# Installing and Interacting With R



## R Installation

To use R, you first need to download your copy of it. R is free and easy to download and install on major platforms such as Windows, macOS and Linux.

To download R, simply open your internet browser and go to the web page of the [Comprehensive R Archive Network](#) (CRAN) and follow the instructions. For example, and as the current CRAN web pages are, for installing R in Windows do the following:

1. Go to [Comprehensive R Archive Network](#)
2. Click at [Download R for Windows](#)
3. Click at [base](#) or [install R for the first time](#)
4. Click “Download R X.X.X for Windows,” where X.X.X is the version number of the most current R release; at the time of writing this link reads “Download R 4.0.2 for Windows.”
5. Run the downloaded file for a Windows-style installer.

The [web-page](#) at the link in step 3 above provides more details about the Windows installation process and some frequently asked questions.

If you have a macOS system, then, after step 1 above click [Download R for \(Mac\) OS X](#) and download and run the linked .pkg file.

A new major release of R happens once a year, and there are typically a few minor releases within the year. It is a good idea to update your installation regularly, especially when a major release becomes available. Note that after updating to a major version you will need to reinstall your packages.

# Using R

There are several ways to use R after installation. Some popular ways are

- Through the Command Prompt (Windows) or a terminal (on Linux and macOS)—the minimalist's choice
- Through the default R GUI (on Windows and macOS)—simple and effective
- Through RStudio—powerful and widely used

Another way popular developers and scientists is through the [Emacs Speaks Statistics](#) (ESS) package for the Emacs editor. ESS provides an extremely rich development experience, if you are familiar with [Emacs](#), but we will not cover it here.

## Assignment Operator

Before we continue with the various ways of using R, we need to know how we can—slightly informally put—give names to the results of an operation for further manipulation later.

One of the most common operations in computer programming is [\[assignment\]](#) ([https://en.wikipedia.org/wiki/Assignment\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Assignment_(computer_science))). An assignment statement links a *value* with a *name*. For example in R

```
a <- 987654321/123456789
```

will compute the ratio 987654321/123456789 (did you know that this ratio is almost 8?) on the right of <- using [floating point arithmetic](#), store the result in computer memory and then bind it to the name `a` on the left of <-. For example, we can then do

```
print(a, 15)
[1] 8.000000729
```

to print the value bound to `a` in 15 significant digits.

**The symbol <- (less and a dash) is R's assignment operator.** Another common convention in recent versions of R is to use = for assignment (as in Python and other programming languages). That's OK but <- is formally more valid because i) it highlights the direction of assignment, and ii) formally `a` and 987654321/123456789 are not really equal (by definition of assignment `a = b` is not the same statement as `b = a!`). For this reason, we are and will be using <- for assignment in all R code in these notes. Note that the statement

```
987654321/123456789 -> a
```

also binds the result of 987654321/123456789 to the name `a`, though it is less commonly used.

Note here that we said “the value bound to `a`” instead of “the value of `a`.“ This is specific to how R works. In R, values do not have a name (!), instead the names have values associated to them! So, `a` is simply a reference to a value. For example, if we type `b <- a`, R does not create another copy the already-computed result of 987654321/123456789 but rather creates

a binding of the already existing value that is in memory to another name  $b$ ! See, the “[Named and values](#)” section of the Advanced R book for more details on this.

Nevertheless, for convenience, you may see “value/object  $a$ ” in these notes being used as a shorthand to “value/object that is bound to the name  $a$ .”

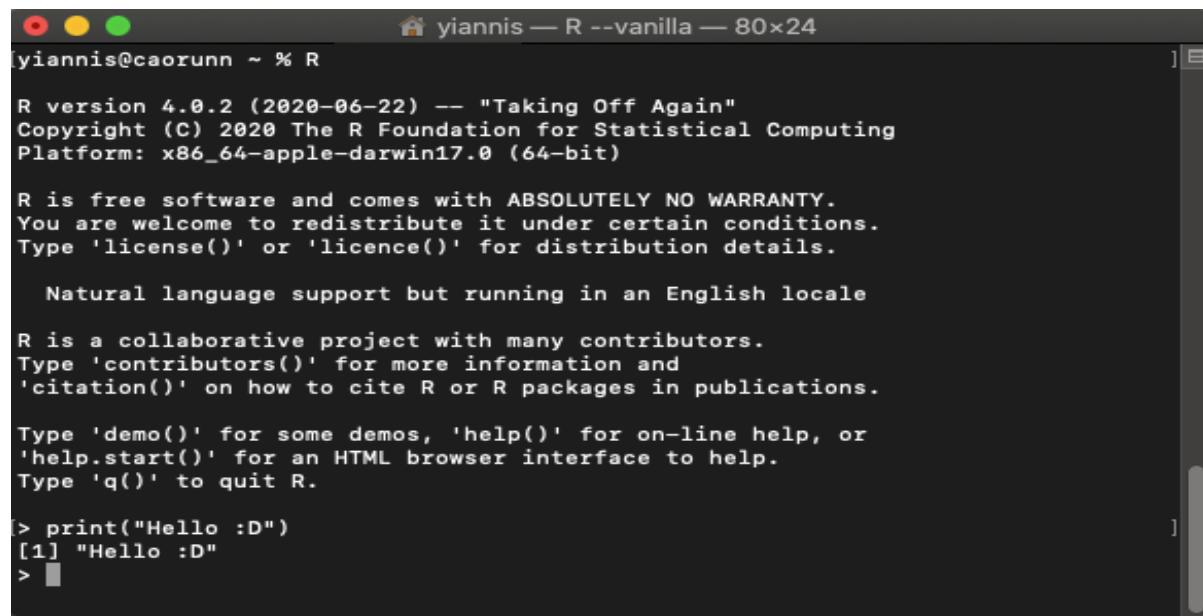
## Through the Terminal

Once you have successfully installed R onto your machine, you may start to use R interactively on the Command Prompt (Windows) or on a terminal (on macOS and Linux). To open R in the Command Prompt or terminal, simply type in `R` into the prompt and hit enter.

Before doing that in Windows you will need to set the path variable to include your R installation. For example, in Windows 10 this can be done as follows:

- Type “Advanced system settings” in the search box
- Open “View advanced system settings”
- Under the “Advanced” tab click “Environment Variables”
- Under System variables click on Path and hit “Edit”
- Hit “New” and add the path to the where the R installer put the R executables (in my Windows 10 computer that is `C:\Program Files\R\R-4.0.2\bin`) in the text box, and click “OK.”

Here is R running through Terminal on a macOS machine



```
yiannis@caorunn ~ % R
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

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

Natural language support but running in an English locale

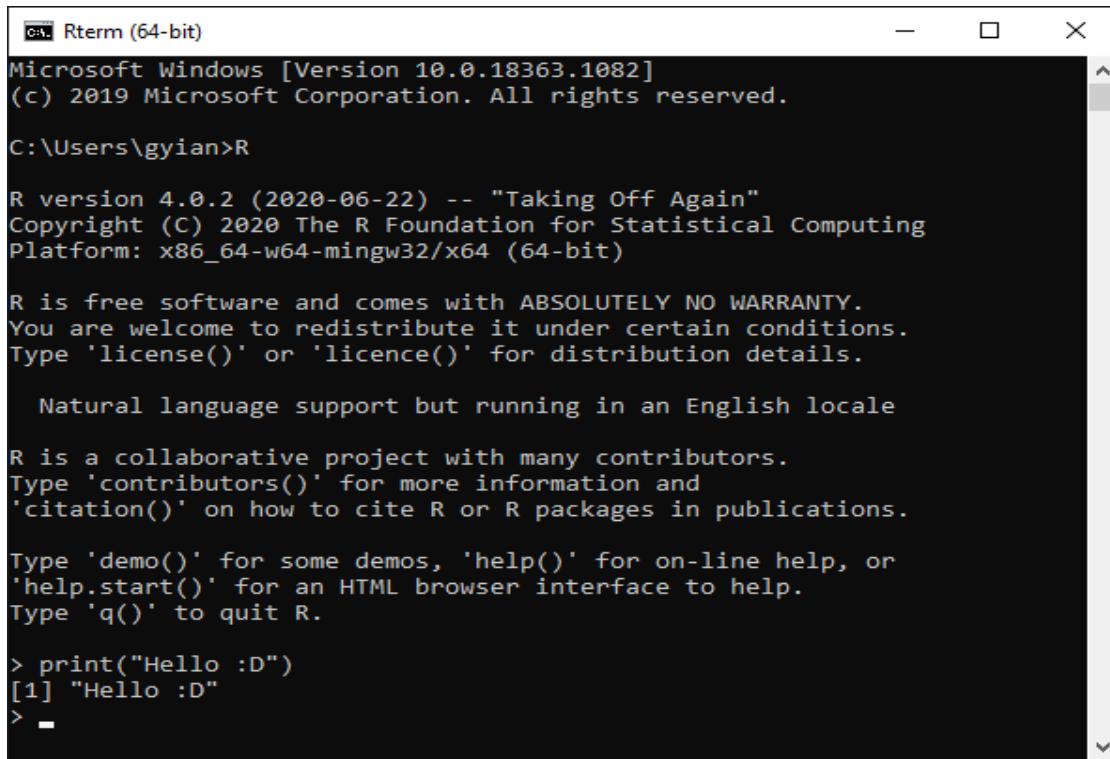
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

> print("Hello :D")
[1] "Hello :D"
```

Screenshot showing R running through Terminal on a macOS machine

and here is R running through Command Prompt on a Windows 10 machine



The screenshot shows the Rterm (64-bit) application window on a Microsoft Windows 10 desktop. The title bar reads "Rterm (64-bit)". The window displays the standard R startup message, including the R version (4.0.2), copyright information, and a welcome message. At the bottom of the window, there is a command prompt where the user has typed the command "print("Hello :D")". The output of this command, "[1] "Hello :D\"", is displayed directly below the prompt.

```
R term (64-bit)
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\gyian>R

R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

Natural language support but running in an English locale

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

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

> print("Hello :D")
[1] "Hello :D"
> -
```

Screenshot showing R running through Command Prompt on a Windows machine

R is a scripting language, which basically means that you can run commands as soon as you type them. When R is running, the greater-than sign (>) is R's "prompt," which indicates that R is ready for you to enter commands. You can then type in R commands here to work with R interactively. When you complete typing your command, press Enter and R will do the computation and print the answer. In the above screenshots, we have issued the command `print("Hello :D")` into the R prompt, and we end up seeing

```
> print("Hello :D")
[1] "Hello :D"
```

The text after [1] is the result from the R code that you wrote after the greater-than sign.

Another example is computing the numeric value of  $\log(2) + 3$ , assigning it to a variable `a`, and then printing the value of `a`:

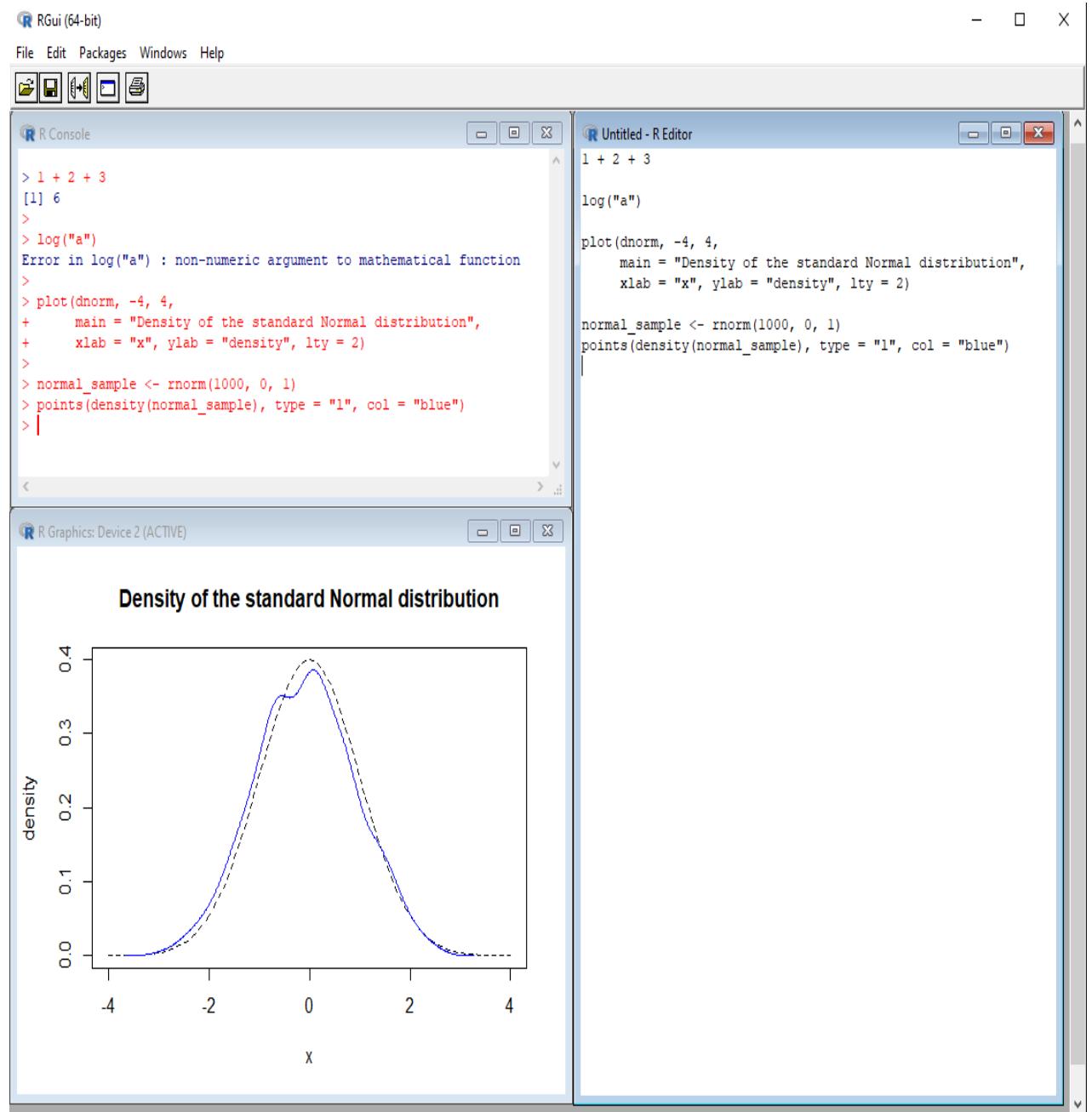
```
> a <- log(2) + 3
> a
[1] 3.693147
```

For more complex computations you may want to use a source-code editor or an IDE for writing the R script. You can then either execute the script interactively by copying and pasting its lines to an R prompt, or execute it non-interactively in a Command Prompt or terminal by doing `Rscript /path/to/script.R`, where `/path/to/script.R` is the R script that you wish to run.

# Through the Default R GUI

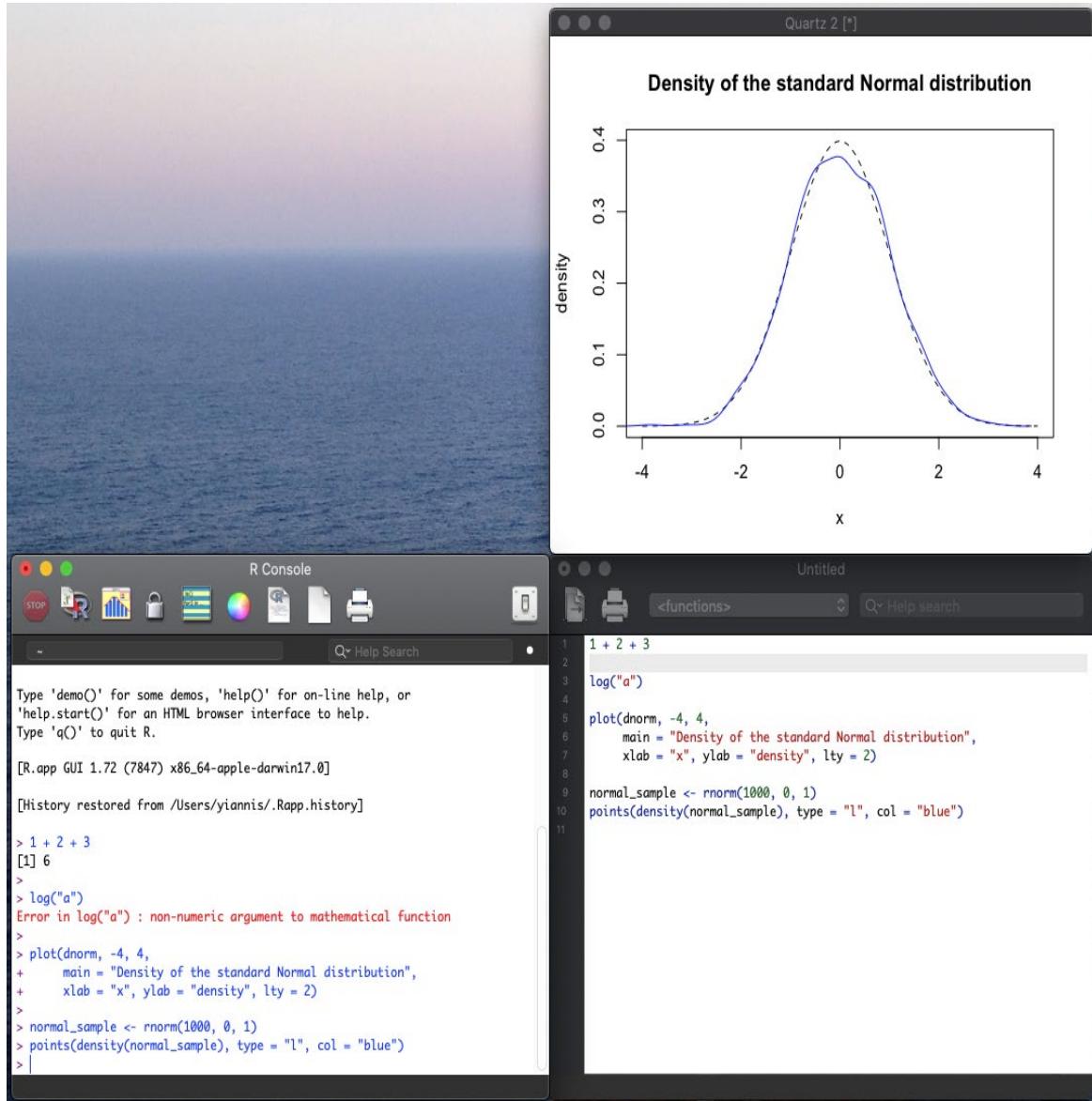
On Windows and macOS machines the R installer installs by default a custom graphical user interface (GUI) for R.

For Windows users, you can open R's GUI by double-clicking on the R icon on the desktop or in the start menu. By doing so, you should get a window that looks like this:



Screenshot showing the default R GUI running on Windows

For Mac users, you can open R's GUI by double-clicking on the R icon in the Applications folder. By doing so, you should get a window that looks like this:



Screenshot showing the default R GUI running on macOS

As you can see in these screenshots, you can interact with R by typing in commands. The R GUIs offer several useful features and utilities, such as an in-built editor to write code, help pages, devices to display graphics, command history and completion, etc.

# Through RStudio

## Downloading and Installing RStudio

RStudio is an cross-platform IDE for R. It is made available as a free, open-source software by [RStudio, PBC](#).

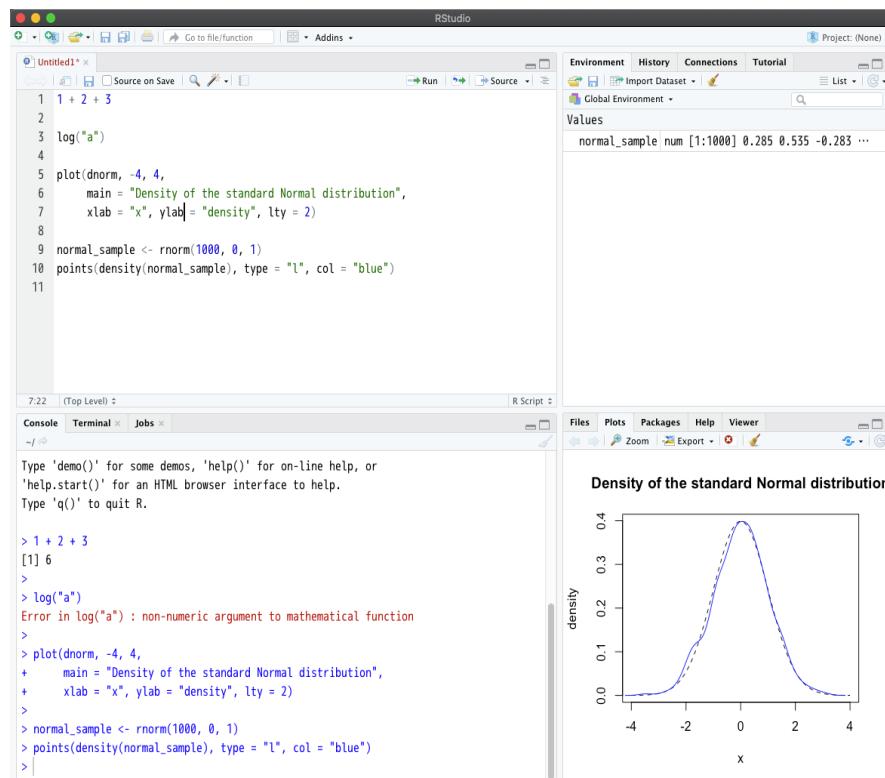
RStudio can be downloaded from [here](#). It comes in two flavours: one is RStudio Desktop, which is a stand-alone computer applications and the other is RStudio Server which runs on a remote server and allows accessing RStudio using a web browser. For the purposes of this course, you will only need to use the **free version of RStudio Desktop**. After hitting the “Download” button you will be redirected at a web page with instructions and a list of installers for RStudio for a wide range of operating systems.

## RStudio Interface

The default RStudio interface is set up as a four-pane workspace that is split up for:

- Creating R scripts (the `Source` pane)
- Typing R commands (the `Console` pane)
- Viewing plots and more
- Viewing the current R workspace and more

Note that you can customize the pane layout slightly if you go to the `Tools` menu → `Global Options`. Here is a screenshot of what RStudio looks like:



Screenshot showing the RStudio interface

The main elements of the default layout are:

1. Top left:

- The `Source` pane is where you write your R scripts. You can also run selected lines from this pane by going to the line you wish to run, and by typing `CTRL + Enter` (Windows, Linux) or `CMD + Enter` (Mac) or hitting the “Run” button on the top-right of the pane. The “Source” button will run all lines from the active script.

2. Top right:

- `Environment` tab: Shows the current workspace in this session and shows the list of R objects that you have created in the session
- `History` tab: Shows the history of all previous commands
- `Connections` tab: Makes it possible to easily connect to a variety of data sources
- `Tutorial` tab: Tutorials for R hosted by the `learnr` package

3. Bottom left:

- `Console` tab: R console for typing R commands
- `Terminal` tab: Opens up a terminal window within RStudio

4. Bottom right:

- `Files` tab: A basic file manager
- `Plots` tab: Shows the history of plots you have created. You can also export a plot to a PDF or image file
- `Packages` tab: Shows external R packages available on your system. If a package is checked, then the package is loaded in the R session
- `Help` tab: Documentation for function or feature if you use `? or help()`

## Interacting With R in RStudio

There are two main ways to interact with R in RStudio:

- Using the R console
- Using a script file

The console pane opens up an R console where you can type in your commands like you would when using the terminal or default R GUIs. Here you can type code in directly. However, for more complicated tasks, it is recommended to put your code in scripts, so that you can have a complete record of what you did in session or what you want to do. In this way, you can share your code with others or continue developing it later on if needed. R script files typically end with the “.R” suffix.

You can write scripts in the `Source` pane and you can open a new script by clicking on `File` → `New File` → `R Script`. After writing a script, you can copy and paste it in the R console. RStudio allows you to send the current line or a currently selected text to the R console by using `Ctrl + Enter` (Windows) or `CMD + Enter` (Mac).

If R is ready to accept commands, the R console will show a > prompt to receive a command. If it is waiting for more data because the command is not complete yet, the console will show a + prompt.

## Changing Working Directory

To change your working directory in RStudio you can click on Session → Set Working Directory → Choose Directory or you can use CTRL + SHIFT + H.

It is also possible to use the R function `setwd()`, which stands for “set working directory.” For example, `setwd("/path/to/directory")` will set the working directory to `/path/to/directory`. Note that in Windows systems paths are customarily given as “” but “” is a special character in strings. So, in Windows machines the R convention is to either do `setwd("/path/to/directory")` or “escape” the special character by doing `setwd("\\\\path\\\\to\\\\directory")`. The former is recommended if you are planning to share your code or if you are working on multiple operating systems.

You can also print your current directory with `getwd()`, which stands for “get working directory”

```
> getwd()
[1] "/Users/yiannis/Documents"
```

Here, R tells us that the current working directory is the Documents directory in the home directory of the user “yiannis” .

## Installing Packages

You can install any R package available in CRAN by typing `install.packages ("packagename")`, where `packagename` is the name of the package you wish to install.

In RStudio, packages can be installed through a GUI after pressing the install button at the top of the Packages tab.

## Getting Help

R has an in-built help facility that can be useful whenever we want to get more information on functions or features. To get more information about any specific function, for example if we wanted to get some documentation for the function `rnorm()` (which is a function to generate random samples from a normal distribution), we type `help(rnorm)` or `?rnorm`.

If we run `?rnorm` in the RStudio console, the documentation of the `rnorm()` function appears in the Help tab. If we run this in one of the R GUIs or through the Windows Command Prompt, the documentation opens by default in HTML format. If R is running on a terminal in macOS or Linux, the documentation opens by default in place in the terminal.

Another useful command is `help.start()`, which will open the HTML version of R's online documentation, which you can browse for help with R and with the installed packages.

## Useful Links and Resources

- [YouTube video](#) describing how to install R and RStudio in macOS, and giving tips in using RStudio
- [YouTube video](#) describing how to install R and RStudio on Windows 7, 8, and 10

# R Markdown and R Notebooks



## Markdown

**Markdown** is a [markup language](#) that consists of a set of rules for adding formatting elements (e.g. boldface, italics, headers, paragraphs, lists, code blocks, images, etc.) to plain text documents.

As the Markdown inventor, [John Gruber](#), states

*The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions.*

John Gruber developed Markdown having in mind a plain-text format that is easy to read and write (so you can use your favourite text editor to author), and that can subsequently be easily converted to HTML. In fact, Jon Gruber's Markdown came with a software tool, written in the [Perl programming language](#), that takes care of the text-to-HTML conversion.

For example, all text before the current sentence can be written in Markdown syntax as:

```
**Markdown** is a [markup language] (https://en.wikipedia.org/wiki/Markup\_language) that consists of a set of rules for adding formatting elements (e.g. boldface, italics, headers, paragraphs, lists, code blocks, images, etc.) to plain text documents.
```

As the Markdown inventor, [\[John Gruber\]](#) (<https://daringfireball.net/projects/markdown/>), states

```
> *The overriding design goal for Markdown's formatting syntax is to  
> make it as readable as possible. The idea is that a  
> Markdown-formatted document should be publishable as-is, as plain  
> text, without looking like it's been marked up with tags or  
> formatting instructions.*
```

John Gruber developed Markdown having in mind a plain text format that is easy to read and write (so you can use your favourite text editor to author), and that can subsequently be easily converted to HTML. In fact, Jon Gruber's Markdown came with a software tool, written in the [\[Perl programming language\]](#) (<https://en.wikipedia.org/wiki/Perl>), that takes care of the text-to-HTML conversion.

Over the coming years after its release, a wide range of Markdown variants sprung up (see for example the [Markdown flavours](#) listed on [CommonMark](#)'s repositories), mainly in an effort to extend the syntax to handle more complex constructs that what the original Markdown implementation provides, such as tables, footnotes, math expressions, and citations.

A landmark development in the markup arena was [John MacFarlane's Pandoc](#), which is an impressive software tool that allows to effortlessly convert between a wide range of markup formats to others and to various file types (see <https://pandoc.org/index.html> for a list). Pandoc supports some key Markdown variants, including [Pandoc's Markdown](#), which significantly enriches John Gruber's Markdown but under the same easy-to-read, easy-to-write principles.

## R Markdown

R Markdown combines the strengths of Pandoc's Markdown and R to produce an authoring framework for data science that allows the combination of text (e.g. describing a narrative), code, and results in a single document, which can in turn, be converted to a wide range of formats (like PDFs, HTML pages, Word documents, etc) using Pandoc.

This makes R Markdown an excellent tool for sharing your analyses and findings with others, either by directly sharing the R Markdown file or by rendering a report from it in an alternative format (e.g. PDF, HTML or Word document).

An R Markdown file has the “.Rmd” extension and typically hosts three types of content:

- [YAML](#) metadata, where some configuration options are provided to guide the R Markdown build process
- Text, for the narrative part of the R Markdown file, written in Markdown syntax
- Code chunks, where R code is placed and options are provided to determine what happens with the code and its outputs

Below are the contents of an example R Markdown file. The YAML metadata are given first and are enclosed between three dashes, like ---. Then, the body follows with text in Markdown syntax, and two code chunks, which are the parts that start with three backticks, typically followed with the language name, like ` ``{r}`, and end with another three backticks ` ``` ).

```
---
title: "Hello Rmd"
author: "[Luke Skywalker] (https://en.wikipedia.org/wiki/Luke\_Skywalker)"
date: 21 December 2112
---
## My First R Markdown File
```

After a hard training day with Yoda, I decided to author my first [R Markdown] (<https://rmarkdown.rstudio.com>) file. This is a text chunk written in \*Markdown syntax\*. I can write \*\*bold\*\* and \*italics\*, and even record quotes I want to remember like

```
> *Do. Or do not. There is no try*
>
```

```
> Yoda, The Empire Strikes Back
```

I can also ask R to run code and return the results. For example, I can ask R to print the quote

```
```{r quote}
print("Do. Or do not. There is no try")
````
```

I can also do complex arithmetic. For example, if your R installation could do infinite arithmetic you could see that `1/81` has all single digits numbers from 0 to 9 repeating in its decimal, except 8!

```
```{r arithmetic}
print(1/81, 15)
````
```

In order to be able to easily convert R Markdown to other formats, open an R prompt in Command Line or terminal, or through your favourite IDE, and install the **rmarkdown** package

```
install.packages("rmarkdown")
```

Then, save the contents of the example R Markdown file above into a file called `hello-Rmd.Rmd`. Then we *load* the **rmarkdown** package and *render* `hello-Rmd.Rmd` by doing

```
library("rmarkdown")
render("/path/to/hello-Rmd.Rmd")
```

where `/path/to/` is the path to the directory you saved `hello-Rmd.Rmd`. This directory will now have an HTML file (which is the default output format) called `hello-Rmd.html`, which if opened in a browser window it will look like

# Hello Rmd

Luke Skywalker

21 December 2112

## My first R Markdown file

After a hard training day with Yoda, I decided to author my first [R Markdown](#) file. This is a text chunk written in *Markdown syntax*. I can write **bold** and *italics*, and even record quotes I want to remember like

*Do. Or do not. There is no try*

Yoda, The Empire Strikes Back

I can also ask R to run code and return the result. For example, I can ask R to print the quote

```
print("Do. Or do not. There is no try")
```

```
## [1] "Do. Or do not. There is no try"
```

I can also do complex arithmetic. For example, if your R installation could do infinite arithmetic you could see that `1/81` has all single digits numbers from 0 to 9 repeating in its decimal, except 8!

```
print(1/81, 15)
```

```
## [1] 0.0123456790123457
```

Screenshot of the HTML output from knitting hello-Rmd.Rmd.

Notice how all code chunks have been evaluated and their results have been returned. Make sure to check the `render()` functions help pages (type `?rmarkdown::render`) for more options on output formats and a wealth of rendering options. RStudio has excellent built-in support for authoring and rendering R Markdown files into a range of formats through its GUI. We will demo that support in the screencast of the current Section.

# R Notebooks

The R Notebook mode is a special way of working with R Markdown files, where the code chunks can be executed independently and interactively, with the chunk output shown immediately under the code chunks. So, R notebooks provide a finer level of interaction with the R Markdown contents than simply rendering the Rmd file does, which makes them great for developing and testing the output of code chunks on the fly.

R Markdown documents can be used as notebooks, and R notebooks can be rendered to other file types for publication or sharing. In fact, RStudio opens R Markdown files in notebook mode by default. For example, below is a screenshot where we have opened the Luke Skywalker's R Markdown file in RStudio and run only the first code chunk (by simply hitting the play button next to the chunk).

```
1 - ---
2 title: "Hello Rmd"
3 author: '[Luke Skywalker](https://en.wikipedia.org/wiki/Luke_Skywalker)'
4 date: "21 December 2112"
5 -
6
7 # My first R Markdown file
8
9 After a hard training day with Yoda, I decided to author my first [R
10 Markdown](https://rmarkdown.rstudio.com) file. This is a text chunk
11 written in *Markdown syntax*. I can write **bold** and *italics*, and
12 even record quotes I want to remember like
13
14 > *Do. Or do not. There is no try*
15 >
16 > Yoda, The Empire Strikes Back
17
18 I can also ask R to run code and return the result. For example, I can
19 ask R to print the quote
20
21 ```{r quote}
22 print("Do. Or do not. There is no try")
23 ```
24
25 I can also do complex arithmetic. For example, if your R installation
26 could do infinite arithmetic you could see that `1/81` has all single
27 digits numbers from 0 to 9 repeating in its decimal, except 8!
28
29 ```{r arithmetic}
30 print(1/81, 15)
31 ```
32
```

Screenshot of hello-Rmd.Rmd opened as an R notebook in RStudio.

Notice how the result is displayed directly after the first code chunk, and that the second code chunk has no output.

The screencast of this segment shows how to create an R Markdown file in RStudio, how to use it as an R notebook, and how to render it in various formats.

# Useful Links and Resources

- [Markdown guide](#): A free online reference guide that explains how to use Markdown
- [Markdown cheat sheet](#): A quick reference to the Markdown syntax
- [R Markdown cheat sheet](#): A PDF giving concise set of notes to be used for quick reference when working with R Markdown. The PDF also provides a quick reference to Markdown syntax
- [RStudio's R Markdown Quick Tour](#): A web-page providing a quick intro to R Markdown
- [Notebooks with R Markdown](#): Video from the talk that J. J. Allaire gave at [useR!2016](#) conference, introducing R notebooks
- [R Markdown: The definitive guide](#): A book that is free to read online (click the link) and is exactly what the title says
- [R Markdown section from the R for data science book](#)

# References

- Wickham, H., & Grolemund, G. (2017). *R for data science: Import, tidy, transform, visualize, and model data* (1st ed.). O'Reilly Media.
- Xie, Y., Allaire, J. J., & Grolemund, G. (2018). *R markdown: The definitive guide*. Chapman & Hall/CRC.

# Installing and Working With Python



## Setup

The simplest and standard way to use Python is by first installing [Anaconda](#). Anaconda is an open-source cross-platform distribution of the Python programming language. It contains several packages such as

- `conda`: Package management system
- `pandas`, `scikit-learn`, `nltk`, etc.: Packages for data science
- *Anaconda Navigator*: A graphical user interface
- *QtConsole*: An interactive Python environment which enhances productivity when developing code
- *Spyder*: A standard cross-platform Integrated Development Environment (IDE) for Python
- *Jupyter Notebook*: An interactive web-browser based application for creating and sharing code

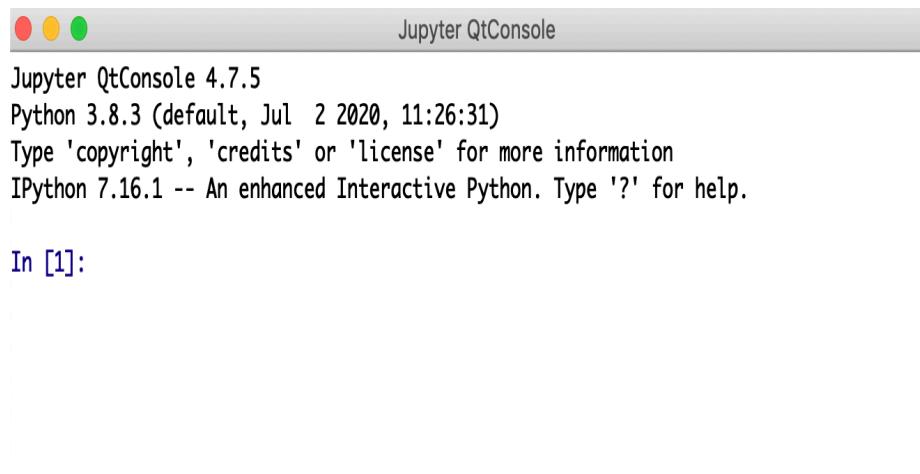
In this set of notes we will work with `QtConsole`, Jupyter Notebook, and Spyder to perform basic operations in Python.

# Running Python via QtConsole

One way to program in Python is by directly executing Python scripts, in other words text files that end with the extension .py, and then using the Python interpreter. Nevertheless, it is generally more efficient to use an interactive session via QtConsole which provides a highly productive environment supporting a number of useful features such as tab completion, integrated help, inline figures etc.

A simple way to launch QtConsole, working both in macOS and Windows and also for Jupyter Notebook and Spyder, is to open the Anaconda Navigator and click [Launch](#) on the QtConsole icon.

You should get a similar result to the picture below



The screenshot shows the Jupyter QtConsole window. At the top, there are three colored buttons (red, yellow, green) and the title "Jupyter QtConsole". Below the title, the text "Jupyter QtConsole 4.7.5" is displayed. Underneath that, it says "Python 3.8.3 (default, Jul 2 2020, 11:26:31)". It also includes the message "Type 'copyright', 'credits' or 'license' for more information" and "IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.". At the bottom left, there is an input cell labeled "In [1]:" which is currently empty.

A screen capture of the initial QtConsole screen

To run your first command in QtConsole type

```
print('Hello Python')
```

and check that the output indeed prints `Hello Python`.

## Your First Python Script

While in some cases it may be possible to complete a task by running several commands one by one, as we just saw, more complex tasks require putting all the commands in a file, also called the script, and run them at once. Python scripts can be run either by directly launching the Python script using the standard interpreter or via the QtConsole environment. The advantage of the latter is that the variables used can be inspected after the script run has completed. Directly calling Python will run the script and then terminate, and so it is necessary to output any important results to a file so that they can be viewed later.

*Task 1:* To test that you can successfully execute a Python script, input the code in the block below into a text file and save it as hello\_python.py.

```
# First Python script
print('Hello Python')
```

Note that the line of the script starting with `#` plays no role at all. The use of the `#` sign is to put comments in the scripts that are useful for other programmers (or the same programmer after some time) to understand what each bit of code is doing.

*Instructions for Task 1:*

- Type the above in a plain text file
- Save this file and make sure the extension is `.py` (if needed simply rename the extension to `.py`)
- Launch the QtConsole and navigate to the directory you saved the file
- Run the program using

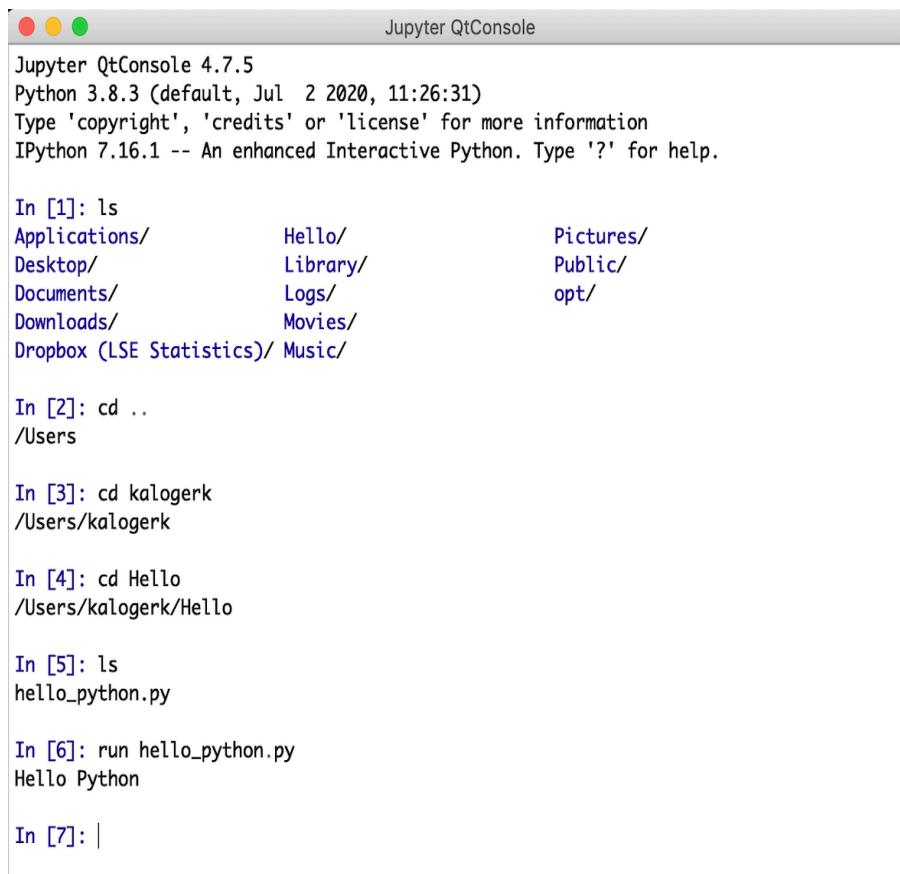
```
run hello_python.py
```

If you get the output `Hello Python` as before you just executed your first Python script!

To navigate within the QtConsole the following commands are needed:

- To list the contents of a directory type `ls`
- To change to a directory named `xname` type `cd xname`
- To go up a folder type `cd ..`

See below how these commands were used to navigate to the directory `/users/kalogerk/Hello` and run the program `hello_python.py`:



The screenshot shows a window titled "Jupyter QtConsole". Inside, a terminal session is displayed:

```
Jupyter QtConsole 4.7.5
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: ls
Applications/          Hello/           Pictures/
Desktop/               Library/        Public/
Documents/              Logs/            opt/
Downloads/             Movies/          Music/
Dropbox (LSE Statistics)/

In [2]: cd ..
/Users

In [3]: cd kalogerk
/Users/kalogerk

In [4]: cd Hello
/Users/kalogerk/Hello

In [5]: ls
hello_python.py

In [6]: run hello_python.py
Hello Python

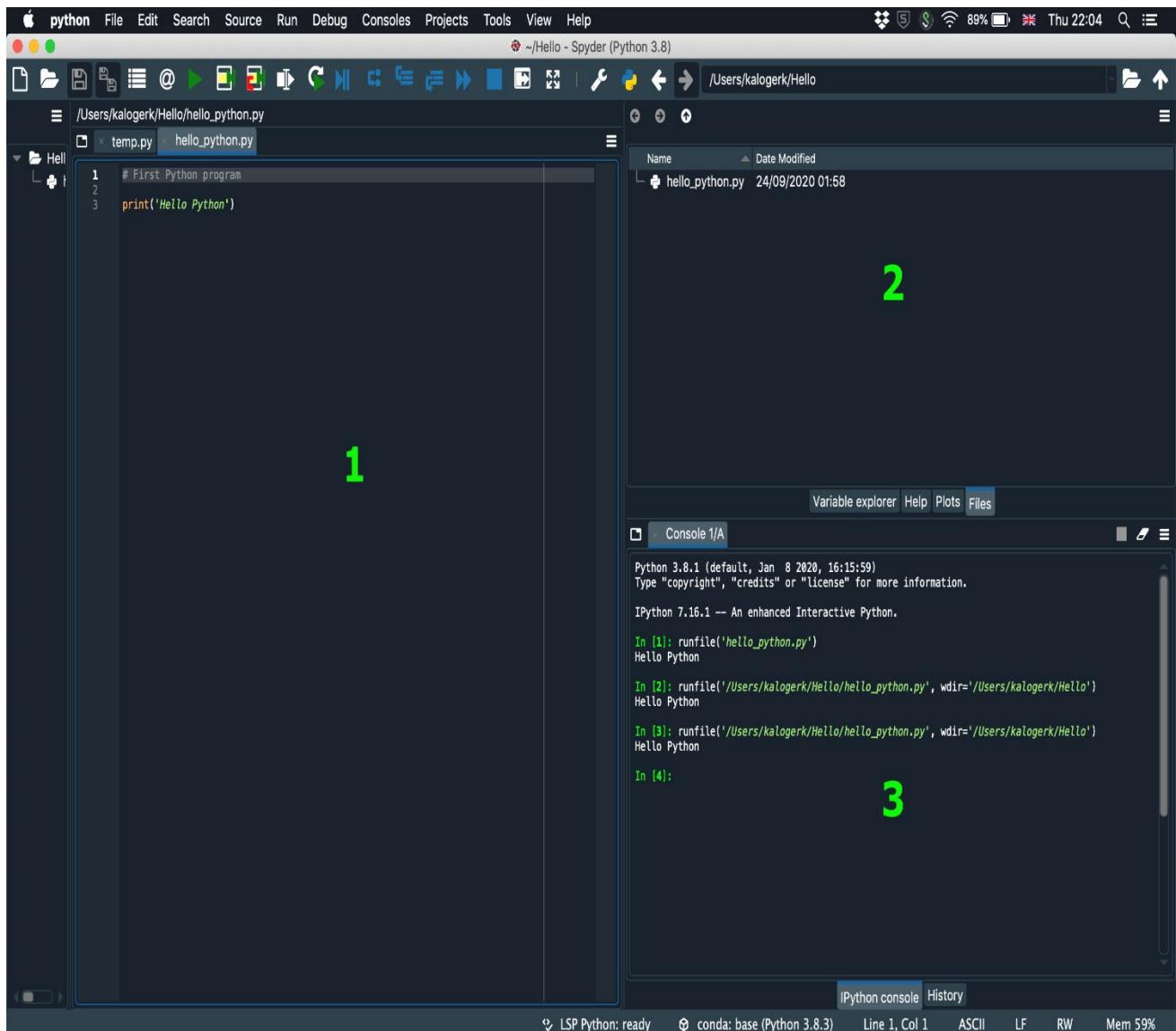
In [7]: |
```

Navigating and running scripts in QtConsole

# Using Python via Spyder

Spyder is an IDE for scientific computing, written in and for the Python programming language. It comes with an Editor to write code, a Console to evaluate it and view the results at any time, a Variable Explorer to examine the variables defined during evaluation, and several other facilities to help you effectively develop the programs you need as a scientist.

To launch Spyder simply click [Launch](#) in the Spyder icon of the Anaconda Navigator, as with QtConsole. Spyder IDE has three main windows as we can see in the screenshot below:



A view of Spyder interface.

1. The Editor window that can be used to write your scripts as the `hello_python.py` program.
2. The Object Inspector where you can access variables, plots and files. In the picture above we see the files in the working directory (where we have saved the file `hello|_python.py`).
3. The Console, which is actually a Python console. In the first line in the image above we typed

```
runfile('hello_python.py')
```

and obtained the `Hello Python` output.

## Running a Python Script in Spyder

It is general good practice to start by creating a project. This offers several advantages:

- Opening, closing or switching to a project automatically saves and restores your Editor panes and open files to exactly how you left off. This allows you to easily switch between many different development tasks without having to manually re-create your session for each one.
- The project path is also used to automatically set your working directory, and can be used as an automatic preset for several modules.
- You can browse all your project files from the Project Explorer, regardless of your current working directory or Files location.

To create a project

- Select Projects → New Project
- Specify the project name and working directory

We are now going to run the previous program but this time using Spyder. To execute the program, either

- Select Run → Run from the menu (or press F5), and confirm the Run settings if required.
- Click the green play button.

In the image of the previous section we did the above and got the `Hello Python` output.

## Jupyter Notebooks

Notebooks integrate code and its output into a single document that combines visualizations, narrative text, mathematical equations, and other rich media. In other words: it is a single document where you can run code, display the output, and also add explanations, formulas, charts, and make your work more transparent, understandable, repeatable, and shareable. Using Notebooks is now a major part of the data science workflow at companies across the globe. If your goal is to work with data, using a Notebook will speed up your workflow and make it easier to communicate and share your results.

- Jupyter Notebooks are open source web-browser based applications to create and share documents that contain
  - Live code
  - Equations
  - Visualizations
  - Explanatory text.
- The name, Jupyter, comes from the core supported programming languages that it supports: Julia, Python, and R. Nevertheless, Jupyter notebooks are most frequently used to code in Python despite the fact that there is support for 40 languages.
- Notebook files have `.ipynb` extension and can be easily shared, e.g. on GitHub

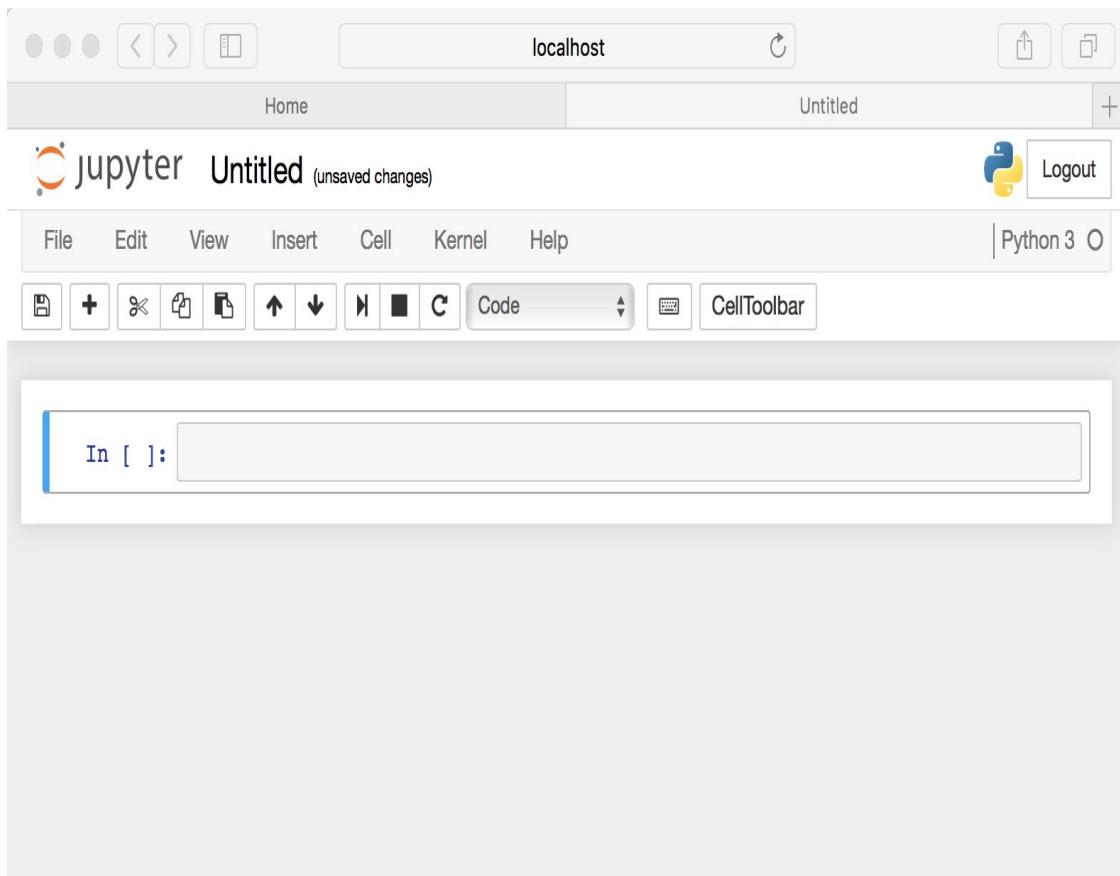
In order to launch open the Anaconda Navigator and click on Jupyter Notebook launch icon, i.e. in the same way as with QtConsole and Spyder.

# Using Jupyter Notebooks

To create a new notebook select from the menu

File → New Notebook → Python 3

Your webpage should look like this



A screen capture of a Jupyter notebook

Notice the cell type set to *Code* below the help menu. This means that you can type Python code in the cell which you can run when finished by pressing the Run button.

You can also use the notebook for writing text. For this change the cell type to *Markdown*, type your Markdown code and hit the Run button when finished.

## The `print` Function in Python

- One of basic function in Python.
- It just prints whatever you put inside its brackets in quotes, e.g.

```
print('Hello')
```

*Task 2:* Your first Jupyter notebook. Get Python to print `Hello Python` using a Jupyter notebook.

*Solution for Task 2:* See the `Hello_Python.ipynb` Jupyter notebook.

## More on the `print` Function

You can put several words together using commas, e.g.

```
print('The', 'losing', 'number', 'is', 7, '.')
```

Note however that when you do that the different words are separated with spaces giving you `7` rather than `7..`. An alternative option is to use the `+` instead of the comma:

```
print('The winning number is '+ str(10) + '.')
```

Note that the above requires text or else string characters input. Hence, we used the command `str(10)` to convert the number 10 into text.

*Task 3:* Print the sentence “4 people out of 3 struggle with math” in the code cell below. Use both `,` and `+` to connect each of the word.

*Solution for Task 3:* See the `Hello_Python.ipynb` Jupyter notebook.

## Exporting Jupyter Notebooks

You can export Jupyter Notebooks to user friendly outputs such HTML or PDF. For HTML the simplest way to do it is by

- File → Download as → HTML (.html)

For PDF you can use

- File → Print Preview

which will open a web page in a different tab. You can then print the page into PDF using your operating system's facilities

*Task 4:* Open the Jupyter notebook you created for tasks 1 and 2 or the `Hello_Python.ipynb` Jupyter notebook. Export to HTML and PDF files.

## Shutting Down Jupyter

- Do not forget to save, `Command+S` (in macOS) or `Ctrl+S` (in Windows and Linux) !
- Jupyter is a server application and closing the browser window will not shut it down. To shut down, click on the `Quit` button
  - File → Close and Halt
  - In the Notebook Dashboard, the initial page when Jupyter notebook launched, → Quit

## Useful Links and Resources

- [Installing Anaconda](#)
- [Spyder website](#)
- [Project Jupyter](#)
- [Jupyter notebooks tutorial from real python.com](#)
- [Jupyter notebooks tutorial from dataquest.io](#)
- [Jupyter notebooks tutorial from towardsdatascience.com](#)

# Version Control



A cartoon from [xkcd.com](https://xkcd.com)

## Version Control Systems

A software project such as R and RStudio is only possible through the collaboration of numerous developers. In projects of that calibre, it is necessary for

- Changes to all files (or to the whole project) and who made to be recorded over time so that they can be undone or reviewed
- Multiple developers to work simultaneously on the project's files
- Developers to experiment with new features without “breaking” the latest stable version of the project.

A [version control system](#) (VCS) is the system that undertakes the tedious task of keeping track of the changes to all project's files and who made them, allowing users to recover any previous version at any given time.

For example, R is developed using a VCS called [Subversion](#) and RStudio is developed using a VCS called [Git](#). You can browse R's Subversion repository at <https://svn.r-project.org/R/>, and RStudio's Git repository at <https://github.com/rstudio/rstudio>.

VCS are useful for all sorts of projects. For example, while developing these lecture notes we had to somehow keep track of the edits and changes that we made to the files, and for that we used Git. You can also use Git to track changes to an R Markdown file or a Jupyter notebook.

## Git: A Distributed Version Control System

Git is a distributed version control system (DVCS) originally created by [Linus Torvalds](#) (the creator and the principal developer of the [Linux kernel](#)) in 2005 for the development of the Linux kernel. Git is distributed in the sense that all project files and their histories are present both remotely and in the computers of all developers contributing to the project. In this way, developers can work offline and asynchronously without a constant connection to a central repository, like other VCSs (e.g. Subversion) require.

## Installation

The [Getting Started](#) section of Git project's documentation provides detailed instructions on how to download and install Git in Windows, macOS, and Linux.

If you have installed Git successfully, then issuing `git --version` in a terminal or the Windows Command Prompt you should get the version of your Git installation. On my system this returns `git version 2.28.0`:

```
> git --version  
git version 2.28.0
```

## Repositories, Commits, and Branches

A *Git repository* is where the entire collection of the files and folders associate with a project are being held, along with their entire history. The history of each file is then organized in snapshots in time called *commits*, and *commits* may be further organized into *branches*.

A Git repository can end up being on your computer in one of two ways:

- A local directory holding a project's files and folders that is not under version control is turned into a Git repository
- A remote Git repository is *cloned* into your computer from elsewhere.

Then, the Git utilities (what we installed earlier) allow you to interact with repositories in all sorts of ways. For example, you can view the history of a file, revert the file to a previous version, add new files into the project, commit changes to the files, create branches, merge changes between branches, etc.

## Basic Git Commands

### Setting up Your Git Credentials

Let's start by creating our first local Git repository. Before we start, you will need to tell Git your name and email address. These will be added to each commit, so that when you start collaborating with others on your project, you can all identify who made each commit. Open the Command Prompt or a terminal and do

```
git config --global user.name "NAME"  
git config --global user.email "EMAIL"
```

after you replace NAME and EMAIL, with your full name and email address.

## Setting up a New Git Repository

First, create a directory called `gitABC` to host the files of your first Git project. In order to create a Git repository in `gitABC`, in the Command Prompt or terminal change to the `gitABC` directory (e.g. using the `cd` command). For example, in my macOS Terminal app, I do

```
cd ~/Repositories/gitABC/
```

Then, we type

```
git init
```

There is no Git repository already in `gitABC`, so this command returns the message  
Initialized empty Git repository in `~/Repositories/gitABC/.git/`. We have just been successful in creating an empty Git project in `gitABC`, ready to host our project files!

## Staging and Committing

Let's now create the first file of `gitABC` project. Using your favourite source-code editor or IDE create an R script with the following R code

```
a <- "Hello Git!"  
print(b)
```

and save it in `gitABC` as `hello-git.R`. However, `hello-git.R` is not yet version controlled!

In order to make Git track changes to the file, the first step to take is to *stage* it. Continuing on the Command Prompt or terminal under the `gitABC` directory, we do

```
git add hello-git.R
```

The current status of our repository can be checked with the `git status` command. For example, in my terminal this returns

```
$ git status  
On branch master  
  
No commits yet  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  new file:   hello-git.R
```

The staging area is the place where all changes have to go before they get *committed* to version control. The staging area is there to give you control on which changes of your project files should be committed and which should not.

Finally, we can commit our staged changes

```
git commit -m "my first commit"
```

This gave me the output

```
[master (root-commit) 96a5cf1] my first commit
1 file changed, 2 insertions(+)
create mode 100644 hello-git.R
```

`hello-git.R` is now under version control and there is nothing to further to commit. `96a5cf1` is the *commit ID*, which is a [SHA-1 hash](#). The text after the `-m` modifier is the *commit message*, which helps to keep a quick record of what happened on each commit. So, try to make those as short and as informative as possible!

## Making Changes

Let's now create a new file under `gitABC` called `README.txt` with the following text

```
This is my first Git repository.
```

Also, if you paid close attention, the R code in `hello-git.R` will not work. In fact, running it in an R prompt returns an error

```
a <- "Hello Git!"
print(b)
Error in print(b): object 'b' not found
```

Clearly our code had a bug! `print(b)` should have really been `print(a)`. Just open the `hello-git.R` file again, fix the bug, and save it.

Now `git status` returns the following

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hello-git.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Git has figured out that `hello-git.R` has been modified, there is a new *untracked* file called `README.txt` in the `gitABC` project, and that there are no changes in the staging area. Again, we do

```
git add hello-git.R README.txt
git commit -m "Fixed bug in hello-git + added readme"
```

The output from the last command is

```
[master a722f3e] Fixed bug in hello-git + added readme
2 files changed, 2 insertions(+), 1 deletion(-)
create mode 100644 README.txt
```

The `git log` command will print a complete log of all the changes in `gitABC` since its creation. In my terminal this gives

```
$ git log
commit a722f3e84e74957e1bda4cc52301748ffe8cceb9 (HEAD -> master)
Author: ...
Date:   Wed Sep 23 12:00:01 2020 +0100

    Fixed bug in hello-git + added readme

commit 96a5cf143fce135a7db11aeb5776ea9bff353fcc
Author: ...
Date:   Wed Sep 23 12:05:02 2020 +0100

    my first commit
```

where I replaced my name and email above with “....” I typically do `git log --pretty=oneline` for a more compact view that involves the commit identifier and the commit message only. See [here](#) for the list of available options for printing logs.

## Branching

It is often the case that we want to diverge from the current state of the project and continue to do work without messing with that state. We can certainly do this by copying the project directory elsewhere and continue to work there, but this is expensive both in time and space for large projects, and can easily cause confusion.

Git offers a lightweight and neat way to support multiple, parallel development of the project without messing with its main state, called *branching*.

So far we have worked on what is known as the *master* branch:

We can create a new branch by doing

```
git branch my-first-branch
```

Note that we are still in the master branch; checking the branch I am at gives

```
$ git branch
* master
  my-first-branch
```

with the \* pointing to `master`. In order to change to the new branch we created we need to *checkout* that branch

```
git checkout my-first-branch
```

This returns the message

```
Switched to branch 'my-first-branch'
```

and the output of `git branch` will now have the \* next to `my-first-branch` (try it!). We can only be at one branch at a time. Any new changes to the project files will be associated with `my-first-branch` and the state of the project in `master` will remain to where we left it.

Let's now open `hello-git.R` again, add a new line `print(paste(a, "Hello branching!"))` and save the file. `hello-git.R` should now have the following lines

```
a <- "Hello Git!"  
print(a)  
print(paste(a, "Hello branching!"))
```

After staging and committing the changes

```
git add hello-git.R  
git commit -m "enriched the message from hello-git.R"
```

you will get the output

```
[my-first-branch bc934c3] enriched the message from hello-git.R  
1 file changed, 1 insertion(+)
```

We can compare the state of the project in `my-first-branch` and `master` using the `git diff` command, followed by the names of the two branches we want to compare. In my terminal, this gives

```
$ git diff master my-first-branch  
diff --git a/hello-git.R b/hello-git.R  
index 0634fae..97bc94e 100644  
--- a/hello-git.R  
+++ b/hello-git.R  
@@ -1,2 +1,3 @@  
 a <- "Hello Git!"  
 print(a)  
+print(paste(a, "Hello branching!"))
```

showing the new line that has been added in `hello-git` (see after +).

## Merging

Once the development in one branch is complete we may want all the changes in that branch to be incorporated into another. For example, currently `my-first-branch` is *one commit ahead* of `master`. In order to update `master` with the changes in `my-first-branch` we need to *merge* `my-first-branch` into `master`. This can be done with the `git merge` command.

We first checkout the `master` branch.

```
git checkout master
```

If you are curious, open `hello-git.R` now to see that it is in the state it was before we switched to `my-first-branch`, i.e. without the new line we added!. Then, you can *merge* `my-first-branch` into it by doing

```
git merge my-first-branch
```

which gives the output

```
Updating a722f3e..bc934c3
Fast-forward
 hello-git.R | 1 +
 1 file changed, 1 insertion(+)
```

You have now successfully brought the changes you made in `my-first-branch` into `master` and you can continue development, either by creating a new branch or directly on `master`.

In large-scale projects there can be *conflicts* when a merge is performed. *Resolving* conflicts is a process that requires familiarity with the project and experience. See [GitHub's guide on resolving conflicts](#) for more information.

## Other Git Functionality

Git offers extremely rich functionality to cater for basic up to highly complex projects. If you are interested in an thorough description of Git's capabilities you may want to check out the [Pro Git](#) book (Chacon & Straub, 2014).

Also make sure to check out the [Git Cheat Sheets](#) for a collection of cheat sheets and manuals about using Git (and GitHub).

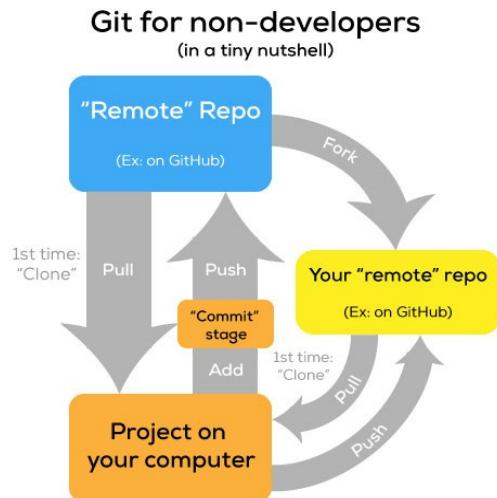
## Useful Links and Resources

- [\*Happy Git and GitHub with R\*](#): A book by [Jenny Bryan](#) aimed at people who use R for data analysis or who work on R packages; the book explains how Git/GitHub are used in data science and what are the differences with the use of Git/GitHub for pure software development.
- [Git handbook](#): An accessible overview of Git and GitHub by the GitHub team.
- [Karl Broman's Git/GitHub guide](#): A guide to Git and GitHub by a statistician.
- [Learning Git branching](#): An interactive app that allows you to learn Git by doing.
- [Git cheat sheets](#): A collection of cheat sheets and manuals about using Git and Github.
- [Git and GitHub section of the second edition of the R packages book](#): An accessible introduction to Git and Github, with focus on projects related to R through RStudio.

## References

Chacon, S., & Straub, B. (2014). *Pro git: Everything you need to know about git* (2nd ed.). Apress.

# Repository Hosting Services and Collaboration Platforms: GitHub



An illustration of version control through git from Anita Chengs blog post about [Git for non-developers \(and total newbies\)](#)

## GitHub

Until now we have been working on a local repository. When working on collaborative projects, it is often necessary to *push* the changes you made on the branches of your local repository to a *remote repository*, so that others can pick them up and continue with development.

There are multiple ways to create remote repositories, however the most popular solution is to host them on a repository hosting service like [GitHub](#), [GitLab](#) or [Bitbucket](#), which also, nowadays, act as collaboration platforms.

In this course we are going to use GitHub. So, go on and create an account on [GitHub](#) (choose the free plan), and use the same email address as you used when you set your Git credentials earlier.

## Pushing to a Remote Repository

Once you login into your account you will be given the option to create a new repository. Create a new repository named `gitABC` and then follow the instructions you will get in order to *push* your local `gitABC` repository there.

If you succeed, you will notice that your project has a web-page where you can browse through your projects branches and files, view commits, add comments, and many other features. You can get a good idea of what is offered by browsing through the GitHub page of a mature project like RStudio at <https://github.com/rstudio/rstudio>.

## Cloning

A core feature that Git offers is the ability to *clone* a remote repository into your computer, in order to continue development locally and, possibly, then *push* your changes. For example, you can easily get the whole Git repository (files, history, and everything!) of **pandas** (a popular Python data analysis toolkit) by doing

```
git clone https://github.com/pandas-dev/pandas.git
```

(make sure you first navigate to the directory you want to keep the **pandas** repository). The URL used above can be found on the [pandas GitHub page](#) by clicking “Code.”

## Other GitHub Features

Another feature GitHub offers is the ability to *fork* a project. This essentially means that a copy of the GitHub project you forked is placed in your account. A GitHub project can be forked by navigating to the project’s page and hitting *Fork* on the top right.

You may then want to *clone* the forked project onto your computer to work locally, and push your changes. Note that your changes are pushed to your fork of the project and not to the project you forked! You can let the project developers know about your edits by raising a *Pull request*, where you discuss what are the changes you made and allow them to review them. See [GitHub’s guide on creating a pull request from a fork](#) for more details.

## Git Integration in RStudio and Spyder

Both RStudio and Spyder offer basic interfaces to core Git capabilities, like staging, committing, branching, pushing, pulling, and more. For a comprehensive description of RStudio’s capabilities and other helpful instructions relating to Git (e.g. setting up SSH keys) see the [Git and GitHub section of the 2nd edition of the “R Packages” book](#). This will give you an accessible introduction to Git and GitHub, with focus on projects related to R through RStudio. For more details about the Git integration in Spyder see at the [Spyder web pages](#).

## GitHub Education

Go to <https://education.github.com/> and get the Student Developer Pack for some cool freebies.

# Useful Links and Resources

- [\*Happy Git and GitHub with R\*](#): A book by [Jenny Bryan](#) aimed at people who use R for data analysis or who work on R packages. The book explains how Git/GitHub are used in data science and what are the differences with the use of Git/GitHub for pure software development.
- [\*Git Cheat Sheets\*](#): A collection of cheat sheets and manuals about using Git and GitHub.
- [\*Git and GitHub section of the 2nd edition of the R packages book\*](#): An accessible introduction to Git and GitHub, with focus on projects related to R through RStudio.

# Block 02 Data

## Structured, Semi-Structured, and Unstructured Data

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:

<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-2>



There are broadly three categories of data: *structured*, *semi-structured* and *unstructured*. This categorization is useful as it can direct as to what solutions may be used for storing and analyzing the various kinds of data, or what steps need to be taken in order to analyze the data. However, this categorization is not as clear-cut as one would hope, and there are data sets that are a combination of structured, unstructured and semi-structured data.

### Structured Data

Structured data is data that is organized according to a predetermined set of rules. A characteristic of structured data is that they are relatively straightforward to read, filter and analyze, and that there is a wealth of statistical and machine learning procedures for directly drawing insights from them. Structured data are typically represented in tabular format, in one or more different tables, with predetermined relationships between the different rows and columns. For example, the data that a smartwatch or a smartphone collects when someone is jogging includes records of latitude, longitude, altitude, distance run, heart rate, and current speed at regular times. So, we can organize the data in a tabular format as follows

|          | timestamp           | latitude | longitude | altitude | distance | heart_rate |
|----------|---------------------|----------|-----------|----------|----------|------------|
| speed    | 2013-06-01 18:40:29 | 50.81381 | -1.712606 | 80.20001 | 1805.94  | 133        |
| 4.060059 | 2013-06-01 18:40:30 | 50.81383 | -1.712649 | 80.00000 | 1810.00  | 133        |
| 4.550049 | 2013-06-01 18:40:31 | 50.81385 | -1.712700 | 79.79999 | 1814.55  | 133        |
| 2.979981 | 2013-06-01 18:40:32 | 50.81387 | -1.712734 | 79.79999 | 1817.53  | 133        |
| 2.969971 |                     |          |           |          |          |            |

|            |          |          |           |          |         |     |
|------------|----------|----------|-----------|----------|---------|-----|
| 2013-06-01 | 18:40:33 | 50.81388 | -1.712777 | 79.59998 | 1820.50 | 133 |
| 3.650024   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:34 | 50.81389 | -1.712826 | 79.59998 | 1824.15 | 133 |
| 3.229980   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:35 | 50.81391 | -1.712862 | 79.40002 | 1827.38 | 133 |
| 4.650024   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:36 | 50.81393 | -1.712911 | 79.40002 | 1832.03 | 133 |
| 4.149902   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:37 | 50.81395 | -1.712963 | 79.20001 | 1836.18 | 133 |
| 2.000000   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:38 | 50.81395 | -1.712994 | 79.20001 | 1838.18 | 133 |
| 4.210083   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:39 | 50.81396 | -1.713053 | 79.00000 | 1842.39 | 133 |
| 5.189941   |          |          |           |          |         |     |

The organization of the data in this way prescribes that the set of values recorded by the smartwatch or smartphone at each timestamp will be organized in rows, and each row will consist of the values that each variable takes, organized in a way so that each column of the resulting table contains the values of one and only one variable. We also know exactly how to organize a new record of the same information in the table. The existence of a predetermined set of rules for structuring the data makes it easier to organize, search, and analyze the information within it. To appreciate this, try to quickly spot all records after 18:40:36 on 2013-06-01, or determine whether at those timestamps the average speed is greater than 3 metres per second.

There is a wealth of structured data sets generated both by machines and humans. Examples of structured data sets that you have encountered include stock prices, addresses, identity information (e.g. social security numbers, names, surname, etc), sensory data produced by smartphones (such as the above example), flight arrival and departure details, etc.

[Relational database](#) management systems (RDBMS), like SQL that you will encounter later, are very powerful tools for the efficient storing and handling of structured data (e.g. searching, filtering, joining, sorting etc).

## Unstructured Data

Unstructured data sets are data sets for which it is difficult to have a predetermined set of rules for organizing them. Examples include images, videos, sounds (music tracks for example), text, websites, books. As a result the vast majority of the data that we interact with nowadays is unstructured!

You can appreciate the challenges involved with unstructured data if you try a simple exercise: organize the text in this paragraph in a tabular format. You will see that there is more than one way of doing so. Would you organize the text by words, by sentences, by letters? Would you keep the order the words appear in? What happens with punctuation?

Clearly, the lack of structure makes unstructured data more difficult to search, organize and analyze. This is the reason why many machine learning and statistical procedures for unstructured data rely on the appropriate transformation of unstructured data into a structured form. For example, [topic models](#) can be used for the discovery of topics or themes in massive and, otherwise unstructured, collections of documents (see, for example, Blei, 2012 for a review of basic topic models). While topic models provide insights for unstructured data,

they typically require the transformation of unstructured text into a [document-term matrix](#) which is then structured data! As an example in R, we use the [tm](#) R package to convert text from four documents into a document-term matrix.

```
library("tm")
doc1 <- "I love programming in R and hate programming in Python"
doc2 <- "I love programming in Python and hate programming in R"
doc3 <- "I love programming in Python and R"
doc4 <- "I hate programming"
## Build a corpus and a document-term matrix
corpus <- Corpus(VectorSource(c(doc1, doc2, doc3, doc4)))
dt_mat <- DocumentTermMatrix(corpus)
as.matrix(dt_mat)

Terms
Docs and hate love programming python
 1   1     1     1       2     1
 2   1     1     1       2     1
 3   1     0     1       1     1
 4   0     1     0       1     0
```

Of course, the document-term matrix is nowhere close to capturing all the information that is in the unstructured text!

The specifics of topic modelling are beyond the scope of this course, but if you want to run the above commands on your own, you first need to type `install.packages("tm")` in R in order to install the [tm](#) R package. Also, if you are curious, you can browse through the help pages of `Corpus()`, `VectorSource()`, and `DocumentTermMatrix()` (e.g. by typing `?Corpus`, `?VectorSource`, and `?DocumentTermMatrix`, respectively) to see what each of these functions does.

## Semi-Structured Data

Semi-structured data does not conform to relational databases such SQL, but still contains some level of organization through semantic elements like tags and metadata. An example of semi-structured data are Markdown documents! For example, the contents of the `hello-Rmd.Rmd` R Markdown file that we encountered in previous weeks are

```
# My first R Markdown file
```

After a hard training day with Yoda, I decided to author my first [R Markdown] (<https://rmarkdown.rstudio.com>) file. This is a text chunk written in \*Markdown syntax\*. I can write \*\*bold\*\* and \*italic\*, and even record quotes I want to remember like

```
> *Do. Or do not. There is no try*
>
> Yoda, The Empire Strikes Back
```

I can also ask R to run code and return the results. For example, I can ask R to print the quote

```
```{r quote}
print("Do. Or do not. There is no try")
```
```

I can also do complex arithmetic. For example, if your R installation could do infinite arithmetic you could see that `1/81` has all single digits numbers from 0 to 9 repeating in its decimal, except 8!

```
```{r arithmetic}
print(1/81, 15)
```
```

Clearly, there is a quite a bit of unstructured data in the above, like free text and code. But, there is a certain level of organization of R Markdown files, by using # for titles, > for quoters and ``` for code blocks, [ ]() for links and so on. So, extracting the code from an R Markdown document, or counting the number of links, paragraphs, sections or quotes is a relatively straightforward process.

See [Wikipedia's page on Semi-structured data](#) for examples.

## Useful Links and Resources

- [What's the difference between structured, semi-structured and unstructured data?](#) by Bernard Marr: A business perspective on structured, unstructured, and semi-structured data.
- [topicmodels](#) R package: a package with a nice [vignette](#) about topic models.

## References

Blei, D. M. (2012). Probabilistic topic models. *Communications of the ACM*, 55(4), 77–84.

# File Formats for Data Exchange

## Plain Text

The simplest way to store information in a computer is in plain text. Plain text files are perhaps the most basic (and perhaps the most reliable) form of data storage. They are not the most sophisticated or efficient to read and write, but they have the advantage of being readable and writable across platforms and with virtually any text editor. The typical extension for plain text file is .txt but they can be also encountered with other extensions or no extension at all.

Just as an example, if you open `nile.txt` file in your favourite text editor — even your source-code editor —, you will see

```
1120 1160 963 1210 1160 813 1230 1370 1140 995 935 1110 994 1020 960
1180 799 958 1140 1100 1210 1150 1250 1260 1220 1030 1100 774 840 874 694
940 833 701 916 692 1020 1050 969 831 726 456 824 702 1120 1100 832 764 821
768 845 864 862 698 845 744 796 1040 759 781 865 845 944 984 897 822 1010
771 676 649 846 812 742 801 1040 860 874 848 890 744 749 838 1050 918 986
797 923 975 815 1020 906 901 1170 912 746 919 718 714 740
```

The data are from Cobb (1978) and represent successive measurements of the annual flow of the river Nile (in  $\backslash(10^8\backslash) \backslash(m^3\backslash)$ ) at [Aswan](#), between 1871 and 1970. `nile.txt` contains the flow measurements in a single separated by spaces.

If you open the `sunset-salvo` file with a text editor you will see

The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data

This is a quote extracted from a worth-reading academic paper titled “Sunset Salvo” that has been written by one of the most prominent statisticians, John W. Tukey (Tukey, 1986). The data in this file can be either the letters and spaces or the words themselves, if, for example, one wants to model the frequency or order of letters or words.

## Delimiter-Separated Values

As we discussed, a widespread way to represent structured data is in two-dimensional arrays. For example, the jogging data we encountered in the previous section is a two-dimensional array:

|            | timestamp | latitude | longitude | altitude | distance | heart_rate |
|------------|-----------|----------|-----------|----------|----------|------------|
| speed      |           |          |           |          |          |            |
| 2013-06-01 | 18:40:29  | 50.81381 | -1.712606 | 80.20001 | 1805.94  | 133        |
| 4.060059   |           |          |           |          |          |            |
| 2013-06-01 | 18:40:30  | 50.81383 | -1.712649 | 80.00000 | 1810.00  | 133        |
| 4.550049   |           |          |           |          |          |            |

|            |          |          |           |          |         |     |
|------------|----------|----------|-----------|----------|---------|-----|
| 2013-06-01 | 18:40:31 | 50.81385 | -1.712700 | 79.79999 | 1814.55 | 133 |
| 2.979981   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:32 | 50.81387 | -1.712734 | 79.79999 | 1817.53 | 133 |
| 2.969971   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:33 | 50.81388 | -1.712777 | 79.59998 | 1820.50 | 133 |
| 3.650024   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:34 | 50.81389 | -1.712826 | 79.59998 | 1824.15 | 133 |
| 3.229980   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:35 | 50.81391 | -1.712862 | 79.40002 | 1827.38 | 133 |
| 4.650024   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:36 | 50.81393 | -1.712911 | 79.40002 | 1832.03 | 133 |
| 4.149902   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:37 | 50.81395 | -1.712963 | 79.20001 | 1836.18 | 133 |
| 2.000000   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:38 | 50.81395 | -1.712994 | 79.20001 | 1838.18 | 133 |
| 4.210083   |          |          |           |          |         |     |
| 2013-06-01 | 18:40:39 | 50.81396 | -1.713053 | 79.00000 | 1842.39 | 133 |
| 5.189941   |          |          |           |          |         |     |

Such data structures are ubiquitous; each line is a data record (in our case a record coming from the smartwatch or smartphone) and each of these records consists of the values that one of more variables take.

A common way that such structured data are stored and a shared is in text files containing [delimiter-separated values](#). Such files are organized as follows: each line corresponds to a separate record, and the values of the record are separated by a special character (the *delimiter*), with the most common being the comma, tab, and colon.

For example, if you open the *tab-delimited file* (`running_dat.tsv`) with the above data in a text editor you should see

```
"timestamp" "latitude" "longitude" "altitude" "distance" "heart_rate"
"speed"
2013-06-01 18:40:29 50.813805 -1.7126063 80.2000122 1805.9399512 133
4.06005860000005
2013-06-01 18:40:30 50.8138298 -1.7126487 80 1810.0000098 133
4.55004880000001
2013-06-01 18:40:31 50.8138543 -1.7127005 79.7999878 1814.5500586 133
2.97998050000001
2013-06-01 18:40:32 50.8138709 -1.7127338 79.7999878 1817.5300391 133
2.96997069999998
2013-06-01 18:40:33 50.8138757 -1.7127769 79.5999756 1820.5000098 133
3.65002439999989
2013-06-01 18:40:34 50.8138862 -1.712826 79.5999756 1824.1500342 133
3.22998040000016
2013-06-01 18:40:35 50.8139051 -1.7128616 79.4000244 1827.3800146 133
4.65002449999997
2013-06-01 18:40:36 50.8139326 -1.7129115 79.4000244 1832.0300391 133
4.14990229999989
2013-06-01 18:40:37 50.8139517 -1.7129626 79.2000122 1836.1799414 133
2
2013-06-01 18:40:38 50.8139537 -1.7129945 79.2000122 1838.1799414 133
4.21008299999994
2013-06-01 18:40:39 50.8139622 -1.7130533 79 1842.3900244 133
5.18994140000018
```

If you open the *comma delimited* file (see `running_dat.csv`) with the above data in a text editor you should see

```
"timestamp","latitude","longitude","altitude","distance","heart_rate","speed"
2013-06-01 18:40:29,50.813805,-
1.7126063,80.2000122,1805.9399512,133,4.06005860000005
2013-06-01 18:40:30,50.8138298,-
1.7126487,80,1810.0000098,133,4.55004880000001
2013-06-01 18:40:31,50.8138543,-
1.7127005,79.7999878,1814.5500586,133,2.97998050000001
2013-06-01 18:40:32,50.8138709,-
1.7127338,79.7999878,1817.5300391,133,2.96997069999998
2013-06-01 18:40:33,50.8138757,-
1.7127769,79.5999756,1820.5000098,133,3.65002439999989
2013-06-01 18:40:34,50.8138862,-
1.712826,79.5999756,1824.1500342,133,3.22998040000016
2013-06-01 18:40:35,50.8139051,-
1.7128616,79.4000244,1827.3800146,133,4.65002449999997
2013-06-01 18:40:36,50.8139326,-
1.7129115,79.4000244,1832.0300391,133,4.14990229999989
2013-06-01 18:40:37,50.8139517,-1.7129626,79.2000122,1836.1799414,133,2
2013-06-01 18:40:38,50.8139537,-
1.7129945,79.2000122,1838.1799414,133,4.21008299999994
2013-06-01 18:40:39,50.8139622,-
1.7130533,79,1842.3900244,133,5.18994140000018
```

`running_dat.tsv` is known as a *tab-separated values* file (TSV file, in short) and `running_dat.csv` is a *comma-separated values* file (CSV file, in short).

TSV, and particularly CSV, files are widely used for storing and sharing data, and can be opened by many applications, including most spreadsheet programs (like [Microsoft Excel](#) and [Libre Office](#) Calc), and have excellent support in many programming languages (including R and Python). Despite the C in “CSV file” standing for comma, you may encounter files with “.csv” extension using different delimiters (e.g. colon, semi-colon, or tab).

You may be wondering what happens if one of the data values is or includes the delimiter; this is a well-studied topic known as *delimiter collision* and most modern spreadsheet applications and programming languages try hard to deal with it; see, for example, [Wikipedia’s delimiter page](#) for more information.

# XML

[XML \(Extensible Markup Language\)](#) is a markup language (like Markdown is) that defines a set of rules for encoding information in objects called *XML documents*. XML is a format for organizing structured and semi-structured data in a way that is both human-readable and machine-readable. XML documents are typically stored in plain text files with extension `.xml`.

Since its proposal by [W3C](#) in 1998, the XML format has found a diverse range of uses, ranging from a way to store and share structured and semi-structured data, up to defining other popular file formats. In fact, you are using XML files on a daily basis; for example, Word document files with `.docx` extension from recent versions of Microsoft Office are bundles of many XML documents (see this [blog post](#) for more information).

An example of an XML document is in `statisticians.xml`. This XML document provides information and URLs to the Wikipedia pages for some famous statisticians. If you open `statisticians.xml` in a text editor you should see:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of famous statisticians --&gt;
&lt;records&gt;
  &lt;statistician&gt;
    &lt;name&gt;Ronald&lt;/name&gt;
    &lt;middle&gt;Aylmer&lt;/middle&gt;
    &lt;surname&gt;Fisher&lt;/surname&gt;
    &lt;dob&gt;17/02/1890&lt;/dob&gt;
    &lt;wiki&gt;<a href="https://en.wikipedia.org/wiki/Ronald_Fisher">https://en.wikipedia.org/wiki/Ronald_Fisher</wiki>
  </statistician>
  <statistician>
    <name>William</name>
    <middle>Sealy</middle>
    <surname>Gosset</surname>
    <dob>13/08/1876</dob>
    <wiki>https://en.wikipedia.org/wiki/William\_Sealy\_Gosset</wiki>
  </statistician>
  <statistician>
    <name>David</name>
    <middle>Roxbee</middle>
    <surname>Cox</surname>
    <dob>15/07/1924</dob>
    <wiki>https://en.wikipedia.org/wiki/David\_Cox\_\(statistician\)</wiki>
  </statistician>
  <statistician>
    <name>Thomas</name>
    <middle></middle>
    <surname>Bayes</surname>
    <dob>07/04/1761</dob>
    <wiki>https://en.wikipedia.org/wiki/Thomas\_Bayes</wiki>
  </statistician>
  <statistician>
    <name>Karl</name>
    <middle></middle>
    <surname>Pearson</surname>
    <dob>27/03/1857</dob>
    <wiki>https://en.wikipedia.org/wiki/Karl\_Pearson</wiki>
  </statistician>
```

```

<statistician>
  <name>John</name>
  <middle>Wilder</middle>
  <surname>Tukey</surname>
  <dob>16/06/1915</dob>
  <wiki>https://en.wikipedia.org/wiki/John_Tukey</wiki>
</statistician>
</records>

```

Let's introduce some terminology. Clearly, an XML document is a string of characters. XML documents typically begin with an *XML declaration* like `<?xml version="1.0" encoding="UTF-8"?>` above. Such declarations describe information about the XML document. The characters making up the XML document are divided into *markup* and *content*. Whatever is between `<` and `>` is markup, and anything that is not markup is content. For example, in the line `<surname>Tukey</surname>`, `<surname>` and `</surname>` are markup, while `Tukey` is content. `<surname>` and `</surname>` are instances of start and end *tags*, respectively, which are the main tag types XML supports. Also,

```

<statistician>
  <name>Karl</name>
  <middle></middle>
  <surname>Pearson</surname>
  <dob>27/03/1857</dob>
  <wiki>https://en.wikipedia.org/wiki/Karl_Pearson</wiki>
</statistician>

```

is an *element* and

```
<surname>Pearson</surname>
```

is one of its *child elements*. Finally, XML supports comments by enclosing text between `<!--` and `-->`. For example, `<!-- A list of famous statisticians -->`.

XML is a great format for storing hierarchical data. For example, as is apparent by the statisticians example above, the contents of an XML document can be graphically represented as a tree. The top node is `records`, which has six *children* nodes `statistician`, each of which has five *children* nodes, `name`, `middle`, `surname`, `dob`, and `wiki`.

An `XML parser` is a special bit of software that processes XML documents and passes the structured information within them to another application or on a file. Mainstream programming languages, like R and Python, have packages and modules that allow parsing XML files.

More technical details on XML (e.g. metadata and attributes on tags, XML schemas, etc) and extensions (e.g. XML namespaces) can be found at [Wikipedia's XML page](#).

# JSON

[JavaScript Object Notation](#) (JSON) is a data interchange file format, which, like XML, aims at being both human-readable and machine-readable. JSON was originally used in JavaScript, but many modern programming languages (like R and Python) provide great support for generation and parsing of JSON-format data.

JSON is seeing a diverse range of technological applications, being used for storing and sharing data sets, for communication between different devices and programs (e.g. server-client communication), for interfacing with [APIs](#), etc. In fact, your web browser and the apps on your mobile phone or tablet are creating, processing, sending and receiving hundreds of JSON files every day. One of the reason for its popularity is that it is a lightweight format to store and transmit information, and easy/fast to parse (using a JSON parser).

For example, `statisticians.json` has the same information as `statisticians.xml`. If you open `statisticians.json` in a text editor you should see

```
[  
  {  
    "name": "Ronald",  
    "middle": "Aylmer",  
    "surname": "Fisher",  
    "dob": "17/02/1890",  
    "wiki": "https://en.wikipedia.org/wiki/Ronald_Fisher"  
  },  
  {  
    "name": "William",  
    "middle": "Sealy",  
    "surname": "Gosset",  
    "dob": "13/08/1876",  
    "wiki": "https://en.wikipedia.org/wiki/William_Sealy_Gosset"  
  },  
  {  
    "name": "David",  
    "middle": "Roxbee",  
    "surname": "Cox",  
    "dob": "15/07/1924",  
    "wiki": "https://en.wikipedia.org/wiki/David_Cox_(statistician)"  
  },  
  {  
    "name": "Thomas",  
    "middle": null,  
    "surname": "Bayes",  
    "dob": "07/04/1761",  
    "wiki": "https://en.wikipedia.org/wiki/Thomas_Bayes"  
  },  
  {  
    "name": "Karl",  
    "middle": null,  
    "surname": "Pearson",  
    "dob": "27/03/1857",  
    "wiki": "https://en.wikipedia.org/wiki/Karl_Pearson"  
  },  
  {  
    "name": "John",  
    "middle": "Wilder",  
    "surname": "Wilder",  
    "dob": "01/01/1869",  
    "wiki": "https://en.wikipedia.org/wiki/John_Wilder_Wilder"  
  }]
```

```

        "surname": "Tukey",
        "dob": "16/06/1915",
        "wiki": "https://en.wikipedia.org/wiki/John_Tukey"
    }
]
```

Clearly, this file contains the same information as `statisticians.xml` but in a less verbose manner (less characters; e.g. there are no end tags), sacrificing a bit in terms of detail that can be transmitted along with the file (e.g. no support for comments or metadata, no namespaces, etc.)

JSON supports some basic variable types, including strings, numbers, Booleans, null, arrays, and objects. For example, the below

```
{
    "language": "Python",
    "release": 1991,
    "os": ["Linux", "macOS", "Windows"],
    "oo": true,
    "pastnames": null
}
```

is a JSON object (what is between `{` and `}`), with five *key-value* combinations; e.g. “language” is a key and “Python” is the value. The value of “release” is a number, the value of “os” is an array, the value of “oo” (object-oriented) is a Boolean, and “pastnames” is null.

More details on JSON and extensions can be found at [Wikipedia’s JSON page](#) and at [json.org](#).

## Spreadsheets

A **spreadsheet** is a computer application for the organization, analysis and storage of tabular data. Examples of spreadsheet are Microsoft Excel, LibreOffice Calc or Apple Numbers. They all support a variety of formats for storing tabular data. For example, opening up `running.csv` in Microsoft Excel one sees the following (you get a similar picture if you open `running.csv` in LibreOffice)

	A	B	C	D	E	F	G	H
1	timestamp	latitude	longitude	altitude	distance	heart_rate	speed	
2	01/06/2013 18:40	50.813805	-1.7126063	80.2000122	1805.939951	133	4.0600586	
3	01/06/2013 18:40	50.8138298	-1.7126487		80	1810.00001	133	4.5500488
4	01/06/2013 18:40	50.8138543	-1.7127005	79.7999878	1814.550059	133	2.9799805	
5	01/06/2013 18:40	50.8138709	-1.7127338	79.7999878	1817.530039	133	2.9699707	
6	01/06/2013 18:40	50.8138757	-1.7127769	79.5999756	1820.50001	133	3.6500244	
7	01/06/2013 18:40	50.8138862	-1.712826	79.5999756	1824.150034	133	3.2299804	
8	01/06/2013 18:40	50.8139051	-1.7128616	79.4000244	1827.380015	133	4.6500245	
9	01/06/2013 18:40	50.8139326	-1.7129115	79.4000244	1832.030039	133	4.1499023	
10	01/06/2013 18:40	50.8139517	-1.7129626	79.2000122	1836.179941	133	2	
11	01/06/2013 18:40	50.8139537	-1.7129945	79.2000122	1838.179941	133	4.210083	
12	01/06/2013 18:40	50.8139622	-1.7130533		79	1842.390024	133	5.1899414
13								
14								
15								

Screenshot of Microsoft Excel showing the data in running.csv

The data are opened in a *sheet*, which is part of a *workbook*. A sheet is customarily a two-dimensional array with rows named with numbers and columns named by letters. Every combination of number and letter corresponds to a *cell*, and each value in the data occupies a single cell.

Modern spreadsheet applications allow for the computation of summaries using in-cell formulas, provide data manipulation utilities and tools to carry out a range of statistical analyses and produce visualizations. A description of spreadsheet features is out of the scope of this course. YouTube provides a range of Excel quick tours for a range of levels (e.g. [this one](#) is for absolute beginners and [this one](#) is for more advanced users). Also, LibreOffice provides a range of guides for its applications, including Calc; see [here](#).

Spreadsheets also provide alternative formats for saving tabular data, along with any formulas or visualizations that have been produced. For example, the default file extension for Microsoft Excel is .xlsx, for LibreOffice Calc is .ods (which are both bundles of XML files), and for Apple Numbers is .numbers.

# Other Data Formats

There is a wide variety of other file formats (and database systems!) to store structured, semi-structured, and unstructured data. Many file formats are associated with software for the statistical analysis of data, like SAS, SPSS and MINITAB, or have been invented for particular purposes, such as the [nifti](#) data format that is used in neuroimaging applications.

There are also *binary* data formats, that is data formats where the data are not stored or communicated as text, as is done, for example, for CSV, TSV, XML, and JSON. Examples of such data formats are NASA's HDF5 ([hierarchical data format](#)) and UCAR's netCDF data files ([network common data form](#)), which allow users to store scientific data in array-oriented ways, including descriptions, labels, formats, units, etc. Other binary data formats are formats used for images, such PNG ([portable network graphics](#)) and JPEG ([Joint Photographic Experts Group](#) format) and videos, such as MP4 ([MPEG-4](#)).

# Useful Links and Resources

- [LibreOffice](#): An advanced office suite that is open-source and can be used as an alternative to Microsoft Office; see [here](#) for documentation.
- [Wikipedia's delimiter page](#)
- [Wikipedia's JSON page](#)
- [Wikipedia's XML page](#)
- [Introduction to XML](#): w3schools.com introduction to XML
- [Introduction to JSON](#): w3schools.com introduction to JSON
- [An informal introduction to DOCX](#) by Stepan Yakovenko: An accessible introduction to the DOCX and the underlying usage of XML files.
- [Introducing JSON](#)
- [Learn JSON in 10 minutes](#): A video providing a very good overview of JSON.

# References

- Cobb, G. W. (1978). The problem of the Nile: Conditional solution to a changepoint problem. *Biometrika*, 65(2), 243–251.
- Tukey, J. W. (1986). Sunset salvo. *The American Statistician*, 40(1), 72–76.

# Import/Export of Data-Exchange Files in R

## Extracting Data From Plain Text

### Import

The R function `scan()` can be used to read data from a file. For example, in order to read the measurements in `nile.txt` (see previous section for more details), we do

```
nile <- scan("nile.txt")
nile
[1] 1120 1160 963 1210 1160 1160 813 1230 1370 1140 995 935 1110 994
1020
[16] 960 1180 799 958 1140 1100 1210 1150 1250 1260 1220 1030 1100 774
840
[31] 874 694 940 833 701 916 692 1020 1050 969 831 726 456 824
702
[46] 1120 1100 832 764 821 768 845 864 862 698 845 744 796 1040
759
[61] 781 865 845 944 984 897 822 1010 771 676 649 846 812 742
801
[76] 1040 860 874 848 890 744 749 838 1050 918 986 797 923 975
815
[91] 1020 906 901 1170 912 746 919 718 714 740
```

Note here that we assign the result of `scan("nile.txt")` to a variable called `nile`, and we assume that `nile.txt` is in our working directory (type `getwd()` to see your working directory). If `nile.txt` is not in your working directory, section “Installing and Interacting with R” describes how you can change it. Alternatively, you can replace

`scan("/path/to/nile.txt")`, where `/path/to/` is the path to the directory that `nile.txt` leaves, or, if you are using the terminal, navigating into the directory where `nile.txt` is and starting R from there. In what follows, we assume that all the data files we import data from are in the working directory.

Each call to `scan()` can read a particular type of data. For example, the above line of code read the contents of `nile.txt` as [double precision numbers](#).

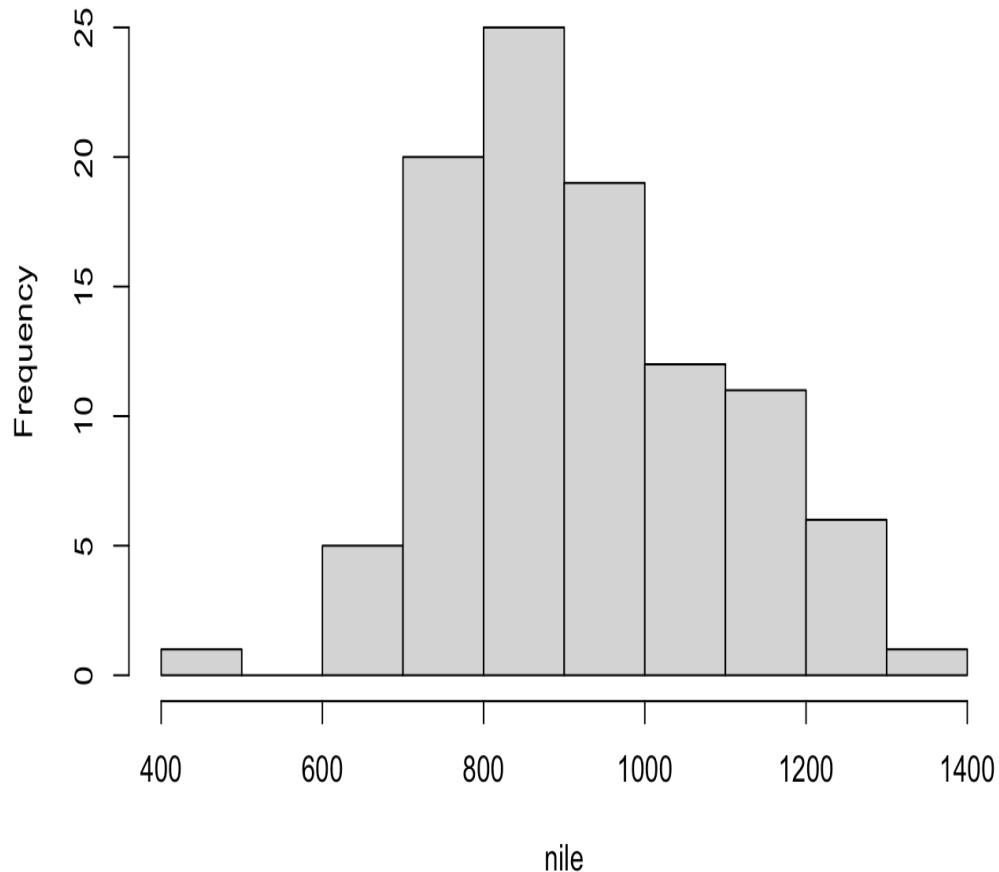
```
typeof(nile)
[1] "double"
```

The `typeof()` function can be used to determine the R internal type or storage mode of any object (here of `nile`).

So, a histogram of the annual flow of the river Nile between 1871 and 1970 is

```
hist(nile)
```

## Histogram of nile



If we want to read the contents of `nile.txt` as characters we do

```
nile_char <- scan("nile.txt", what = character())
nile_char
[1] "1120" "1160" "963" "1210" "1160" "1160" "813" "1230" "1370" "1140"
[11] "995" "935" "1110" "994" "1020" "960" "1180" "799" "958" "1140"
[21] "1100" "1210" "1150" "1250" "1260" "1220" "1030" "1100" "774" "840"
[31] "874" "694" "940" "833" "701" "916" "692" "1020" "1050" "969"
[41] "831" "726" "456" "824" "702" "1120" "1100" "832" "764" "821"
[51] "768" "845" "864" "862" "698" "845" "744" "796" "1040" "759"
[61] "781" "865" "845" "944" "984" "897" "822" "1010" "771" "676"
[71] "649" "846" "812" "742" "801" "1040" "860" "874" "848" "890"
[81] "744" "749" "838" "1050" "918" "986" "797" "923" "975" "815"
[91] "1020" "906" "901" "1170" "912" "746" "919" "718" "714" "740"
typeof(nile_char)
[1] "character"
```

By default, `scan()` expects to read “white-space” delimited files. For reading files with different delimiters, the `sep` argument can be used. For example, the contents of `sunset-salvo-sem` are as follows

```
The;combination;of;some;data;and;an;aching;desire;for;an;answer;does;not;en
sure;that;a;reasonable;answer;can;be;extracted;from;a;given;body;of;data
```

Hence, the words are delimited by semi-colon (;), and we need to tell that to `scan()` in order to extract them:

```
scan("sunset-salvo-sem", what = character(), sep = ";")
[1] "The"           "combination" "of"          "some"        "data"
[6] "and"           "an"          "aching"      "desire"      "for"
[11] "an"            "answer"      "does"        "not"         "ensure"
[16] "that"          "a"           "reasonable" "answer"      "can"
[21] "be"            "extracted"   "from"        "a"           "given"
[26] "body"          "of"          "data"
```

`scan()` is one of the most basic utilities for reading data in R. It may seem simple at first but it offers a wide range of options, including skipping lines, reading only the first few lines of a file and reading different data types in tabular data; for example, we can read the data in `running_dat.tsv` by skipping the first line with the names, specifying that the file is tab-delimited (`sep = "\t"`), and that the first column is a character and the others are numbers (note that the type of data is specified here by simply supplying a `list` of a character and a number to `what`).

```
scan("running_dat.tsv", what = list("", 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0), sep
= "\t", skip = 1)
[[1]]
[1] "2013-06-01 18:40:29" "2013-06-01 18:40:30" "2013-06-01 18:40:31"
[4] "2013-06-01 18:40:32" "2013-06-01 18:40:33" "2013-06-01 18:40:34"
[7] "2013-06-01 18:40:35" "2013-06-01 18:40:36" "2013-06-01 18:40:37"
[10] "2013-06-01 18:40:38" "2013-06-01 18:40:39"

[[2]]
[1] 50.81381 50.81383 50.81385 50.81387 50.81388 50.81389 50.81391
50.81393
[9] 50.81395 50.81395 50.81396

[[3]]
[1] -1.712606 -1.712649 -1.712700 -1.712734 -1.712777 -1.712826 -1.712862
[8] -1.712911 -1.712963 -1.712994 -1.713053

[[4]]
[1] 80.20001 80.00000 79.79999 79.79999 79.59998 79.59998 79.40002
79.40002
[9] 79.20001 79.20001 79.00000

[[5]]
[1] 1805.94 1810.00 1814.55 1817.53 1820.50 1824.15 1827.38 1832.03
1836.18
[10] 1838.18 1842.39

[[6]]
[1] 133 133 133 133 133 133 133 133 133 133

[[7]]
```

```
[1] 4.060059 4.550049 2.979981 2.969971 3.650024 3.229980 4.650024  
4.149902  
[9] 2.000000 4.210083 5.189941
```

## Export

A convenient way to write lines to a file is the function `cat()`. `cat()` is used to output objects either in the standard output connection (typically directly in the R prompt), or to a file. For example,

```
cat(nile, file = "nile1.txt", sep = ",")
```

will create a file called `nile1.txt` in the working directory, where the values stored in `nile` are separated by `,`. See `?cat` for more information. Other useful, low-level functions to export lines whose help pages you may want to check out are `write` and `writeLines`.

# Delimiter-Separated Values

## Import

The R function `read.table()` is the principal means for importing tables from text files containing delimiter-separated values (like CSV files, for example). It uses `scan()` internally but it packs it in a convenient interface. For example, we can import the data in `running_dat.tsv` by doing

```
running <- read.table("running_dat.tsv", header = TRUE, sep = "\t")  
running  
      timestamp latitude longitude altitude distance heart_rate  
speed  
1 2013-06-01 18:40:29 50.81381 -1.712606 80.20001 1805.94      133  
4.060059  
2 2013-06-01 18:40:30 50.81383 -1.712649 80.00000 1810.00      133  
4.550049  
3 2013-06-01 18:40:31 50.81385 -1.712700 79.79999 1814.55      133  
2.979981  
4 2013-06-01 18:40:32 50.81387 -1.712734 79.79999 1817.53      133  
2.969971  
5 2013-06-01 18:40:33 50.81388 -1.712777 79.59998 1820.50      133  
3.650024  
6 2013-06-01 18:40:34 50.81389 -1.712826 79.59998 1824.15      133  
3.229980  
7 2013-06-01 18:40:35 50.81391 -1.712862 79.40002 1827.38      133  
4.650024  
8 2013-06-01 18:40:36 50.81393 -1.712911 79.40002 1832.03      133  
4.149902  
9 2013-06-01 18:40:37 50.81395 -1.712963 79.20001 1836.18      133  
2.000000  
10 2013-06-01 18:40:38 50.81395 -1.712994 79.20001 1838.18      133  
4.210083  
11 2013-06-01 18:40:39 50.81396 -1.713053 79.00000 1842.39      133  
5.189941
```

First, note that we did not need to specify the type of the data as we did with `scan`; indeed, `read.table()` managed to figure out the type correctly, which is clear if we use the `str()` function to get a concise description of the internal structure of the object `running`

```
str(running)
```

The ability to automatically figure out the variable types comes in extremely handy when it comes to large structured data sets or data sets that we are not otherwise familiar with.

Secondly, we tell `read.table()` that the first row of `running_dat.tsv` has the variable names, by setting the argument `header` to `TRUE`, and that `running_dat.tsv` is tab-separated.

We can do the same with comma-separated files. For example,

```
read.table("running_dat.csv", header = TRUE, sep = ",")  
  timestamp latitude longitude altitude distance heart_rate  
speed  
1 2013-06-01 18:40:29 50.81381 -1.712606 80.20001 1805.94 133  
4.060059  
2 2013-06-01 18:40:30 50.81383 -1.712649 80.00000 1810.00 133  
4.550049  
3 2013-06-01 18:40:31 50.81385 -1.712700 79.79999 1814.55 133  
2.979981  
4 2013-06-01 18:40:32 50.81387 -1.712734 79.79999 1817.53 133  
2.969971  
5 2013-06-01 18:40:33 50.81388 -1.712777 79.59998 1820.50 133  
3.650024  
6 2013-06-01 18:40:34 50.81389 -1.712826 79.59998 1824.15 133  
3.229980  
7 2013-06-01 18:40:35 50.81391 -1.712862 79.40002 1827.38 133  
4.650024  
8 2013-06-01 18:40:36 50.81393 -1.712911 79.40002 1832.03 133  
4.149902  
9 2013-06-01 18:40:37 50.81395 -1.712963 79.20001 1836.18 133  
2.000000  
10 2013-06-01 18:40:38 50.81395 -1.712994 79.20001 1838.18 133  
4.210083  
11 2013-06-01 18:40:39 50.81396 -1.713053 79.00000 1842.39 133  
5.189941
```

or simply

```
read.csv("running_dat.csv")  
  timestamp latitude longitude altitude distance heart_rate  
speed  
1 2013-06-01 18:40:29 50.81381 -1.712606 80.20001 1805.94 133  
4.060059  
2 2013-06-01 18:40:30 50.81383 -1.712649 80.00000 1810.00 133  
4.550049  
3 2013-06-01 18:40:31 50.81385 -1.712700 79.79999 1814.55 133  
2.979981  
4 2013-06-01 18:40:32 50.81387 -1.712734 79.79999 1817.53 133  
2.969971  
5 2013-06-01 18:40:33 50.81388 -1.712777 79.59998 1820.50 133  
3.650024  
6 2013-06-01 18:40:34 50.81389 -1.712826 79.59998 1824.15 133  
3.229980
```

```

7 2013-06-01 18:40:35 50.81391 -1.712862 79.40002 1827.38      133
4.650024
8 2013-06-01 18:40:36 50.81393 -1.712911 79.40002 1832.03      133
4.149902
9 2013-06-01 18:40:37 50.81395 -1.712963 79.20001 1836.18      133
2.000000
10 2013-06-01 18:40:38 50.81395 -1.712994 79.20001 1838.18      133
4.210083
11 2013-06-01 18:40:39 50.81396 -1.713053 79.00000 1842.39      133
5.189941

```

In fact, `read.csv()` (for comma-separated values using `.` for decimal points) and `read.csv2()` (for semi-colon-separated values using `,` for decimal points) are convenience functions that are the same as `read.table()` apart from having different default arguments (e.g. `write.csv()` has `sep = ",", dec = "."` and `header = TRUE` by default; see `?read.table` for details and type `read.csv()` in R command to see that it simply calls `read.table()`). The same is true for `read.delim()` (for tab-delimited files using `.` for decimal points) and `read.delim2()` (for tab-delimited files using `,` for decimal points). For example, treating `sunset-salvo-sem` as semi-colon-delimited tabular data gives

```

read.csv2("sunset-salvo-sem", header = FALSE)
      V1      V2 V3   V4   V5   V6 V7     V8      V9 V10 V11     V12 V13 V14
1 The combination of some data and an aching desire for an answer does not
      V15 V16 V17     V18     V19 V20 V21     V22 V23 V24     V25 V26
V27
1 ensure that a reasonable answer can be extracted from a given body
of
      V28
1 data

```

Note here that we had to set `header = FALSE` in order to let `read.csv2()` (and implicitly `read.table()`) know that `sunset-salvo-sem` does not have names and the imported data get the default column names `v1` to `v28`. Otherwise,

```

read.csv2("sunset-salvo-sem")
[1] The combination of some data and
[7] an aching desire for. an.1 answer
[13] does not ensure that a reasonable
[19] answer.1 can be extracted from a.1
[25] given body of.1 data.1
<0 rows> (or 0-length row.names)

```

which imports the words in `sunset-salvo-sem` as the column names and imports no (0 rows of) observations! Now, try to import the data in `running_dat.csv` with `header = FALSE` to see what happens.

[readr](#) is a noteworthy R package when it comes to importing tabular data from files, offering fast reading of tabular data, the option to provide column specifications, and many other features. See the [readr's vignettes](#) for an introduction.

## Export

We also can write data in files with delimiter-separated values. For example, if for some reason we wanted to write the data in `running` in a file with `&&`-delimited values with decimal characters represented by `*` (both quite atypical choices!), we do

```
write.table(running, file = "running.dat", sep = "&&", dec = "*")
```

We can quickly inspect the lines in the file that was generated by doing

```
readLines("running.dat")
[1]
"\\"timestamp\\"\&\\"latitude\\"\&\\"longitude\\"\&\\"altitude\\"\&\\"distance\\"\&\\"
heart_rate\\"\&\\"speed\\""
[2] "\\"1\"&\\"2013-06-01 18:40:29\"&&50*813805&&-
1*7126063&&80*200122&&1805*9399512&&133&&4*06005860000005"
[3] "\\"2\"&\\"2013-06-01 18:40:30\"&&50*8138298&&-
1*7126487&&80&&1810*0000098&&133&&4*55004880000001"
[4] "\\"3\"&\\"2013-06-01 18:40:31\"&&50*8138543&&-
1*7127005&&79*7999878&&1814*5500586&&133&&2*97998050000001"
[5] "\\"4\"&\\"2013-06-01 18:40:32\"&&50*8138709&&-
1*7127338&&79*7999878&&1817*5300391&&133&&2*96997069999998"
[6] "\\"5\"&\\"2013-06-01 18:40:33\"&&50*8138757&&-
1*7127769&&79*5999756&&1820*5000098&&133&&3*65002439999989"
[7] "\\"6\"&\\"2013-06-01 18:40:34\"&&50*8138862&&-
1*712826&&79*5999756&&1824*1500342&&133&&3*22998040000016"
[8] "\\"7\"&\\"2013-06-01 18:40:35\"&&50*8139051&&-
1*7128616&&79*4000244&&1827*3800146&&133&&4*65002449999997"
[9] "\\"8\"&\\"2013-06-01 18:40:36\"&&50*8139326&&-
1*7129115&&79*4000244&&1832*0300391&&133&&4*14990229999989"
[10] "\\"9\"&\\"2013-06-01 18:40:37\"&&50*8139517&&-
1*7129626&&79*2000122&&1836*1799414&&133&&2"
[11] "\\"10\"&\\"2013-06-01 18:40:38\"&&50*8139537&&-
1*7129945&&79*2000122&&1838*1799414&&133&&4*2100829999994"
[12] "\\"11\"&\\"2013-06-01 18:40:39\"&&50*8139622&&-
1*7130533&&79&&1842*3900244&&133&&5*18994140000018"
```

See `?write.table` for more options. There are also the convenience functions `write.csv()` and `write.csv2()` that simply have different default arguments to `write.table()`.

## XML

### Import

There are several ways to import XML data into R, with the most commonly used being the [XML](#) and [xml2](#) R packages.

From the two, [XML](#) is perhaps the most feature-rich providing options for the very fine control of the process of importing and exporting XML files (see the [pages of the Omega project](#) for tutorials and details), but [xml2](#) is particularly straightforward to use, because it takes care of several details relating to XML files automatically (see the [xml2](#) page for details).

After installing [xml2](#) (type `install.packages("xml2")` to do this) we do

```

library("xml2")
stats_people <- read_xml("statisticians.xml")
stats_people
{xml_document}
<records>
[1] <statistician>\n   <name>Ronald</name>\n   <middle>Aylmer</middle>\n
<surn ...>
[2] <statistician>\n   <name>William</name>\n   <middle>Sealy</middle>\n
<surn ...>
[3] <statistician>\n   <name>David</name>\n   <middle>Roxbee</middle>\n
<surna ...>
[4] <statistician>\n   <name>Thomas</name>\n   <middle/>\n
<surname>Bayes</sur ...>
[5] <statistician>\n   <name>Karl</name>\n   <middle/>\n
<surname>Pearson</sur ...>
[6] <statistician>\n   <name>John</name>\n   <middle>Wilder</middle>\n
<surnam ...>

```

Printing the object `stats_people` we see the first tag `<records>` and a printout of its 6 `<statistician>` children. **xml2** provides functions to *interrogate* `stats_people`.

For example, `xml_name()` and `xml_children()` can be used to extract the tag name of `stats_people`, and its children.

```

xml_name(stats_people)
[1] "records"
xml_children(stats_people)
{xml_nodeSet (6)}
[1] <statistician>\n   <name>Ronald</name>\n   <middle>Aylmer</middle>\n
<surn ...>
[2] <statistician>\n   <name>William</name>\n   <middle>Sealy</middle>\n
<surn ...>
[3] <statistician>\n   <name>David</name>\n   <middle>Roxbee</middle>\n
<surna ...>
[4] <statistician>\n   <name>Thomas</name>\n   <middle/>\n
<surname>Bayes</sur ...>
[5] <statistician>\n   <name>Karl</name>\n   <middle/>\n
<surname>Pearson</sur ...>
[6] <statistician>\n   <name>John</name>\n   <middle>Wilder</middle>\n
<surnam ...>

```

If we want to extract the surnames of the famous statistics we first have to find all children with tag `<surname>`. This can be done using XPath (XML Path language) syntax, which allows to select nodes from an XML document (see the [w3schools.com tutorial](http://www.w3schools.com/xml/xpath_intro.asp) for learning XPath). We do

```

surname_nodes <- xml_find_all(stats_people, "./surname")
surname_nodes
{xml_nodeSet (6)}
[1] <surname>Fisher</surname>
[2] <surname>Gosset</surname>
[3] <surname>Cox</surname>
[4] <surname>Bayes</surname>
[5] <surname>Pearson</surname>
[6] <surname>Tukey</surname>

```

where “`“./surname”` is the XPath expression to find the “surname” tag. Then we can extract the surnames as

```
xml_text(surname_nodes)
[1] "Fisher"   "Gosset"   "Cox"       "Bayes"    "Pearson"  "Tukey"
```

The data in `statisticians.xml` can be represented in tabular form, where we have a statistician in each row and the various attributes for each statistician on the columns. The **XML** R package provides a handy function called `xmlToDataFrame()` that allows us to extract data from XML files in tabular form. For example,

```
library("XML")
stats_people_df <- xmlToDataFrame("statisticians.xml")
stats_people_df
  name middle surname      dob
1 Ronald Aylmer Fisher 17/02/1890
2 William Sealy Gosset 13/08/1876
3 David Roxbee Cox 15/07/1924
4 Thomas Bayes 07/04/1761
5 Karl Pearson 27/03/1857
6 John Wilder Tukey 16/06/1915
                                         wiki
1 https://en.wikipedia.org/wiki/Ronald_Fisher
2 https://en.wikipedia.org/wiki/William_Sealy_Gosset
3 https://en.wikipedia.org/wiki/David_Cox_(statistician)
4 https://en.wikipedia.org/wiki/Thomas_Bayes
5 https://en.wikipedia.org/wiki/Karl_Pearson
6 https://en.wikipedia.org/wiki/John_Tukey
```

We can now export `stats_people_df` using `write.table()`, `write.csv()` and `write.csv2()`.

Bear in mind though, that in most cases, it is not possible to represent data in XML files in tabular form. See `?xmlToDataFrame` for which data types are supported and about what `xmlToDataFrame()` can do.

## Export

Objects like `stats_people`, i.e. objects that represent XML documents or nodes (in R terminology, objects of class `xml_document` or `xml_node`) can be written on disk using the `write_xml()` functions from **xml2**. This is useful, for example, if we have an R script whose results need to be exported as XML to be later read by another utility in a data-analytic pipeline.

For example, if we want to remove Thomas Bayes’ record from `stats_people` and write the result on disk we do

```
bayes_record <- xml_children(stats_people) [[4]]
xml_remove(bayes_record)
stats_people
{xml_document}
<records>
[1] <statistician>\n  <name>Ronald</name>\n  <middle>Aylmer</middle>\n<surn ...
```

```

[2] <statistician>\n  <name>William</name>\n    <middle>Sealy</middle>\n<surn ...
[3] <statistician>\n  <name>David</name>\n    <middle>Roxbee</middle>\n<surna ...
[4] <statistician>\n  <name>Karl</name>\n    <middle/>\n<surname>Pearson</sur ...
[5] <statistician>\n  <name>John</name>\n    <middle>Wilder</middle>\n<surnam ...
write_xml(stats_people, "statisticians_no_bayes.xml")
NULL
readLines("statisticians_no_bayes.xml")
[1] "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
[2] "<!-- A list of famous statisticians -->"
[3] "<records>"
[4] "  <statistician>"
[5] "    <name>Ronald</name>"
[6] "    <middle>Aylmer</middle>"
[7] "    <surname>Fisher</surname>"
[8] "    <dob>17/02/1890</dob>"
[9] "    <wiki>https://en.wikipedia.org/wiki/Ronald_Fisher</wiki>"
[10] "  </statistician>"
[11] "  <statistician>"
[12] "    <name>William</name>"
[13] "    <middle>Sealy</middle>"
[14] "    <surname>Gosset</surname>"
[15] "    <dob>13/08/1876</dob>"
[16] "    <wiki>https://en.wikipedia.org/wiki/William_Sealy_Gosset</wiki>"
[17] "  </statistician>"
[18] "  <statistician>"
[19] "    <name>David</name>"
[20] "    <middle>Roxbee</middle>"
[21] "    <surname>Cox</surname>"
[22] "    <dob>15/07/1924</dob>"
[23] "
<wiki>https://en.wikipedia.org/wiki/Thomas_Bayes_(statistician)</wiki>"
[24] "  </statistician>"
[25] "  <statistician>"
[26] "    <name>Karl</name>"
[27] "    <middle/>"
[28] "    <surname>Pearson</surname>"
[29] "    <dob>27/03/1857</dob>"
[30] "    <wiki>https://en.wikipedia.org/wiki/Karl_Pearson</wiki>"
[31] "  </statistician>"
[32] "  <statistician>"
[33] "    <name>John</name>"
[34] "    <middle>Wilder</middle>"
[35] "    <surname>Tukey</surname>"
[36] "    <dob>16/06/1915</dob>"
[37] "    <wiki>https://en.wikipedia.org/wiki/John_Tukey</wiki>"
[38] "  </statistician>"
[39] "</records>"

```

We knew that Thomas Bayes' record was at the fourth node (this is what `[[4]]` above simply extracts the fourth record) by looking at the order the surnames appear when we extracted those earlier.

See, the [xml2 vignette on node modification](#).

# JSON

## Import

The **jsonlite** R package provides a JSON parser and a range of tools for working with JSON documents in R. [jsonlite's CRAN page](#) provides a range of vignettes that demonstrate its functionality, in particular, reading JSON documents from files, getting JSON documents from APIs, combining JSON documents, and more.

If we want to read `statisticians.json`, then we do

```
library("jsonlite")
stats_people <- fromJSON("statisticians.json")
stats_people
  name middle surname      dob
1 Ronald Aylmer Fisher 17/02/1890
2 William Sealy Gosset 13/08/1876
3 David Roxbee Cox 15/07/1924
4 Thomas <NA> Bayes 07/04/1761
5 Karl <NA> Pearson 27/03/1857
6 John Wilder Tukey 16/06/1915
                                         wiki
1 https://en.wikipedia.org/wiki/Ronald_Fisher
2 https://en.wikipedia.org/wiki/William_Sealy_Gosset
3 https://en.wikipedia.org/wiki/David_Cox_(statistician)
4 https://en.wikipedia.org/wiki/Thomas_Bayes
5 https://en.wikipedia.org/wiki/Karl_Pearson
6 https://en.wikipedia.org/wiki/John_Tukey
```

There is also a function `read_json()`, which will read the contents of the JSON document and bring into R as a nested *list* (we will see what this is in later sections), that is somewhat closer to the hierarchy present in a JSON document.

We can also take existing R objects and convert them to JSON

```
running_json <- toJSON(running)
running_json
[{"timestamp": "2013-06-01 18:40:29", "latitude": 50.8138, "longitude": -1.7126, "altitude": 80.2, "distance": 1805.94, "heart_rate": 133, "speed": 4.0601}, {"timestamp": "2013-06-01 18:40:30", "latitude": 50.8138, "longitude": -1.7126, "altitude": 80, "distance": 1810, "heart_rate": 133, "speed": 4.55}, {"timestamp": "2013-06-01 18:40:31", "latitude": 50.8139, "longitude": -1.7127, "altitude": 79.8, "distance": 1814.5501, "heart_rate": 133, "speed": 2.98}, {"timestamp": "2013-06-01 18:40:32", "latitude": 50.8139, "longitude": -1.7127, "altitude": 79.8, "distance": 1817.53, "heart_rate": 133, "speed": 2.97}, {"timestamp": "2013-06-01 18:40:33", "latitude": 50.8139, "longitude": -1.7128, "altitude": 79.6, "distance": 1820.5, "heart_rate": 133, "speed": 3.65}, {"timestamp": "2013-06-01 18:40:34", "latitude": 50.8139, "longitude": -1.7128, "altitude": 79.6, "distance": 1824.15, "heart_rate": 133, "speed": 3.23}, {"timestamp": "2013-06-01 18:40:35", "latitude": 50.8139, "longitude": -1.7129, "altitude": 79.4, "distance": 1827.38, "heart_rate": 133, "speed": 4.65}, {"timestamp": "2013-06-01 18:40:36", "latitude": 50.8139, "longitude": -1.7129, "altitude": 79.4, "distance": 1832.03, "heart_rate": 133, "speed": 4.1499}, {"timestamp": "2013-06-01 18:40:37", "latitude": 50.814, "longitude": -1.713, "altitude": 79.2, "distance": 1836.1799, "heart_rate": 133, "speed": 2}, {"timestamp": "2013-06-01 18:40:38", "latitude": 50.814, "longitude": -1.713, "altitude": 79.2, "distance": 1840.3, "heart_rate": 133, "speed": 2.5} ]
```

```

"mestamp": "2013-06-01 18:40:38", "latitude": 50.814, "longitude": -1.713, "altitude": 79.2, "distance": 1838.1799, "heart_rate": 133, "speed": 4.2101}
, {"timestamp": "2013-06-01 18:40:39", "latitude": 50.814, "longitude": -1.7131, "altitude": 79, "distance": 1842.39, "heart_rate": 133, "speed": 5.1899}]

```

This long string is the *minified* version of the JSON representation of the data in running. We can *prettyify* it (or indent the document) to more clearly see what `toJSON()` does by doing

```

prettify(running_json)
[
  {
    "timestamp": "2013-06-01 18:40:29",
    "latitude": 50.8138,
    "longitude": -1.7126,
    "altitude": 80.2,
    "distance": 1805.94,
    "heart_rate": 133,
    "speed": 4.0601
  },
  {
    "timestamp": "2013-06-01 18:40:30",
    "latitude": 50.8138,
    "longitude": -1.7126,
    "altitude": 80,
    "distance": 1810,
    "heart_rate": 133,
    "speed": 4.55
  },
  {
    "timestamp": "2013-06-01 18:40:31",
    "latitude": 50.8139,
    "longitude": -1.7127,
    "altitude": 79.8,
    "distance": 1814.5501,
    "heart_rate": 133,
    "speed": 2.98
  },
  {
    "timestamp": "2013-06-01 18:40:32",
    "latitude": 50.8139,
    "longitude": -1.7127,
    "altitude": 79.8,
    "distance": 1817.53,
    "heart_rate": 133,
    "speed": 2.97
  },
  {
    "timestamp": "2013-06-01 18:40:33",
    "latitude": 50.8139,
    "longitude": -1.7128,
    "altitude": 79.6,
    "distance": 1820.5,
    "heart_rate": 133,
    "speed": 3.65
  },
  {
    "timestamp": "2013-06-01 18:40:34",
    "latitude": 50.8139,
    "longitude": -1.7128,
    "altitude": 79.6,
    "distance": 1823.49,
    "heart_rate": 133,
    "speed": 3.65
  }
]

```

```

        "distance": 1824.15,
        "heart_rate": 133,
        "speed": 3.23
    },
    {
        "timestamp": "2013-06-01 18:40:35",
        "latitude": 50.8139,
        "longitude": -1.7129,
        "altitude": 79.4,
        "distance": 1827.38,
        "heart_rate": 133,
        "speed": 4.65
    },
    {
        "timestamp": "2013-06-01 18:40:36",
        "latitude": 50.8139,
        "longitude": -1.7129,
        "altitude": 79.4,
        "distance": 1832.03,
        "heart_rate": 133,
        "speed": 4.1499
    },
    {
        "timestamp": "2013-06-01 18:40:37",
        "latitude": 50.814,
        "longitude": -1.713,
        "altitude": 79.2,
        "distance": 1836.1799,
        "heart_rate": 133,
        "speed": 2
    },
    {
        "timestamp": "2013-06-01 18:40:38",
        "latitude": 50.814,
        "longitude": -1.713,
        "altitude": 79.2,
        "distance": 1838.1799,
        "heart_rate": 133,
        "speed": 4.2101
    },
    {
        "timestamp": "2013-06-01 18:40:39",
        "latitude": 50.814,
        "longitude": -1.7131,
        "altitude": 79,
        "distance": 1842.39,
        "heart_rate": 133,
        "speed": 5.1899
    }
]

```

## Export

We can write tabular objects like `running` as JSON to disk, using `jsonlite`'s `write_json()` function. For example,

```
write_json(running, "running.json", pretty = TRUE)
```

will write the tabular data in `running` in a JSON document in `running.json`. `pretty = TRUE` tells `write_json()` to indent the objects in the JSON document before writing.

## Spreadsheets

There are various packages to read spreadsheet formats. For example, the [readxl](#) R package provides excellent out-of-the-box support for reading Microsoft Excel files (with extensions `.xls` for older versions, and `.xlsx` for newer versions of Microsoft Excel). See, [readxl's page](#) for demonstrations. Also, the [writexl](#) allows to write R objects with rectangular data into `.xlsx` files. See, [writexl's page](#) for demonstrations.

`writexl` is part of a collection of packages by the [rOpenSci](#) initiative, which is worth taking a look at.

The [readODS](#) R package provides utilities to read and write Open Document Spreadsheets (with extension `.ods`) files from R. See the package's vignettes for more information.

## Other File Formats

There are additional R and Python packages that provide great support for reading data from a wide variety of file formats for data exchange. See, for example, the file types that the [foreign](#) R package supports, the R's [Data Import/Export documentation](#), and the files that can be read by the [pandas](#) Python package.

## Useful Links and Resources

- [R data import/export documentation](#)
- [pandas IO tools](#)
- [XPath cheat sheet](#) by devhints.io and [w3schools.com XPath tutorial](#).
- [readr](#) R package for importing tabular data from files.
- [XML](#) provides options for the very fine control of the process of importing and exporting XML files; see, also the [pages of the Omega project](#) for tutorials and details.
- [xml2](#) pages
- [jsonlite](#) CRAN page and the vignettes there.
- [readxl](#) and [writexl](#) R packages for reading and writing spreadsheets from Microsoft Excel files.
- [readODS](#) R package provides utilities to read and write Open Document Spreadsheets.

# Data Types in R

## Data Types in R

The most common data types in R are

- Logical
- Character
- Integer
- Numeric
- Factor.

Objects with these data types can be organized into data structures, such as

- Atomic vector
- Matrix
- Array
- List
- Data frame.

## Logical

One of the most useful data types in R (and programming in general) is logical. An object of type *logical* in R has two possible values, TRUE and FALSE. For example, if we do

```
a <- TRUE  
b <- FALSE
```

we have created two objects of type logical with names a and b and values TRUE and FALSE, respectively.

The class of object a is

```
class(a)  
[1] "logical"
```

R provides the key logical operators to carry out [Boolean algebra](#). In particular,

Operator	Description
&	AND (conjunction)
	OR (disjunction)
!	NOT (negation)

So, we can easily reproduce the results of the basic Boolean algebra operations in R

```
a & a  
[1] TRUE
```

```
b & b
[1] FALSE
a & b
[1] FALSE
a | a
[1] TRUE
b | b
[1] FALSE
a | b
[1] TRUE
!a
[1] FALSE
!b
[1] TRUE
```

## Character

A character object represents text (or string) values in R. We use double or single quotation marks to represent text. For example,

```
str1 <- "Data"
str2 <- 'Structures'
class(str1)
[1] "character"
```

We can also combine single and double quotes if we want to make quotes part of the string. For example,

```
str3 <- "in 'R'"
str3
[1] "in 'R'"
```

The `paste()` function can be used to concatenate the strings

```
str <- paste(str1, str2, str3)
print(str)
[1] "Data Structures in 'R'"
```

We can also use the `cat()` function directly. For example

```
cat(str1, str2, str3, "\n")
Data Structures in 'R'
```

Adding "`\n`" tells R to break a line after concatenating the strings. See `?Quotes` for a descriptions of the various “character constants” (also known as special characters) in R.

Of course, forgetting quotation marks can result in errors. For example,

```
str5 <- in 'R'
Error: <text>:1:9: unexpected 'in'
1: str5 <- in
               ^
```

and

```
str6 <- Structures  
Error in eval(expr, envir, enclos): object 'Structures' not found
```

In the last code chunk, R tries, unsuccessfully, to find an object called `Structures` and assign its value to `str6`. Note that this could be potentially dangerous, and definitely not what we had in mind (`str6 <- "Structures"`), if an object named `Structures` actually existed!

## Numeric and Integer

In R, real numbers are represented as `numeric`. For example,

```
x <- 2.5  
class(x)  
[1] "numeric"  
z <- 2  
class(z)  
[1] "numeric"
```

Even though the number 2 is an integer, R assigns `z` above as numeric by default. If we want an integer value we have to explicitly write

```
z <- as.integer(5)  
z  
[1] 5  
class(z)  
[1] "integer"
```

Another popular way to declare an integer in R is appending an `L` to the number

```
z <- 5L  
z  
[1] 5  
class(z)  
[1] "integer"
```

As you may expect, R can be used to do basic arithmetic. Below are the basic arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^` or `**`
- Modulo: `%%`

Below are some examples

```
# Addition  
3 + 2  
[1] 5  
# Subtraction  
5 - 2  
[1] 3
```

```

# Multiplication
5 * 3
[1] 15
5 * -2
[1] -10
5 * 2.5
[1] 12.5
# Division
5 / 3
[1] 1.666667
5 / -2
[1] -2.5
5 / 2.5
[1] 2
# Exponentiation
2^2
[1] 4
5^2.5
[1] 55.9017
5^-2
[1] 0.04
# Modulus
10 %% 3
[1] 1
9 %% 3
[1] 0

```

We can also compare numbers with the result of the comparison being a logical. For example

```

2 > 4
[1] FALSE
exp(pi) < pi^exp(1)
[1] FALSE
log(4) < 4
[1] TRUE
a <- 2.321
a == 2.321
[1] TRUE

```

## Coerce and Test

We can coerce objects from one type to another using the `as.*()` functions. Try to understand what the following functions do, and why the result is as is below.

```

as.character(FALSE)
[1] "FALSE"
as.character(1.123)
[1] "1.123"
as.integer(1.123)
[1] 1
as.logical(4)
[1] TRUE
as.logical(0)
[1] FALSE
as.numeric(1L)
[1] 1
as.numeric("abc")
Warning: NAs introduced by coercion

```

```

[1] NA
as.list(mtcars)
$mpg
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8
19.7
[31] 15.0 21.4

$cyl
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4

$disp
 [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6
275.8
[13] 275.8 275.8 472.0 460.0 440.0 78.7 75.7 71.1 120.1 318.0 304.0
350.0
[25] 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0

$hp
 [1] 110 110 93 110 175 105 245 62 95 123 123 180 180 180 205 215 230
66 52
[20] 65 97 150 150 245 175 66 91 113 264 175 335 109

$drat
 [1] 3.90 3.90 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 3.92 3.07 3.07 3.07
2.93
[16] 3.00 3.23 4.08 4.93 4.22 3.70 2.76 3.15 3.73 3.08 4.08 4.43 3.77 4.22
3.62
[31] 3.54 4.11

$wt
 [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
4.070
[13] 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520 3.435
3.840
[25] 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780

$qsec
 [1] 16.46 17.02 18.61 19.44 17.02 20.22 15.84 20.00 22.90 18.30 18.90
17.40
[13] 17.60 18.00 17.98 17.82 17.42 19.47 18.52 19.90 20.01 16.87 17.30
15.41
[25] 17.05 18.90 16.70 16.90 14.50 15.50 14.60 18.60

$vs
 [1] 0 0 1 1 0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1

$am
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1

$gear
 [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4

$carb
 [1] 4 4 1 1 2 1 4 2 2 4 4 3 3 3 4 4 4 1 2 1 1 2 2 4 2 1 2 2 4 6 8 2

```

We can also test the type of an object using `is.*()` functions. For example,

```
is.character(3)
```

```
[1] FALSE
is.character("hello")
[1] TRUE
is.logical(4)
[1] FALSE
is.logical(TRUE)
[1] TRUE
is.numeric(4)
[1] TRUE
is.numeric(FALSE)
[1] FALSE
is.complex(FALSE)
[1] FALSE
is.complex(43.3)
[1] FALSE
is.complex(43.3i + 2)
[1] TRUE
```

## NA and NaN

NA (not available) is a logical constant in R of length 1 which contains a missing value indicator. For example, if you compute the mean of a character, or try to convert a character to numeric, you get NA

```
mean("a")
Warning in mean.default("a") : argument is not numeric or logical: returning
NA
[1] NA
as.numeric("b")
Warning: NAs introduced by coercion
[1] NA
```

NAs occur often when subsetting vectors, as we see in the next section.

Another special object is [NaN](#) (not available number), which is often confused to the NA. For example, the logarithm and the sqrt of  $\sqrt{-1}$  are NaN, simply because they do not exist in the reals.

```
log(-1)
Warning in log(-1) : NaNs produced
[1] NaN
sqrt(-1)
Warning in sqrt(-1) : NaNs produced
[1] NaN
```

Note though that NaN is always NA, but NA is not always NaN, as the following tests show.

```
is.na(NA)
[1] TRUE
is.nan(NA)
[1] FALSE
is.na(NaN)
[1] TRUE
is.nan(NaN)
[1] TRUE
```

Also, note that almost every operation performed with NA or NaN will return NA or NaN.

```
NA + 1  
[1] NA  
NA + NaN  
[1] NA  
NaN * 3  
[1] NaN  
3^NaN  
[1] NaN
```

but

```
length(NA)  
[1] 1  
length(NaN)  
[1] 1
```

## Useful Links and Resources

- [Chapter 14 in \*R for data science\*](#)
- [Chapter 5 in \*Hands-on programming with R\*](#)

# Data Structures in R



## Vectors

Vectors are one-dimensional arrays that can hold data. They are simple objects to store numeric data, character data and logical data. Bear in mind that a vector can contain only one single data type. You cannot have different data types inside the same vector.

In R, we can create a vector with the combine function `c()`. The vector elements are separated by a comma inside the parentheses. For example,

```
marks <- c(80, 75, 90, 99, 100)
names <- c("John", "Jane", "Richard", "Emma")
logicals <- c(FALSE, TRUE, TRUE, FALSE, FALSE, TRUE)
marks
[1] 80 75 90 99 100
names
[1] "John"      "Jane"       "Richard"    "Emma"
logicals
[1] FALSE  TRUE  TRUE FALSE FALSE  TRUE
class(marks)
[1] "numeric"
class(names)
[1] "character"
class(logicals)
[1] "logical"
```

We can even give names to the elements of our vectors. This is very useful to keep track of what each element represents.

To name elements of a vector, you can use the `names()` function. For example,

```
# Creating a simple character vector
Full_Name <- c("John", "Doe")
Full_Name
[1] "John" "Doe"
# Naming the vector
names(Full_Name) <- c("First Name", "Last Name")
Full_Name
First Name  Last Name
```

```

"John"      "Doe"
# Creating a simple numeric vector
marks <- c(80, 75, 90, 99, 100)
marks
[1] 80 75 90 99 100
# Naming the vector
names(marks) <- c("John", "Jane", "Richard", "Emma", "Tim")
marks
John      Jane Richard      Emma      Tim
80        75       90        99       100

```

Now that we know how to create vectors, we can perform calculations with them. These are performed element-wise. Here is an example:

```

# Creating two simple numeric vectors
x <- c(2, 3, 4)
y <- c(5, 6, 7)
# Addition
x + y
[1] 7 9 11
# Subtraction
x - y
[1] -3 -3 -3
# Multiplication
x * y
[1] 10 18 28
# Division
y / x
[1] 2.50 2.00 1.75
# Exponentiation
y ^ x
[1] 25 216 2401
# Modulo
y %% x
[1] 1 0 3

```

In addition, here are some useful functions that can be used with vectors:

- `length()`: Returns the length of the vector.
- `sum()`: Returns the sum of the elements of the vector.
- `min()`: Returns the lowest value found inside the vector.
- `max()`: Returns the highest value found inside the vector.
- `mean()`: Returns the mean of the elements of the vector.
- `median()`: Returns the median of the elements of the vector.
- `sd()`: Returns the standard deviation of the elements of the vector.
- `var()`: Returns the variance of the elements of the vector.
- `cov()`: Returns the covariance of two vectors.
- `cor()`: Returns the correlation of two vectors.
- `sort()`: Sorts a vector into ascending or descending order.

Here is a demonstration using the `marks` vector we previously created.

```

marks
John      Jane Richard      Emma      Tim
80        75       90        99       100
length(marks)

```

```

[1] 5
sum(marks)
[1] 444
min(marks)
[1] 75
max(marks)
[1] 100
mean(marks)
[1] 88.8
median(marks)
[1] 90
sd(marks)
[1] 11.16692
var(marks)
[1] 124.7
marks2 <- c(85, 50, 64, 95, 45)
marks2
[1] 85 50 64 95 45
cov(marks, marks2)
[1] 27.95
cor(marks, marks2)
[1] 0.1152433
sort(marks, decreasing = FALSE)
  Jane   John Richard   Emma   Tim
  75     80      90     99    100
sort(marks, decreasing = TRUE)
  Tim   Emma Richard   John   Jane
  100    99      90     80     75

```

We can even make element-wise comparisons between vectors using relational operators. Let us compare average daily weekday temperatures for 2 weeks, namely `temperature_week_1` and `temperature_week_2` respectively:

```

temperature_week_1 <- c(10,11,12,15,13)
temperature_week_2 <- c(10,9,13,15,16)
temperature_week_1
[1] 10 11 12 15 13
temperature_week_2
[1] 10 9 13 15 16
temperature_week_1 < temperature_week_2
[1] FALSE FALSE TRUE FALSE TRUE
temperature_week_1 <= temperature_week_2
[1] TRUE FALSE TRUE TRUE TRUE
temperature_week_1 > temperature_week_2
[1] FALSE TRUE FALSE FALSE FALSE
temperature_week_1 >= temperature_week_2
[1] TRUE TRUE FALSE TRUE FALSE
temperature_week_1 == temperature_week_2
[1] TRUE FALSE FALSE TRUE FALSE
temperature_week_1 != temperature_week_2
[1] FALSE TRUE TRUE FALSE TRUE

```

Now, suppose we want to select specific elements of a vector. R lets us do this using square brackets [ ]:

```

# Access first element
marks[1]
John
80

```

```

# Access second element
marks[2]
Jane
75
# Access last element
marks[length(marks)]
Tim
100
# Access first two elements
marks[1:2]
John Jane
80 75
# Access second to fourth element
marks[2:4]
  Jane Richard    Emma
  75      90      99
# Access second to fourth element (alternative method)
marks[c(2,3,4)]
  Jane Richard    Emma
  75      90      99
# Access first, third and fifth elements of marks vector
marks[c(1,3,5)]
  John Richard    Tim
  80      90     100
# Access elements of marks vector using names
marks[c("John", "Jane", "Tim")]
John Jane Tim
80 75 100

```

Finally, you can also pass a logical vector inside square brackets. R will only return elements that correspond to TRUE. For week 1, suppose we only want the day(s) where the temperature was higher than week 2. Here is how to do it:

```

weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
# Naming temperature vectors
names(temperature_week_1) <- weekdays
names(temperature_week_2) <- weekdays
# Making comparisons
temperature_week_1 > temperature_week_2
  Monday   Tuesday Wednesday Thursday     Friday
  FALSE      TRUE     FALSE    FALSE     FALSE
# Selecting based on condition
temperature_week_1[temperature_week_1 > temperature_week_2]
Tuesday
11

```

# Factors

Data often fall into a limited number of categories. For instance, cars can be categorized according to their brand, such as Audi, Jaguar, Mercedes, BMW and so on. Categorical data in R are typically stored in “factors.” Factors are very important, especially when visualizing data and building models, so it is important to learn about them now.

We use the `factor()` function to create factors in R. For example,

```
colour_vector <- c("blue", "red", "green", "green", "blue", "green",  
"yellow", "grey")  
class(colour_vector)  
[1] "character"  
colour_vector_factor <- factor(colour_vector)  
class(colour_vector_factor)  
[1] "factor"  
colour_vector_factor  
[1] blue   red    green  green  blue   green  yellow grey  
Levels: blue green grey red yellow
```

We first construct a character vector called `colour_vector` with five character values “red,” “blue,” “yellow,” “green” and “grey.” Using the `factor()` function, we convert it into a factor variable called `colour_vector_factor`. After printing `colour_vector_factor`, R conveniently displays the levels (or the categories) of our factor variable.

There are two types of categorical variables, namely “ordinal categorical variables” and “nominal categorical variables.” Nominal categorical variables have categories without an implied order, that is, it is not possible to say that “one category ranks higher or lower than the other.” An example is hair colour. On the other hand, ordinal categorical variables have categories with a natural ordering. For instance, suppose the categorical variable `customer_satisfaction` has the following categories; “Low,” “Medium” and “High.” Then, clearly, “Low” ranks below “Medium,” which in turn ranks below “High.”

Sometimes, the factor levels don’t have entirely meaningful names. For example, we might have “L,” “M” and “H” instead of “Low”, “Medium” and “High.” Fortunately, R lets us change the names of the levels (or categories) using the `levels()` function as follows:

```
customer_satisfaction <- factor(c("L", "M", "L", "L", "H"))  
customer_satisfaction  
[1] L M L L H  
Levels: H L M  
# Renaming factor levels  
levels(customer_satisfaction) <- c("High", "Low", "Medium")  
customer_satisfaction  
[1] Low   Medium Low    Low    High  
Levels: High Low Medium
```

Note that we did not specify the order of the levels. R automatically ranked the categories in alphabetical order. This is why “H” is before “L” and “L” is before “M.” When using the `levels()` function, we should be careful to respect this ordering, else R will not name the levels correctly.

To explicitly instruct R to order the factors, we should set `ordered = TRUE` when calling the `factor()` function. We can also specify the levels in the same command. Here are some examples:

```
speed <- c("fast", "slow", "slow", "fast", "medium")
speed_factor <- factor(speed, ordered = TRUE, levels = c("slow", "medium",
"fast"))
speed_factor
[1] fast   slow   slow   fast   medium
Levels: slow < medium < fast
speed_factor[1] > speed_factor[2]
[1] TRUE
speed_factor[4] > speed_factor[5]
[1] TRUE
speed_factor[3] > speed_factor[4]
[1] FALSE
summary(speed_factor)
  slow medium   fast
    2      1      2
```

Here, we create a factor variable called `speed_factor` with three levels, namely “slow,” “medium” and “fast” in that order. We then compare a few elements of the `speed_factor` variable to see whether they rank above or below each other. Finally, we use the `summary()` function to obtain a convenient summary of our factor.

## Matrices

A matrix is similar to a vector in the sense that it holds elements of the same data type. However, a matrix is two-dimensional whereas a vector is only one-dimensional. This means that the elements of a matrix are arranged into a number of rows and columns. You cannot have different data types inside the same matrix.

We use the `matrix()` function to create a matrix in R.

Here are some examples:

```
# 3 x 4 matrix, byrow = TRUE
my_matrix <- matrix(1:12, byrow = TRUE, nrow = 3)
# 3 x 4 matrix, byrow = FALSE
my_matrix_1 <- matrix(1:12, byrow = FALSE, nrow = 3)
# 4 x 3 matrix, byrow = FALSE
my_matrix_2 <- matrix(1:12, byrow = FALSE, nrow = 4)
my_matrix
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
my_matrix_1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
my_matrix_2
     [,1] [,2] [,3]
[1,]    1    5    9
```

```
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

You can also create matrices from a collection of vectors. Here, we will be using the temperature vectors from the previous subsection:

```
temperature_week_1
Monday   Tuesday Wednesday Thursday     Friday
      10        11       12       15       13
temperature_week_2
Monday   Tuesday Wednesday Thursday     Friday
      10        9        13       15       16
# Combine temperature_week_1 and temperature_week_2 into a single vector
temperature_combined <- c(temperature_week_1, temperature_week_2)
temperature_combined
Monday   Tuesday Wednesday Thursday     Friday     Monday   Tuesday
Wednesday
      10        11       12       15       13       10        9
13
Thursday     Friday
      15        16
# Converting into a matrix
temperature_week_1_and_2 <- matrix(temperature_combined, byrow = FALSE,
nrow = 2)
temperature_week_1_and_2
[,1] [,2] [,3] [,4] [,5]
[1,] 10   12   13   9   15
[2,] 11   15   10   13   16
# R does not throw any error, but does the above matrix make sense in our
context? What is wrong?
# Let us try setting byrow = TRUE
temperature_week_1_and_2 <- matrix(temperature_combined, byrow = TRUE, nrow
= 2)
temperature_week_1_and_2
[,1] [,2] [,3] [,4] [,5]
[1,] 10   11   12   15   13
[2,] 10   9    13   15   16
# This matrix makes more sense as the data for each week is shown on a
separate row
```

We can also use the `rownames()` and `colnames()` functions to give names to the rows and the columns of a matrix. For example,

```
temperature_week_1_and_2
[,1] [,2] [,3] [,4] [,5]
[1,] 10   11   12   15   13
[2,] 10   9    13   15   16
weeks <- c("Week 1", "Week 2")
weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
rownames(temperature_week_1_and_2) <- weeks
colnames(temperature_week_1_and_2) <- weekdays
temperature_week_1_and_2
Monday Tuesday Wednesday Thursday Friday
Week 1      10        11       12       15       13
Week 2      10        9        13       15       16
```

Just like vectors, we can perform a lot of interesting calculations on matrices. In addition to the functions introduced in the previous subsection, here are four additional functions that are useful when working with matrices:

- `rowSums()`: Returns a vector where each element represents the total of the values of the corresponding row.
- `colSums()`: Returns a vector where each element represents the total of the values of the corresponding column.
- `rbind()`: Adds a row or multiple rows to a matrix. It can also be used to merge multiple matrices or vectors together by row.
- `cbind()`: Adds a column or multiple column to a matrix. It can also be used to merge multiple matrices or vectors together by column.

In the following demo, we are interested in the daily earnings of two merchants, John and Jane over a period of one week. First, we create an earnings vector for John (`earnings_John`) and Jane (`earnings_Jane`), respectively. Then, we combine them into a named matrix (`earnings_combined`) and we display the results:

```
# Create earnings vector for merchant John
earnings_John <- c(50, 60, 55, 74, 80)
# Create earnings vector for merchant Jane
earnings_Jane <- c(53, 57, 79, 88, 93)
# Create vector containing names of the days of the week
weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
# Create vector containing names of the merchants
merchant_names <- c("John", "Jane")
# Creating a matrix to store the merchants' earnings together
earnings_combined <- matrix(c(earnings_John, earnings_Jane), byrow = TRUE,
nrow = 2)
# Naming a matrix
rownames(earnings_combined) <- merchant_names
colnames(earnings_combined) <- weekdays
# Viewing matrix
earnings_combined
    Monday Tuesday Wednesday Thursday Friday
John      50       60       55       74       80
Jane      53       57       79       88       93
```

If we want to add the earnings for Saturday and Sunday, we first create a named matrix to store the weekend earnings (`earnings_combined_weekend`) and then merge it with `earnings_combined` using `cbind()` to create a new matrix called `earnings_whole_week`. As the name suggests, the latter contains the earnings for Jane and John for the entire week.

```
# Creating earnings vectors for weekend
earnings_John_weekend <- c(110, 120)
earnings_Jane_weekend <- c(100, 130)
weekends <- c("Saturday", "Sunday")
merchant_names
[1] "John" "Jane"
earnings_combined_weekend <- matrix(c(earnings_John_weekend,
earnings_Jane_weekend), byrow = TRUE, nrow = 2)
earnings_combined_weekend
 [,1] [,2]
[1,] 110 120
[2,] 100 130
```

```

# Assigning row and column names
rownames(earnings_combined_weekend) <- merchant_names
colnames(earnings_combined_weekend) <- weekends
# Viewing weekend matrix
earnings_combined_weekend
  Saturday Sunday
John      110     120
Jane      100     130
# Viewing weekday matrix
earnings_combined
  Monday Tuesday Wednesday Thursday Friday
John      50       60       55       74       80
Jane      53       57       79       88       93
# We now merge them together
earnings_whole_week <- cbind(earnings_combined, earnings_combined_weekend)
earnings_whole_week
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
John      50       60       55       74       80      110      120
Jane      53       57       79       88       93      100      130

```

Now, suppose we want to add a third merchant called Tim. We can do so using the `rbind()` function:

```

# Create earnings vector for Tim
earnings_Tim <- c(40, 48, 75, 65, 29, 67, 84)
# Add earnings_Tim to earnings_whole_week
earnings_whole_week <- rbind(earnings_whole_week, earnings_Tim)
# Naming rows
rownames(earnings_whole_week) <- c("John", "Jane", "Tim")
# Viewing updated matrix
earnings_whole_week
  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
John      50       60       55       74       80      110      120
Jane      53       57       79       88       93      100      130
Tim       40       48       75       65       29       67       84

```

Finally, suppose we want to find the:

- Total earnings for the three merchants per day.
- Total earnings for each merchant for the whole week.

We can find these using the `colSums()` and `rowSums()` functions respectively:

```

# Total earnings for the three merchants per day
total_earnings_per_day <- colSums(earnings_whole_week)
total_earnings_per_day
  Monday   Tuesday   Wednesday   Thursday   Friday   Saturday   Sunday
    143      165      209      227      202      277      334
# Total earnings for each merchant for the whole week
total_earnings_for_week <- rowSums(earnings_whole_week)
total_earnings_for_week
John  Jane  Tim
  549   600   408

```

Just like vectors, we can also select specific matrix elements. Here are some examples:

```

# Select only first row
John_Only <- earnings_whole_week[1,]

```

```

John_Only
Monday   Tuesday Wednesday Thursday Friday Saturday Sunday
      50       60      55       74      80      110      120
# Select only first and third rows
John_and_Tim_only <- earnings_whole_week[c(1,3),]
John_and_Tim_only
Monday   Tuesday Wednesday Thursday Friday Saturday Sunday
John      50       60      55       74      80      110      120
Tim       40       48      75       65      29       67      84
# Select only third column
Wednesday_only <- earnings_whole_week[,3]
Wednesday_only
John Jane Tim
  55    79   75
# Selecting only fourth to seventh columns
Thursday_to_Sunday_only <- earnings_whole_week[,4:7]
Thursday_to_Sunday_only
Thursday Friday Saturday Sunday
John      74       80      110      120
Jane     88       93      100      130
Tim      65       29       67      84
# Selecting only fourth to sixth columns and first and third rows
selection <- earnings_whole_week[c(1,3), 4:7]
selection
Thursday Friday Saturday Sunday
John      74       80      110      120
Tim      65       29       67      84
# Select by name
selection_1 <- earnings_whole_week[c("John", "Tim"), c("Thursday",
"Friday", "Saturday", "Sunday")]
selection_1
Thursday Friday Saturday Sunday
John      74       80      110      120
Tim      65       29       67      84

```

Finally, just like vectors, we can perform element-wise arithmetic operations with matrices:

```

mat1 <- matrix(1:4, nrow = 2)
mat2 <- matrix(5:8, nrow = 2)
mat1
 [,1] [,2]
[1,] 1 3
[2,] 2 4
mat2
 [,1] [,2]
[1,] 5 7
[2,] 6 8
mat1 + mat2
 [,1] [,2]
[1,] 6 10
[2,] 8 12
mat2 - mat1
 [,1] [,2]
[1,] 4 4
[2,] 4 4
mat1 * mat2
 [,1] [,2]
[1,] 5 21
[2,] 12 32
mat1 / mat2

```

```

[,1]      [,2]
[1,] 0.2000000 0.4285714
[2,] 0.3333333 0.5000000
mat1 ^ mat2
[,1]      [,2]
[1,]      1 2187
[2,]     64 65536

```

Note that R performed the matrix multiplication element-wise. This is not the standard way that matrices are multiplied. To perform matrix multiplication in the standard way, we can use the `%*%` operator like this:

```

A <- matrix(c(2,4,6,8), nrow = 2)
B <- matrix(c(1,3,4,7,9,10), nrow = 2)
A
[,1] [,2]
[1,]    2    6
[2,]    4    8
B
[,1] [,2] [,3]
[1,]    1    4    9
[2,]    3    7   10
# Standard matrix multiplication to calculate AB
AB <- A %*% B
AB
[,1] [,2] [,3]
[1,]   20   50   78
[2,]   28   72  116

```

## Arrays

An array is the multi-dimensional extension of matrices in R. In fact, a matrix can be viewed as a two-dimensional array. Just like vectors and matrices, an array can only store one data type at a time.

To create an array, we use the `array()` function. The two main arguments of the `array()` function are:

- `data`: This is a vector which contains the elements to be stored in the array.
- `dim`: This specifies the dimension of the array. For example, if we set `dim = c(4, 3, 2)`, we are telling R to create an array consisting of 2 matrices, each with 4 rows and 3 columns.
- `dimnames`: This optional argument specifies the names of the dimensions.

Again, you cannot have different data types inside the same matrix.

Here is a simple example of how to create an array:

```

# Create a vector of input elements
vector <- c(1,2,3,5,7,1,3,6,7,8,9,9)
# Use the above vector to create an array of two 3x2 matrices
my_array <- array(vector, dim = c(3,2,2))
my_array
, , 1

```

```
[,1] [,2]
[1,]    1    5
[2,]    2    7
[3,]    3    1
```

```
, , 2
```

```
[,1] [,2]
[1,]    3    8
[2,]    6    9
[3,]    7    9
```

Suppose there are only 3 students in a school. Suppose further that there are 3 terms in an academic year and that each student takes 5 tests during each term. The names of the students are “John,” “Jane” and “Tim.” Assume that we have 9 vectors of length 5, each containing the test results for one student for one term.

```
John_Term1 <- c(74, 72, 71, 79, 90)
John_Term2 <- c(78, 80, 80, 88, 82)
John_Term3 <- c(85, 90, 72, 77, 86)
Jane_Term1 <- c(63, 43, 62, 85, 65)
Jane_Term2 <- c(60, 27, 74, 63, 57)
Jane_Term3 <- c(55, 72, 64, 47, 75)
Tim_Term1 <- c(81, 83, 90, 84, 94)
Tim_Term2 <- c(99, 77, 99, 87, 91)
Tim_Term3 <- c(80, 94, 95, 87, 80)
```

The above code block is not complete. Complete it by constructing the 9 vectors using the above names, using some numeric values for the test results.

Currently, this data is stored in 9 different objects (9 different vectors). We want to store them in one single object, that is, we want to store them in one single array. There are many ways we can define the dimensions of our array but a natural choice is to store the values in an array consisting of three  $(5 \times 3)$  matrices, where each matrix represents the marks for one student only. Inside a matrix, each row will represent a test and each column will correspond to a term.

```
marks_combined <- array(c(John_Term1, John_Term2, John_Term3,
                           Jane_Term1, Jane_Term2, Jane_Term3,
                           Tim_Term1, Tim_Term2, Tim_Term3),
                           dim = c(5, 3, 3))
marks_combined
, , 1

[,1] [,2] [,3]
[1,] 74   78   85
[2,] 72   80   90
[3,] 71   80   72
[4,] 79   88   77
[5,] 90   82   86

, , 2

[,1] [,2] [,3]
[1,] 63   60   55
[2,] 43   27   72
```

```
[3,]   62   74   64
[4,]   85   63   47
[5,]   65   57   75
```

```
, , 3
```

```
[,1] [,2] [,3]
[1,]   81   99   80
[2,]   83   77   94
[3,]   90   99   95
[4,]   84   87   87
[5,]   94   91   80
```

The dimension of an array (and that of matrices!) can be found as

```
dim(marks_combined)
[1] 5 3 3
```

We can use the `dimnames` argument of the `array()` function to add names to the dimensions of the array.

```
matrix_names <- c("John", "Jane", "Tim")
row_names <- c("Test 1", "Test 2", "Test 3", "Test 4", "Test 5")
column_names <- c("Term 1", "Term 2", "Term 3")
# Let us create an upgraded version of our array
marks_combined_upgraded <- array(c(John_Term1, John_Term2, John_Term3,
                                    Jane_Term1, Jane_Term2, Jane_Term3,
                                    Tim_Term1, Tim_Term2, Tim_Term3),
                                    dim = c(5, 3, 3),
                                    dimnames = list(row_names, column_names,
matrix_names))
marks_combined_upgraded
, , John

      Term 1 Term 2 Term 3
Test 1     74    78    85
Test 2     72    80    90
Test 3     71    80    72
Test 4     79    88    77
Test 5     90    82    86

, , Jane

      Term 1 Term 2 Term 3
Test 1     63    60    55
Test 2     43    27    72
Test 3     62    74    64
Test 4     85    63    47
Test 5     65    57    75

, , Tim

      Term 1 Term 2 Term 3
Test 1     81    99    80
Test 2     83    77    94
Test 3     90    99    95
Test 4     84    87    87
Test 5     94    91    80
```

Just like with vectors and matrices, we can access array elements in the usual way by using square brackets [ ]:

```
# Select first matrix only
marks_combined_upgraded[, , 1]
  Term 1 Term 2 Term 3
Test 1     74     78     85
Test 2     72     80     90
Test 3     71     80     72
Test 4     79     88     77
Test 5     90     82     86
# Select Tim's matrix only (using name instead of index)
marks_combined_upgraded[, , "Tim"]
  Term 1 Term 2 Term 3
Test 1     81     99     80
Test 2     83     77     94
Test 3     90     99     95
Test 4     84     87     87
Test 5     94     91     80
# Selecting John and Tim's matrices
marks_combined_upgraded[, , c("John", "Tim")]
, , John

  Term 1 Term 2 Term 3
Test 1     74     78     85
Test 2     72     80     90
Test 3     71     80     72
Test 4     79     88     77
Test 5     90     82     86

, , Tim

  Term 1 Term 2 Term 3
Test 1     81     99     80
Test 2     83     77     94
Test 3     90     99     95
Test 4     84     87     87
Test 5     94     91     80
# Selecting only Term 1 and Term 3 marks
marks_combined_upgraded[, c("Term 1", "Term 3"), ]
, , John

  Term 1 Term 3
Test 1     74     85
Test 2     72     90
Test 3     71     72
Test 4     79     77
Test 5     90     86

, , Jane

  Term 1 Term 3
Test 1     63     55
Test 2     43     72
Test 3     62     64
Test 4     85     47
Test 5     65     75

, , Tim
```

```

      Term 1 Term 3
Test 1     81     80
Test 2     83     94
Test 3     90     95
Test 4     84     87
Test 5     94     80
# Selecting John and Tim's Test 3 results for Term 2
marks_combined_upgraded["Test 2", "Term 2", c("John", "Tim")]
John   Tim
80     77

```

Finally, we can perform calculations across the array elements using the `apply()` function. This function takes three arguments:

- `x`: An array (or a matrix) containing the data we are interested in.
- `MARGIN`: This specifies whether the function will be applied on rows only (set `MARGIN = c(1)`), columns only (set `MARGIN = c(2)`), or both rows and columns (set `MARGIN = c(1,2)`).
- `FUN`: This specifies the function to be applied. It can be any compatible function. Examples include but are not limited to: `sum()`, `min()`, `max()`, `mean()`, `median()`, `sd()`, `var()`.

In the following examples, we use the `sum()` function to calculate various interesting totals.

```

# Find total marks for all students, across all tests and terms
sum(marks_combined_upgraded)
[1] 3437
# Find total marks for all students per term
apply(marks_combined_upgraded, c(2), sum)
Term 1 Term 2 Term 3
1136    1142    1159
# Find the total marks for all students per test per term
apply(marks_combined_upgraded, c(1,2), sum)
      Term 1 Term 2 Term 3
Test 1     218    237    220
Test 2     198    184    256
Test 3     223    253    231
Test 4     248    238    211
Test 5     249    230    241
# (Less useful in this context) Find the total marks for all students per test
apply(marks_combined_upgraded, c(1), sum)
Test 1 Test 2 Test 3 Test 4 Test 5
675     638     707     697     720

```

# Lists

Lists can hold elements of different data types. They allow us to store multiple objects under one single object in an orderly fashion. Lists can store vectors, matrices, data frames and even other lists.

To create a list, we use the `list()` function. The arguments of the `list()` function represent the components of the list.

Below is an example where we create a list containing a vector, a matrix and a data frame. We create the data frame `my_data_frame` by selecting the first 8 rows of the build-in `mtcars` dataset.

```
# Creating a simple vector
my_vector <- 1:10
# Creating a simple matrix
my_matrix <- matrix(c(4, 6, 7, 1), nrow = 2)
# Creating a simple data frame
my_data_frame <- mtcars[1:8, ]
my_vector
[1] 1 2 3 4 5 6 7 8 9 10
my_matrix
[,1] [,2]
[1,]    4    7
[2,]    6    1
my_data_frame
      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant       18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360    14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D     24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
# Creating a list from the above variables
my_list <- list(my_vector, my_matrix, my_data_frame)
my_list
[[1]]
[1] 1 2 3 4 5 6 7 8 9 10

[[2]]
[,1] [,2]
[1,]    4    7
[2,]    6    1

[[3]]
      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant       18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360    14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D     24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
```

Again, if we want to name the components of our list, we use the `names()` function:

```
my_names <- c("I am a Vector",
             "I am a Matrix",
             "I am a Data Frame")
names(my_list) <- my_names
my_list
$I am a Vector
[1] 1 2 3 4 5 6 7 8 9 10

$I am a Matrix
[,1] [,2]
[1,] 4 7
[2,] 6 1

$I am a Data Frame
  mpg cyl disp hp drat wt qsec vs am gear carb
Mazda RX4     21.0   6 160.0 110 3.90 2.620 16.46 0 1 4 4
Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02 0 1 4 4
Datsun 710    22.8   4 108.0  93 3.85 2.320 18.61 1 1 4 1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44 1 0 3 1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02 0 0 3 2
Valiant      18.1   6 225.0 105 2.76 3.460 20.22 1 0 3 1
Duster 360    14.3   8 360.0 245 3.21 3.570 15.84 0 0 3 4
Merc 240D     24.4   4 146.7  62 3.69 3.190 20.00 1 0 4 2
```

Now, it's time to learn how to access the components of our list. We use:

The `$` sign to access a component by name. Double square brackets `[[ ]]` to access a component by index or by name. Any components in the components of the list can, then, be further accessed according to their type.

Here is a demonstration of this:

```
# Selecting first component by index
my_list[[1]]
[1] 1 2 3 4 5 6 7 8 9 10
# Selecting first component by name
my_list$I am a Vector
[1] 1 2 3 4 5 6 7 8 9 10
# Selecting second component by name using the $ notation
my_list$I am a Matrix
[,1] [,2]
[1,] 4 7
[2,] 6 1
# Selecting second component by name using square brackets
my_list[["I am a Matrix"]]
[,1] [,2]
[1,] 4 7
[2,] 6 1
# Selecting fourth element of first component by index only
my_list[[1]][4]
[1] 4
# Selecting fourth element of first component by name and index
my_list$I am a Vector'[4]
[1] 4
# Selecting first row of second component by index only
my_list[[2]][1,]
[1] 4 7
```

```

# Selecting first element of second component by name and index
my_list$`I am a Matrix`[1,1]
[1] 4
# Selecting first element of second component by index only
my_list[[2]][1,1]
[1] 4

```

## Data Frames

A data frame in R is a matrix-like object where each column contains values of one variable, with one big advantage: it can store different data types. While each column can only contain values of a single data type, we can have columns of different data types. Another requirement is that each column should contain the same number of items or observations.

In the previous section, we saw the built-in `mtcars` data frame. The name is an abbreviated version of Motor Trend Car Road Tests. According to the dataset description (see `?mtcars`), “the data was extracted from the 1974 *Motor Trend US magazine*, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles.”

Here is the dataset

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Each column represents one aspect of the cars and each row represents all the aspects of one car.

This dataset contains only 32 observations, so we can display it entirely on one single page. However, what if we have a very long dataset? Can we only display part of it to gain a feel of the data? The answer is yes. To do so, we can use the `head()` and `tail()` functions. Let's try this with another built-in dataset called `iris`:

```
# How many rows are there in the iris data set?  
nrow(iris)  
[1] 150  
# Displaying only the first six rows  
head(iris, n = 6)  
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
1       5.1      3.5        1.4       0.2   setosa  
2       4.9      3.0        1.4       0.2   setosa  
3       4.7      3.2        1.3       0.2   setosa  
4       4.6      3.1        1.5       0.2   setosa  
5       5.0      3.6        1.4       0.2   setosa  
6       5.4      3.9        1.7       0.4   setosa  
# Displaying only the first ten rows  
head(iris, n = 10)  
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
1       5.1      3.5        1.4       0.2   setosa  
2       4.9      3.0        1.4       0.2   setosa  
3       4.7      3.2        1.3       0.2   setosa  
4       4.6      3.1        1.5       0.2   setosa  
5       5.0      3.6        1.4       0.2   setosa  
6       5.4      3.9        1.7       0.4   setosa  
7       4.6      3.4        1.4       0.3   setosa  
8       5.0      3.4        1.5       0.2   setosa  
9       4.4      2.9        1.4       0.2   setosa  
10      4.9      3.1        1.5       0.1  setosa  
# Displaying only the last six rows. The argument n is set to 6 by default.  
tail(iris)  
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
145      6.7      3.3        5.7       2.5 virginica  
146      6.7      3.0        5.2       2.3 virginica  
147      6.3      2.5        5.0       1.9 virginica  
148      6.5      3.0        5.2       2.0 virginica  
149      6.2      3.4        5.4       2.3 virginica  
150      5.9      3.0        5.1       1.8 virginica
```

Another way to get rapid insight into a data frame is via the `str()` function. The following information is returned when calling the `str()` function on a data frame:

1. The total number of observations (or rows)
2. The total number of variables (or columns)
3. A complete list of the names of the variables
4. The data type of each variable
5. A glimpse of the first few observations

```
str(mtcars)  
'data.frame': 32 obs. of 11 variables:  
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...  
 $ cyl : num 6 6 4 6 8 6 8 4 4 6 ...  
 $ disp: num 160 160 108 258 360 ...
```

```

$ hp   : num  110 110 93 110 175 105 245 62 95 123 ...
$ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
$ wt   : num  2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num  16.5 17 18.6 19.4 17 ...
$ vs   : num  0 0 1 1 0 1 0 1 1 1 ...
$ am   : num  1 1 1 0 0 0 0 0 0 0 ...
$ gear: num  4 4 4 3 3 3 3 4 4 4 ...
$ carb: num  4 4 1 1 2 1 4 2 2 4 ...
str(iris)
'data.frame': 150 obs. of 5 variables:
$ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
$ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
$ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
$ Species     : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1
1 1 1 ...

```

We can create data frames through the `data.frame()` function. As arguments, we simply pass the vectors that will become the various columns of our data frame. While the vectors can be of different data types, they need to be of the same length.

Suppose there are 5 students in a class and they sit an exam where the pass mark is 75 out of 100. Let us create a data frame called `Class_DF` that contains the following variables:

- Student: The name of the student
- Marks: The number of marks (out of 100) for each student
- Rank: The rank of each student
- Passed: Whether each student has passed or not

Here is how we can create the data frame:

```

# Creating variables
Student <- c("Jane", "John", "Tim", "Michael", "Emma")
Marks <- c(80,95,98.5,78.4,50.25)
Rank <- as.integer(c(3,2,1,4,5))
Passed <- c(TRUE,TRUE,TRUE,TRUE,FALSE)
# Creating Data Frame
Class_DF <- data.frame(Student, Marks, Rank, Passed)
str(Class_DF)
'data.frame': 5 obs. of 4 variables:
$ Student: chr "Jane" "John" "Tim" "Michael" ...
$ Marks  : num 80 95 98.5 78.4 50.2
$ Rank   : int 3 2 1 4 5
$ Passed : logi TRUE TRUE TRUE TRUE FALSE
Class_DF
  Student Marks Rank Passed
1   Jane  80.00    3   TRUE
2   John  95.00    2   TRUE
3    Tim  98.50    1   TRUE
4 Michael  78.40    4   TRUE
5   Emma  50.25    5  FALSE

```

Observe that each column has a different data type. We would not have been able to store this data in a matrix, so, in this sense, data frames are super convenient. In fact, data frames are very popular. You are very likely to encounter them regularly when working with R. In a

later week, we will use the `dplyr` package to do all sorts of interesting operations on data frames.

Just like matrices, we can select elements of data frames in the usual way by using square brackets `[ ]`.

```
# Select first column
Class_DF[, 1]
[1] "Jane"      "John"      "Tim"       "Michael"   "Emma"
# Select first to third columns
Class_DF[, 1:3]
  Student Marks Rank
1    Jane  80.00    3
2   John  95.00    2
3   Tim  98.50    1
4 Michael 78.40    4
5  Emma  50.25    5
# Select second and fourth columns
Class_DF[, c(2, 4)]
  Marks Passed
1 80.00  TRUE
2 95.00  TRUE
3 98.50  TRUE
4 78.40  TRUE
5 50.25 FALSE
# Select first row
Class_DF[1,]
  Student Marks Rank Passed
1    Jane    80     3  TRUE
# Select first to third rows
Class_DF[1:3,]
  Student Marks Rank Passed
1    Jane  80.0    3  TRUE
2   John  95.0    2  TRUE
3   Tim  98.5    1  TRUE
# Select second and fourth rows
Class_DF[c(2, 4), ]
  Student Marks Rank Passed
2   John  95.0    2  TRUE
4 Michael 78.4    4  TRUE
```

In the above demo, we have used selection by index. We could have done the same thing via selection by name. Data frames are formally a special kind of list. So, we can also use the `$` notation to select specific columns:

```
Class_DF$Student
[1] "Jane"      "John"      "Tim"       "Michael"   "Emma"
Class_DF$Marks
[1] 80.00 95.00 98.50 78.40 50.25
Class_DF$Rank
[1] 3 2 1 4 5
Class_DF$Passed
[1] TRUE  TRUE  TRUE  TRUE FALSE
# Selecting name of second student
Class_DF$Student[2]
[1] "John"
```

Now, suppose we want to select rows based on some condition. Consider the following different scenarios. Suppose we only want to display observations where:

- The student has passed
- The marks are higher than 90
- The students are ranked among the top 3
- The student's name contains exactly 4 letters.

To subset a data frame based on some condition, we use the `subset()` function as follows:

```
# Scenario 1: Display only students who have passed
subset(Class_DF, Passed == TRUE)
  Student Marks Rank Passed
1   Jane  80.0    3    TRUE
2   John  95.0    2    TRUE
3   Tim  98.5    1    TRUE
4 Michael 78.4    4    TRUE
# Scenario 2: Display only students who scored higher than 90 marks
subset(Class_DF, Marks > 90)
  Student Marks Rank Passed
2   John  95.0    2    TRUE
3   Tim  98.5    1    TRUE
# Scenario 3: Display only students ranked among the top 3
subset(Class_DF, Rank <= 3)
  Student Marks Rank Passed
1   Jane  80.0    3    TRUE
2   John  95.0    2    TRUE
3   Tim  98.5    1    TRUE
# Scenario 4: Display only students whose name contain exactly 4 letters
subset(Class_DF, nchar(as.character(Student)) == 4)
  Student Marks Rank Passed
1   Jane  80.00    3    TRUE
2   John  95.00    2    TRUE
5  Emma  50.25    5   FALSE
```

## Coerce and Test

As with types we can use `as.*()` and `is.*()` functions to coerce and test data structures in R. Try to understand the result of the commands below.

```
is.data.frame(mtcars)
[1] TRUE
is.matrix(mtcars)
[1] FALSE
is.list(mtcars)
[1] TRUE
as.matrix(iris[1:5, ])
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 "5.1"        "3.5"       "1.4"       "0.2"      "setosa"
2 "4.9"        "3.0"       "1.4"       "0.2"      "setosa"
3 "4.7"        "3.2"       "1.3"       "0.2"      "setosa"
4 "4.6"        "3.1"       "1.5"       "0.2"      "setosa"
5 "5.0"        "3.6"       "1.4"       "0.2"      "setosa"
as.list(matrix(1:10, nrow = 2))
[[1]]
[1] 1
```

```
[[2]]  
[1] 2  
  
[[3]]  
[1] 3  
  
[[4]]  
[1] 4  
  
[[5]]  
[1] 5  
  
[[6]]  
[1] 6  
  
[[7]]  
[1] 7  
  
[[8]]  
[1] 8  
  
[[9]]  
[1] 9  
  
[[10]]  
[1] 10  
is.list(letters[1:4])  
[1] FALSE
```

## Useful Links and Resources

- “[R objects](#)” section from *Hands-on programming with R*

# Data Structures in Python



## Working With Python

We begin with some basics on Python before we talk specifically about data structures.

### Dependencies

An essential part of Python is to incorporate dependencies, also known as packages. These are modules of existing code and functions which you can base your code upon. They may actually include code written by you. A wide range of dependencies is available with the Anaconda installation so you will probably only need to activate them. But if you need to install a package with Anaconda you need to use `conda` commands in the QtConsole (macOS) or the Anaconda prompt (Windows). For example, `conda install Quandl`. Sometimes a package is not available as a `conda` package but can be installed by `pip install`, for example `pip install Quandl`.

To find out more about `conda` please consult [this guide](#). That guide also provides a handy cheat-sheet if you want to learn, for example, about updating packages.

To use a dependency in a Python package, the basic code is

```
import _package_ as _alias_
```

For example, the below code is using the alias `np` for the Python package `numpy` and printing the number `\(pi\)`:

```
import numpy as np
print(np.pi)
3.141592653589793
```

Note that the alias `np` above (or any other you want to use) need to be placed before `pi`, i.e. `np.pi`. In fact, using an alias is optional, yet recommended practice. For more on the `numpy` package see the dedicated set of notes that follows.

## Data Types and Structures

Python has a number of built in data types; some of them are given below:

1. `float`
2. `int`
3. Strings
4. Boolean

5. Tuples
6. Lists
7. Sets
8. Dictionaries

## **float, int, String and Boolean Types**

`float` and `int` are for real numbers and integers respectively. The `type()` method can identify the kind of a data type, see the code below for an example:

```
a=4.6
print(type(a))
<class 'float'>
b=10
print(type(b))
<class 'int'>
```

Strings are for text and have type `str`. The code below demonstrates Python's character processing capabilities along with their output:

```
phrase='All models are wrong, but some are useful.'
print(phrase[0:3]) #print the first three characters
All
print(phrase.find('models')) #Find the index where the word 'models' starts.
4
print(phrase.find('right')) # when the word does not exist
-1
print(phrase.lower()) #set all to lower case
all models are wrong, but some are useful.
print(phrase.upper()) #set all to upper case
ALL MODELS ARE WRONG, BUT SOME ARE USEFUL.
a=phrase.split(',')
print(a)
['All models are wrong', ' but some are useful.']}
```

Finally Boolean type is `bool` and is typically used for the results of true/false statements. For example,

```
k=1>3
print(k)
False
print(type(k))
<class 'bool'>
```

# Data Structures

Data structures or collections in Python are containers that are used to store collections of data of potentially different types. There are four collection data types in the Python programming language:

- Tuples are collections which is ordered and unchangeable. A tuple can have duplicate members.
- Lists are collections which is ordered and changeable. A list can have duplicate members.
- Sets are collections which is unordered and unindexed. A set cannot have duplicate members.
- Dictionaries are collections which is unordered, changeable and indexed. A dictionary cannot have duplicate members.

## Tuples and Indexing

A tuple is a collection of data types which is ordered and unchangeable. In Python tuples are written with round brackets.

```
tuple1 = ("apple", "banana", "cherry")
print(tuple1)
('apple', 'banana', 'cherry')
tuple2=(1.0,3.0,7.0)
print(tuple2)
(1.0, 3.0, 7.0)
```

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

You can access tuple items by referring to the index number, inside square brackets.

*Python indexing:* Note that in Python the first item of a tuple (or any other collection) has index 0, not 1! Similarly the second item has index 1, the third item has index 2 and so on. This is different to R where the first item is indexed with 1, the second with 2 and so on.

```
tuple1 = ("apple", "banana", "cherry")
print(tuple1[0])
apple
print(tuple1[2])
cherry
```

Negative indexing is also possible and it means starting counting from the end; -1 refers to the last item, -2 refers to the second last item etc.

```
tuple1 = ("apple", "banana", "cherry")
print(tuple1[-1])
cherry
```

You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

For example, if we want to return the third, fourth, and the fifth item, we can use the following code:

```
tuple3 = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(tuple3[2:5])
('cherry', 'orange', 'kiwi')
```

## Lists, Sets and Dictionaries

Here we see three other types of Python collections. One of the ways in which they all differ from tuples is that, unlike tuples, they are mutable. In other words you can change the values of a list, set or a dictionary but you cannot do that with tuples.

A list is a collection which is ordered and changeable. In Python lists are defined with square brackets.

```
list1 = ["apple", "banana", "cherry"]
print(list1)
['apple', 'banana', 'cherry']
```

For example if we want to print the second item of the list, we can use the following code:

```
print(list1[1])
banana
```

To illustrate that a list is mutable let us check the following example:

```
list1 = ["apple", "banana", "cherry"]
print(list1)
['apple', 'banana', 'cherry']
list1[1] = "apple"
print(list1)
['apple', 'apple', 'cherry']
```

A set is a collection which is unordered and unindexed. In Python, sets are defined with curly brackets.

```
set = {"apple", "banana", "cherry"}
print(set)
{'banana', 'cherry', 'apple'}
```

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are defined with curly brackets, and they have keys and values.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964}
```

```
}

print(thisdict)

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Items of a dictionary can be accessed by referring to their key names, inside square brackets:

```
thisdict["model"]

'Mustang'
```

You can change the value of a specific item by referring to its key name:

For example if we want to change the “year” to 2018, we can use the following code:

```
thisdict = {

    "brand": "Ford",
    "model": "Mustang",
    "year": 1964

}

thisdict["year"] = 2018
```

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict = {

    "brand": "Ford",
    "model": "Mustang",
    "year": 1964

}

thisdict["color"] = "red"

print(thisdict)

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

The `pop()` method removes the item with the specified key name:

```
thisdict = {

    "brand": "Ford",
    "model": "Mustang",
    "year": 1964

}

thisdict.pop("model")

'Mustang'

print(thisdict)

{'brand': 'Ford', 'year': 1964}
```

# Useful Links and Resources

- [Data structures documentation from python.org](#)

# Working With NumPy and Pandas

\*\* Note: The code chunks below should be run in the following order \*\*

## NumPy

**NumPy** is the key package for numerical computing in Python. It allows you to perform operations on whole blocks of data without writing loops. The fast numerical processing it permits makes it the basis of most visualisation packages as well more sophisticated scientific and machine learning packages.

To activate it in Spyder scripts or Jupyter notebooks, simply add the following line at the beginning of your script or notebook.

```
import numpy as np
```

Note that the code blocks that follow in this page assume that **NumPy** has been activated. So make sure you execute the above line beforehand.

## NumPy Arrays

A **NumPy array**, or else `ndarray`, is a grid of values of the same type, which is indexed by a tuple of non-negative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can create a `ndarray` object by using the `array()` function. Also the function `type()` gives us the type of the array.

```
array1 = np.array([1, 2, 3, 4, 5])
print(array1)
[1 2 3 4 5]
print(type(array1))
<class 'numpy.ndarray'>
```

To create an `ndarray`, we can also pass a list, tuple or any array-like object into the `array()` function, and it will be converted into an `ndarray`:

```
#NumPy arrays from tuples
arr = np.array((1, 2, 3, 4, 5))
print(arr)
[1 2 3 4 5]
```

Arrays may have dimensions of 0 (single numbers, strings or Booleans), 1 (vectors), 2 (matrices) or even 3 (tensors).

```
array0d = np.array(42) #0-dimensional array
print(array0d)
```

```

42
array1d = np.array([1, 2, 3, 4, 5]) #1-dimensional array
print(array1d)
[1 2 3 4 5]
array2d = np.array([[1, 2, 3], [4, 5, 6]]) #2-dimensional array
print(array2d)
[[1 2 3]
 [4 5 6]]

```

**NumPy** also provides many methods to create some commonly used arrays:

```

a = np.zeros((2,2))    # Create an array of all zeros
print(a)                # Prints "[[ 0.  0.]
                        #           [ 0.  0.]]"
[[0. 0.]
 [0. 0.]]
b = np.ones((1,2))     # Create an array of all ones
print(b)                # Prints "[[ 1.  1.]]"
[[1. 1.]]
c = np.full((2,2), 7)   # Create a constant array
print(c)                # Prints "[[ 7.  7.]
                        #           [ 7.  7.]]"
[[7 7]
 [7 7]]
d = np.eye(2)          # Create a 2x2 identity matrix
print(d)                # Prints "[[ 1.  0.]
                        #           [ 0.  1.]]"
[[1. 0.]
 [0. 1.]]
np.random.seed(10)
e = np.random.random((2,2)) # Create an array filled with random values
print(np.round(e,3))       # print the array with values rounded to 3 d.p.
[[0.771 0.021]
 [0.634 0.749]]

```

In the example below `np.sqrt` is a *vectorised* calculation on a **NumPy** array set up using the function `arange`. In other words the function `np.sqrt()` is applied to all the elements of the array `rng` at the same time rather than each of them separately.

```

rng=np.arange(10)
print(rng)
[0 1 2 3 4 5 6 7 8 9]

```

```
print(type(rng))
<class 'numpy.ndarray'>
print(np.sqrt(rng))
[0.           1.           1.41421356 1.73205081 2.           2.23606798
 2.44948974 2.64575131 2.82842712 3.           ]
```

## NumPy Operations

NumPy provides some handy methods to summarise the data. For example

```
a=np.array([[1.0,2.0,4.0],[-1.0,2.0,-5.0]])
print(a.shape)
(2, 3)
print(a)
[[ 1.  2.  4.]
 [-1.  2. -5.]]
print(a.sum(axis=0))
[ 0.  4. -1.]
print(a.sum(axis=1))
[ 7. -4.]
print(a.sum())
3.0
```

Note the Python indexing feature we saw on Python collections: `axis=0` refers to the first dimension of the 2-D array above; namely the rows. Similarly `axis=1` refers to its second dimension; namely the columns.

NumPy can also be used for matrix operations such as transposing and multiplying matrices:

```
a=np.array([[1.0,2.0,4.0],[-1.0,2.0,-5.0]])
b=np.transpose(a)    # b is the transpose of a
print(b.shape)
(3, 2)
print(b)
[[ 1. -1.]
 [ 2.  2.]
 [ 4. -5.]]
c=np.dot(a,b)        #c = a * b
print(c)
[[ 21. -17.]
 [-17.  30.]]
c=a @ b      #c = a * b as before
print(c)
```

```
[[ 21. -17.]
 [-17.  30.]]
```

**NumPy** arrays can be easily subsetted.

```
a=np.array([[1.0,2.0,4.0],[-1.0,2.0,-5.0]])

print(a)
[[ 1.  2.  4.]
 [-1.  2. -5.]]
print(a[0:2,1:3])
[[ 2.  4.]
 [ 2. -5.]]
```

You can also set the values in a **NumPy** array using simple Boolean expressions:

```
np.random.seed(0)

arr=np.random.random((3,3))

print(np.round(arr,3))
[[0.549 0.715 0.603]
 [0.545 0.424 0.646]
 [0.438 0.892 0.964]]

arr[arr<0.5]=0

print(arr)
[[0.5488135  0.71518937  0.60276338]
 [0.54488318  0.          0.64589411]
 [0.          0.891773   0.96366276]]

print(np.sum((arr<0) & (arr>1))) # The operator '&' stands for 'and'
0

print(np.sum((arr<0) | (arr>0.7))) # The operator '|' stands for 'or'
3
```

## Pandas

The **pandas** package provides a major tool in the Python ecosystem. While it is possible to work with data with other Python tools such as the **NumPy** package (see the notes on Data structures with Python), the key difference is that **pandas** is designed for working with *heterogeneous* data. Moreover it gives Python a functionality similar to R Data frames and integrates powerful low level modelling tools such as data import, aggregation and cleaning.

**Pandas** is included in the Anaconda installation. To activate it in Spyder scripts or Jupyter notebooks, simply add the following lines in the beginning of your program.

```
import pandas as pd
```

**Note:** The code blocks that follow in this document assume that `Pandas` has been activated. So make sure you execute the above line beforehand. Some of the blocks also require `NumPy` so make sure this is activated as well.

## Pandas Series

A `Series` is a one-dimensional labelled array capable of holding any data type (integers, strings, floating point numbers, etc.). The axis labels are collectively referred to as the index. The basic method to create a `Series` is to call:

```
s = pd.Series(data, index=index)
```

The object `data` can be several things, for example a Python dictionary or a `ndarray`. For `ndarrays`, the index must be the same length as data. If no index is passed, one will be created having values [0, ..., len(data) - 1].

```
np.random.seed(1)
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
s
a    1.624345
b   -0.611756
c   -0.528172
d   -1.072969
e    0.865408
dtype: float64
```

## Pandas Data Frames

A **Pandas** data frame is a 2-dimensional labelled data structure with columns of potentially different types. You can think of it like a spreadsheet or R data frame or a dictionary of `Series` objects. It is generally the most commonly used `pandas` object. Like `Series`, data frames accept many different kinds of input, such as dictionaries, lists, sets, `Series`, `ndarrays` etc.

Along with the data, we can optionally pass index (row labels) and columns (column labels) arguments. If axis labels are not passed, they will be constructed from the input data based on some default options.

We can create a `pandas` data frame from a structured array in the following way

```
data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])
pd.DataFrame(data, index=['first', 'second'])
```

that gives the output

	A	B	C
<b>first</b>	0	0.0	b"
<b>second</b>	0	0.0	b"

pandas data frame

Below is how we can create a `pandas` data frame from dictionary of `ndarrays` or lists

```
d = {'one': [1., 2., 3., 4.],  
     'two': [4., 3., 2., 1.]}  
  
pd.DataFrame(d)  
  
# repeat with pd.DataFrame(d, index=['a', 'b', 'c', 'd']) instead of pd.DataFrame(d)  
)
```

that gives the output

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

## Another pandas Data Frame

**Pandas** provides a nice tool to get basic summaries from your data set. The following code creates the `pandas` data frame `df` and applies the `describe()` function on it.

```
d = {'one': [1., 2., 3., 4.],  
     'two': [4., 3., 2., 1.]}  
  
df=pd.DataFrame(d)  
  
df.describe()
```

that gives the output

	one	two
count	4.000000	4.000000
mean	2.500000	2.500000
std	1.290994	1.290994
min	1.000000	1.000000
25%	1.750000	1.750000
50%	2.500000	2.500000
75%	3.250000	3.250000
max	4.000000	4.000000

## Descriptive statistics from Pandas.

Also, in a very similar manner as with **NumPy**, we can use Boolean indexing to generate a subset of your data:

```

d = {'one': [1., 2., 3., 4.],
      'two': [4., 3., 2., 1.]}

df=pd.DataFrame(d)

print(df[df>0])

   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

```

Finally, indexing also works in a similar manner with **NumPy**. The additional feature here is that we can use the rows/column names in addition to the numbered indices.

```

d = {'one': [1., 2., 3., 4.],
      'two': [4., 3., 2., 1.]}

df=pd.DataFrame(d)

print(df['one'])

0    1.0
1    2.0
2    3.0
3    4.0

Name: one, dtype: float64

print(df['two'])

0    4.0
1    3.0
2    2.0
3    1.0

Name: two, dtype: float64

```

## Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapters 4 and 5.
- [NumPy website](#)
- [10 minutes to pandas from the pandas website](#)

# Data Exchange File Formats With Python

\*\* Note: The code chunks below should be run in the following order \*\*

## File Formats

We have already described several file formats that provide standard ways to encode information to be stored in a file. They can usually be identified by looking at the file extension; e.g. a file saved with name `Data` in CSV format will appear as `Data.csv`. By noticing `.csv` extension we can clearly identify that it is a comma-separated file and data is stored in a specific way, in this case a tabular format.

An essential skill for a data scientist is to be able to understand the underlying structure of various file formats, and their advantages and disadvantages. There may also be situations in which decisions need to be taken on how to store data in order to improved the performance.

Next we will look in various file formats and provide information on how to import their information in Python.

### Plain Text (txt)

As the name suggests, plain text file format contain only plain text. Usually, this text is in unstructured form and there is no metadata associated with it.

Let's take a simple example of a text file. The following example shows text file data that contain text:

```
All models are wrong, but some are useful
```

The above text is written in the attached file called `Sentence.txt` and can be read in various ways one of them being with the code below:

```
text_file = open('Sentence.txt', 'r')
lines = text_file.read()
lines
'All models are wrong, but some are useful'
```

### Comma-Separated Values (CSV)

As mentioned in previous notes the comma-separated values (CSV) file format is for cases when data is stored in cells that are organized in rows and columns. Each column can be of different types; strings (text characters), dates, integers, etc. Each line in CSV file represents an observation or commonly called a record. Each record may contain one or more fields which are separated by a comma.

The image below shows the `titanic.csv` file (attached to this notes) when opened withTextEdit on a macOS. This file contains information for each of the passengers of the Titanic.

survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0,3	male,22,1,0,7.25,S,Third,man,TRUE,,Southampton,no,FA	lse,												
1,1	female,38,1,0,71.2833,C,First,woman,FA	LSE,C,Cherbourg,yes,FA	lse,											
1,3	female,26,0,0,7.925,S,Third,woman,FA	LSE,,Southampton,yes,TRU	E,											
1,1	female,35,1,0,53.1,S,First,woman,FA	LSE,C,Southampton,yes,FA	LSE,											
0,3	male,35,0,0,8.05,S,Third,man,TRUE,,Southampton,no,TRU	E,												
0,3	male,,0,0,8.4583,0,Third,man,TRUE,,Queenstown,no,TRU	E,												
0,1	male,54,0,0,51.8625,S,First,man,TRUE,E,Southampton,no,TRU	E,												
0,3	male,2,3,1,21.075,S,Third,child,FA	LSE,,Southampton,no,FA	LSE,											
1,3	female,27,0,2,11.1333,S,Third,woman,FA	LSE,,Southampton,yes,FA	LSE,											
1,2	female,14,1,0,30.0708,C,Second,child,FA	LSE,,Cherbourg,yes,FA	LSE,											
1,3	female,4,1,1,16.7,S,Third,child,FA	LSE,G,Southampton,yes,FA	LSE,											
1,1	female,58,0,0,26.55,S,First,woman,FA	LSE,C,Southampton,yes,TRU	E,											
0,3	male,20,0,0,8.05,S,Third,man,TRUE,,Southampton,no,TRU	E,												
0,3	male,39,1,5,31.275,S,Third,man,TRUE,,Southampton,no,FA	LSE,												
0,3	female,14,0,0,7.8542,S,Third,child,FA	LSE,,Southampton,no,TRU	E,											
1,2	female,55,0,0,16,S,Second,woman,FA	LSE,,Southampton,yes,TRU	E,											
0,3	male,2,4,1,29.125,Q,Third,child,FA	LSE,,Queenstown,no,FA	LSE,											
1,2	male,,0,0,13,S,Second,man,TRUE,,Southampton,yes,TRU	E,												
0,3	female,31,1,0,18,S,Third,woman,FA	LSE,,Southampton,no,FA	LSE,											
1,3	female,,0,0,7.225,C,Third,woman,FA	LSE,,Cherbourg,yes,TRU	E,											
0,2	male,35,0,0,26,S,Second,man,TRUE,,Southampton,no,TRU	E,												
1,2	male,34,0,0,13,S,Second,man,TRUE,D,Southampton,yes,TRU	E,												
1,3	female,15,0,0,8.0292,Q,Third,child,FA	LSE,,Queenstown,yes,TRU	E,											
1,1	male,28,0,0,35.5,S,First,man,TRUE,A,Southampton,yes,TRU	E,												
0,3	female,8,3,1,21.075,S,Third,child,FA	LSE,,Southampton,no,FA	LSE,											
1,3	female,38,1,5,31.3875,S,Third,woman,FA	LSE,,Southampton,yes,FA	LSE,											
0,3	male,,0,0,7.225,C,Third,man,TRUE,,Cherbourg,no,TRU	E,												
0,1	male,19,3,2,263,S,First,man,TRUE,C,Southampton,no,FA	LSE,												
1,3	female,,0,0,7.8792,Q,Third,woman,FA	LSE,,Queenstown,yes,TRU	E,											

A view of a CSV file.

To load data in Python from a CSV file, you can use the **Pandas** package. The code below activates the `pandas` package (first line), reads the data from the attached file `titanic.csv`, stores them in the `pandas` data frame `pd` (second line), which we then view by simply typing its name (third line).

```
import pandas as pd
df = pd.read_csv('titanic.csv')
df
```

The output from the program above is shown below:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True
...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.0	0	0	13.0000	S	Second	man	True
887	1	1	female	19.0	0	0	30.0000	S	First	woman	False
888	0	3	female	NaN	1	2	23.4500	S	Third	woman	False
889	1	1	male	26.0	0	0	30.0000	C	First	man	True
890	0	3	male	32.0	0	0	7.7500	Q	Third	man	True

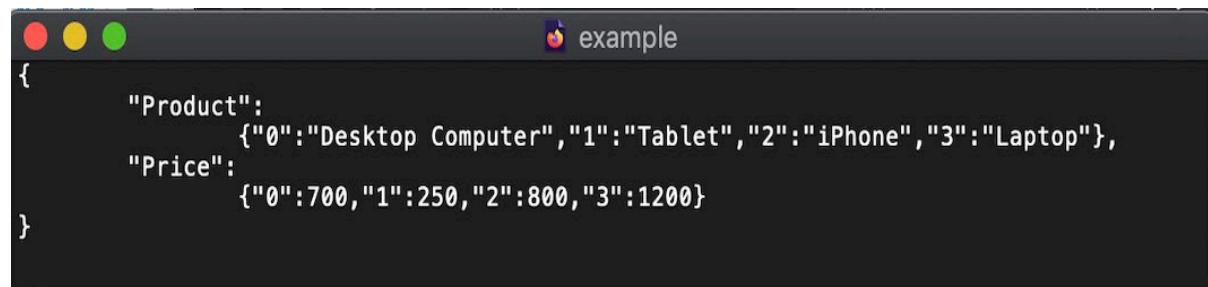
A view of the previous CSV file as a pandas data frame.

To export `pandas` data frames to csv files the following line can be used `df.to_csv(r'Path to store the exported CSV file\File Name.csv')` e.g.

```
df.to_csv (r'Path to store the exported CSV file\export_dataframe.csv', index = False, header=True)
```

## JavaScript Object Notation (JSON)

Recall that JavaScript Object Notation (JSON) is a standard format for sending structured data over the web. The JSON file format can be easily read in any programming language because it is language-independent data format. It is a much more flexible data format than a tabular text form like CSV. Here is an example of a JSON file:



```
example
{
  "Product": {
    "0": "Desktop Computer", "1": "Tablet", "2": "iPhone", "3": "Laptop"
  },
  "Price": {
    "0": 700, "1": 250, "2": 800, "3": 1200
  }
}
```

A view of a JSON file.

Here is how it can be imported into Python using `pandas`.

```
import pandas as pd
```

```
df = pd.read_json('example.json')
df
```

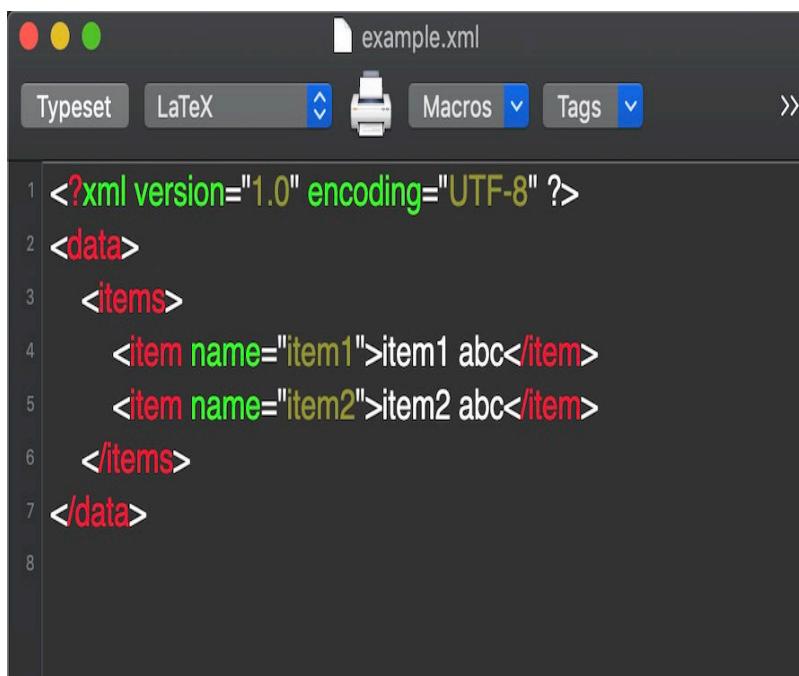
	Product	Price
0	Desktop Computer	700
1	Tablet	250
2	iPhone	800
3	Laptop	1200

A view of the previous json file as a pandas data frame.

To export `pandas` Data Frames to JSON files the following line can be used `df.to_json(r'Path to store the exported JSON file\File Name.json')`

## Extensible Markup Language (XML)

As mentioned in Week 3, Extensible Markup Language (XML) file format is a markup language that has certain rules for encoding data. The following example shows an xml document which is attached as `example.xml`:



```
<?xml version="1.0" encoding="UTF-8" ?>
<data>
  <items>
    <item name="item1">item1 abc</item>
    <item name="item2">item2 abc</item>
  </items>
</data>
```

A view of a XML file.

To read this file into Python, the `ElementTree` package can be used for example with the code below:

```
import xml.etree.ElementTree as et
tree = et.parse('example.xml')
root = tree.getroot()
```

XML files contain a single root that contains all the data. In the above code the `ElementTree` package is activated (1st line), the file is read and parsed into the object `tree` (2nd line) and the the root of the xml file is extracted (3rd line).

The code below illustrates how to read and access the data from the XML file.

```
print('Item #1 attribute: ',root[0][0].attrib) # one specific item attribute
Item #1 attribute: {'name': 'item1'}
print('\Item #2 data: ', root[0][1].text)# one specific item's data
\Item #2 data: item2 abc
```

## Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapter 6.
- [Article from Ankit Gupta on most commonly used file formats using Python.](#)
- [Documentation of the `ElementTree` package:](#)

# Block 03

## Introduction to Relational Database Management Systems

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:

<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-3>

### Database Management Systems

A *database* is a collection of data, typically stored electronically in a computer system.

A *database management system (DBMS)* is software package serving as an interface between the database and its end users or programs. DBMS allows users to

- Store
- Retrieve / query
- Update
- Manage
- Control access to

data in a database.

Databases are needed because they allow

- Storage of massive amounts of data
- Data access to multiple users
- Simultaneous change of data (concurrency)
- Efficient manipulation of large amounts of data
- Storage and access to the data in a safe, convenient, and reliable way
  - Safe: Data are consistent even if something bad happens (e.g. hardware or software failure)
  - Convenient: It is easy to manipulate or query a large amount of data (e.g. through a high level query language)
  - Reliable: Chance for the database to be down is very low.

# Relational Databases

A *relational database* is a type of database, where

- All data is represented in terms of tuples and stored in tables with columns and rows
- The structure on how the data is stored is pre-defined and imposed by the programmer
- Structured data can be held.

A *relational database management system (RDBMS)* is a DBMS designed specifically for relational databases. *Structured Query Language (SQL)* is one of the most widely used database languages designed for managing data held in a RDBMS. We will formally introduce SQL later this week.

# Nonrelational Databases

An alternative to relational database model is the nonrelational database model. Nonrelational databases, also known as NoSQL, store data in a different format to relational tables. Their key feature is that they allow unstructured and semi-structured data to be stored and manipulated. There are many different kinds of NoSQL databases, including:

- **Graph database:** A graph database stores data in terms of entities (or nodes) and the relationships (or edges) between entities. It is suitable for data with relations well represented as a graph
- **Document-oriented database:** A document-oriented database stores semi-structured data (or document-oriented information) in the form of JSON-like documents.

Document databases store all information for a given object in a single instance in the database, and every stored object can be different from every other

- Below shows a JSON-like document storing information about two books.  
Note that data is stored in nested key-value pairs (e.g. `year` is the key and `1995` is the value for the first book) and the information available for each book is different (e.g. `edition` and `description` are only available for the first book).

- [
- {
- `"year" : 1995,`
- `"title" : "An introduction to database systems",`
- `"info" : {`
- `"authors" : "C. J. Date",`
- `"edition" : 6,`
- `"subject" : "database management",`
- `"description" : "A comprehensive treatment of database technology. Features of this edition include: a proposal for rapprochement between object-oriented and relational technologies; expanded treatment of distributed databases; and chapters on functional dependencies, views, domains and missing information.",`
- `"publisher" : "Reading, Mass. : Addison-Wesley Pub. Co."`
- `}`
- `},`
- `{`
- `"year" : 2016,`

- "title" : "Introduction to computation and programming using Python: with application to understanding data",
  - "info" : {
  - "authors" : "Guttag, John V",
  - "subject" : ["Computer programming", "Python (Computer program language)"],
  - "publisher" : "The MIT Press"
  - }
  - }
- ]

# RDBMS Terminology

## Relation/Table

In a relational database, a relation is a table, with its rows representing a set of records (called tuples). . For example, a university database may have a relation (table) called `Student` to store the information for all students, a table `Course` for the course information and a table `Grade` for the grade of students in different courses:

`Student`:

<b>student_id</b>	<b>name</b>	<b>year</b>
201921323	Ava Smith	2
201832220	Ben Johnson	3
202003219	Charlie Jones	1

`Course`:

<b>course_id</b>	<b>name</b>	<b>capacity</b>
ST101	Programming for Data Science	60
ST115	Managing and Visualising Data	60
ST207	Databases	30

`Grade`:

<b>course_id</b>	<b>student_id</b>	<b>final_mark</b>
ST101	202003219	47
ST115	201921323	92
ST115	202003219	67
ST207	201933222	73

## Attribute

In relational databases, the *attribute* is a column in the table (or relation). Each attribute has a type (or domain). For example, the `student` table above has the attributes

- `student_id` (for student ID)
- `name`
- `year`

with the corresponding types being string, string and integer.

## Tuple

A *tuple* is a set of attribute values. In a relational database, a tuple is a row in a table. For example, each row in the *Student* table is a tuple storing the information for one student.

## Schema

The *schema* is the description on how the database and the database tables are constructed. For example, the *Student* table can have the schema:

```
Students(student_id: string, name: string, year: integer)
```

## Key

A primary key is the attribute used to uniquely identify a tuple, or the set of attributes whose combined values are unique. For the examples above, the primary keys are:

- Student: `student_id`
- Course: `course_id`
- Grade: `student_id` and `course_id`

A foreign key is an attribute or a set of attributes in a relational database table that provides a link between data in two tables. For example, `student_id` is the foreign key to link between the tables *Grade* and *Student*.

## Useful Links and Resources

- [What is NoSQL?](#) from MongoDB to learn more about NoSQL databases
- [What is a database?](#) from Oracle
- From Wikipedia
  - [Databases](#)
  - [Relational database](#)
  - [Document-oriented database](#)
  - [SQL](#)
  - [Foreign key](#)

# Introduction to SQL and SQLite Structured Query Language

Structured query language (SQL) is one of the most widely used database languages designed for managing data held in a relational database management system (RDBMS). It is used to:

- Create (databases/tables)
- Manipulate (insert, update and delete tables/tuples)
- Query

a relational database.

## SQLite

SQLite is the most widely used database engine. The other two popular database engines are MySQL and PostgreSQL. SQLite:

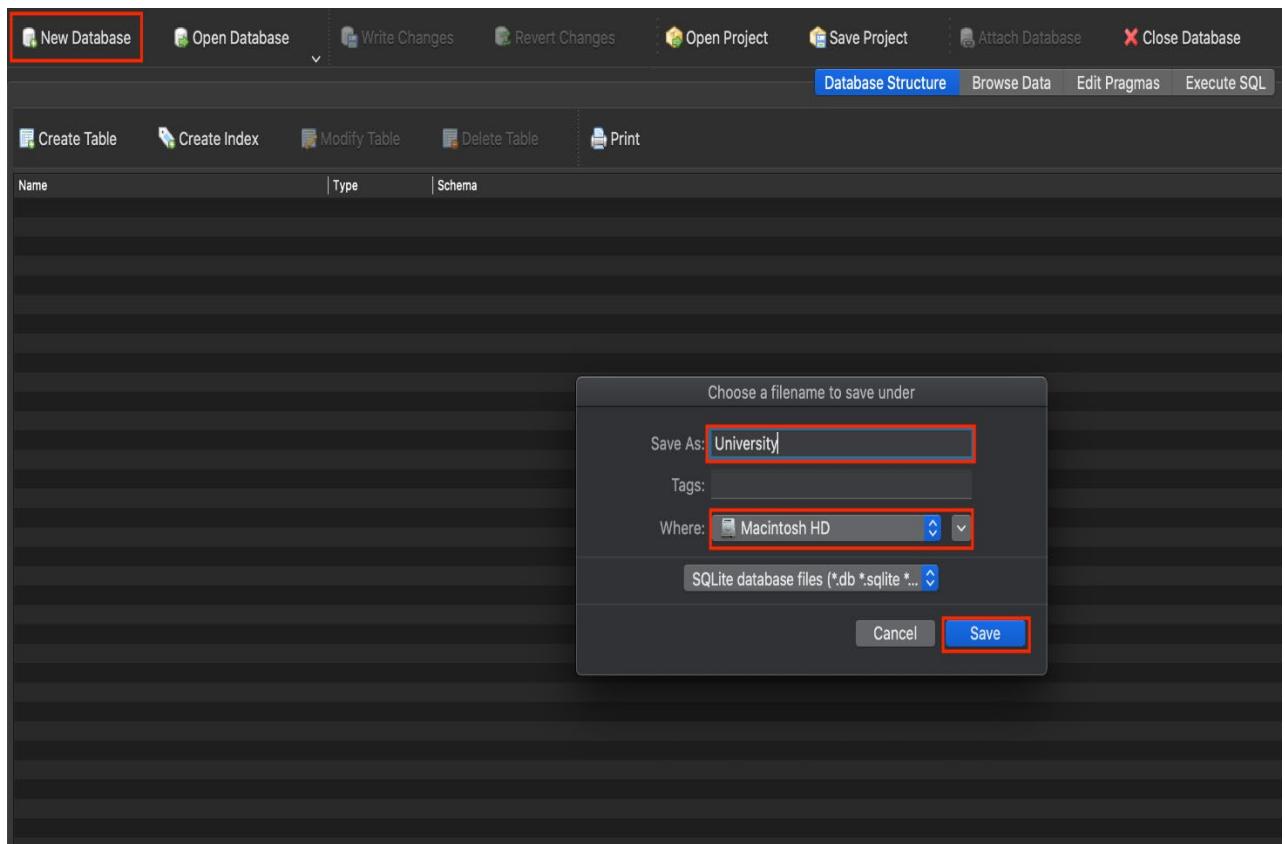
- Is *lite*
- Does not need configuration and uses little space
- Does not require a separate server process like other RDBMS systems do—SQLite reads and writes to a single file on your disk
- Is *self-contained* in the sense that it has very few dependencies
- Offers full-featured SQL.

## Creating Databases

In this course we use **DB Browser for SQLite (DB4S)** to create, query and edit database files compatible with SQLite. You can download it at the [DB Browser for SQLite](#) web pages.

In this document we consider a database `University`. We want to use it to store information about students, courses information and student grades.

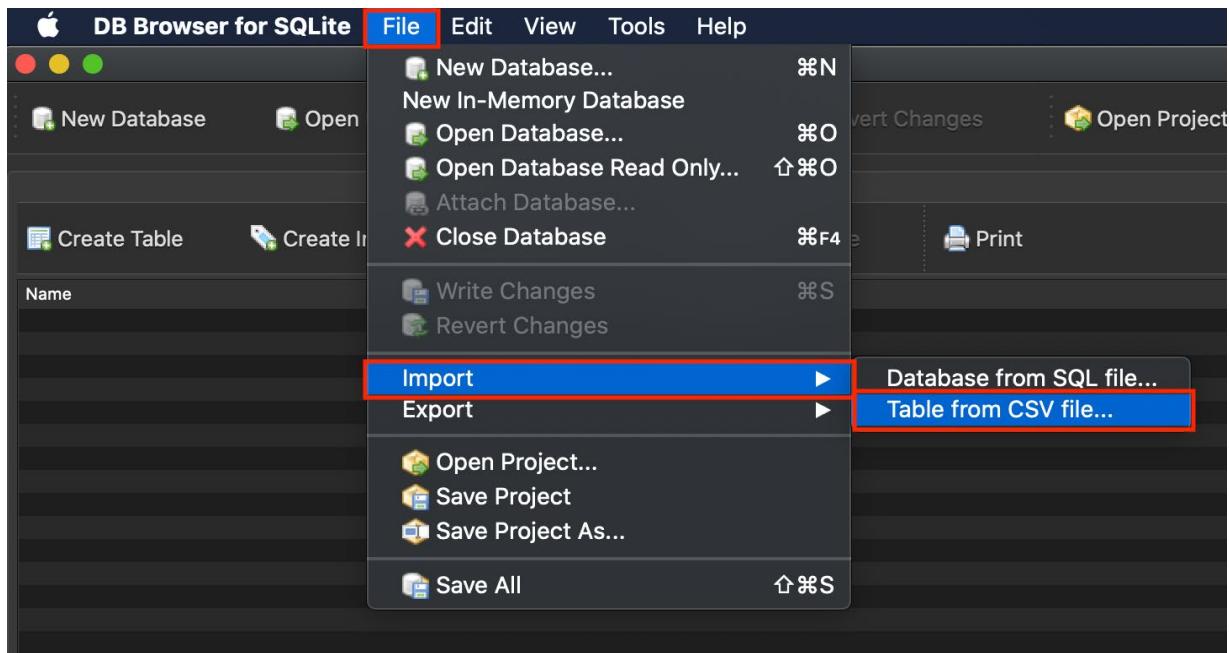
Click on `New Database` and use `University` as the name of the database. You can safely close the “Edit table definition” pop-up window by hitting `Cancel`.



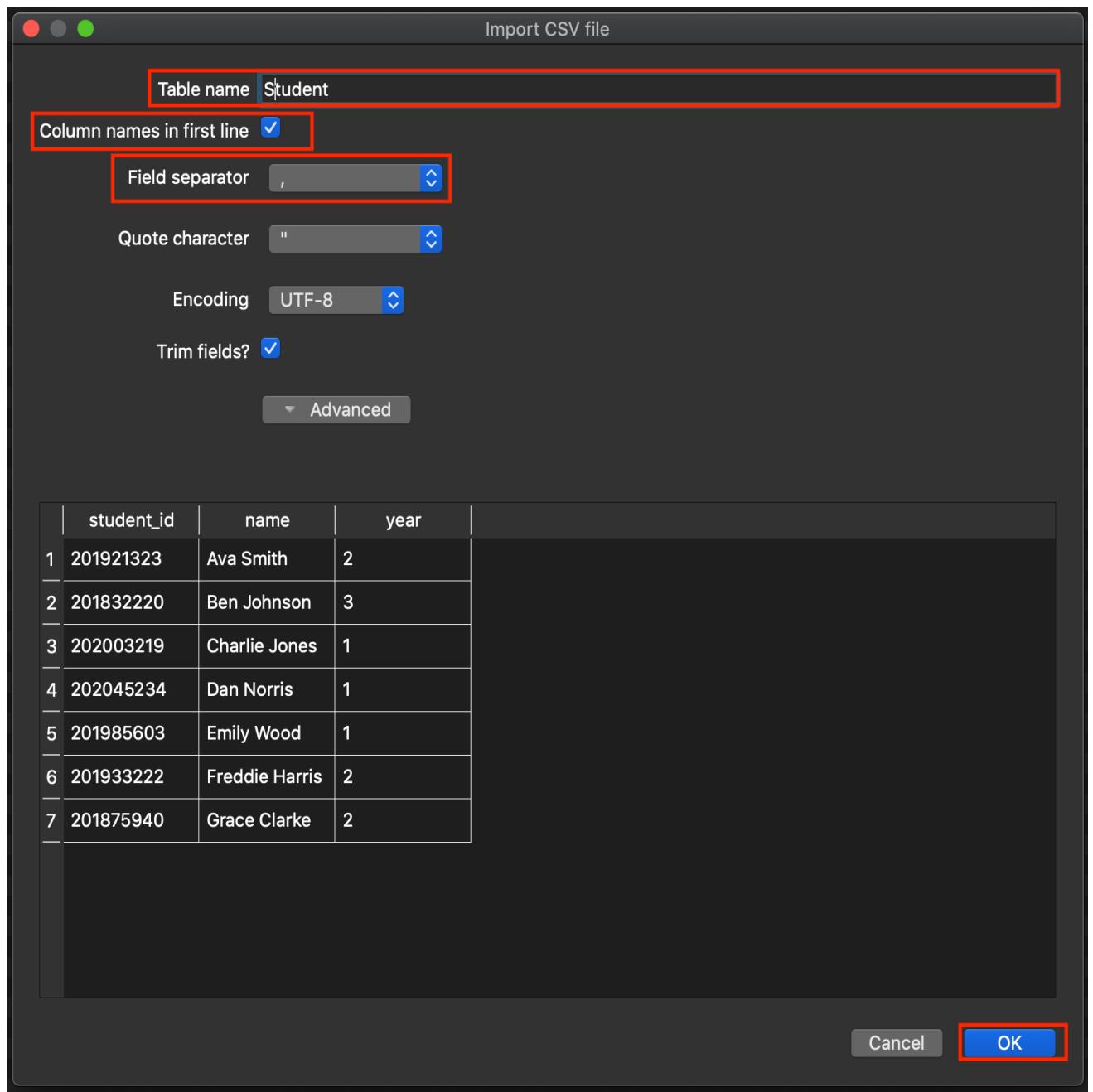
Screenshot on how to create a database

## Add Tables to the Database From CSV Files

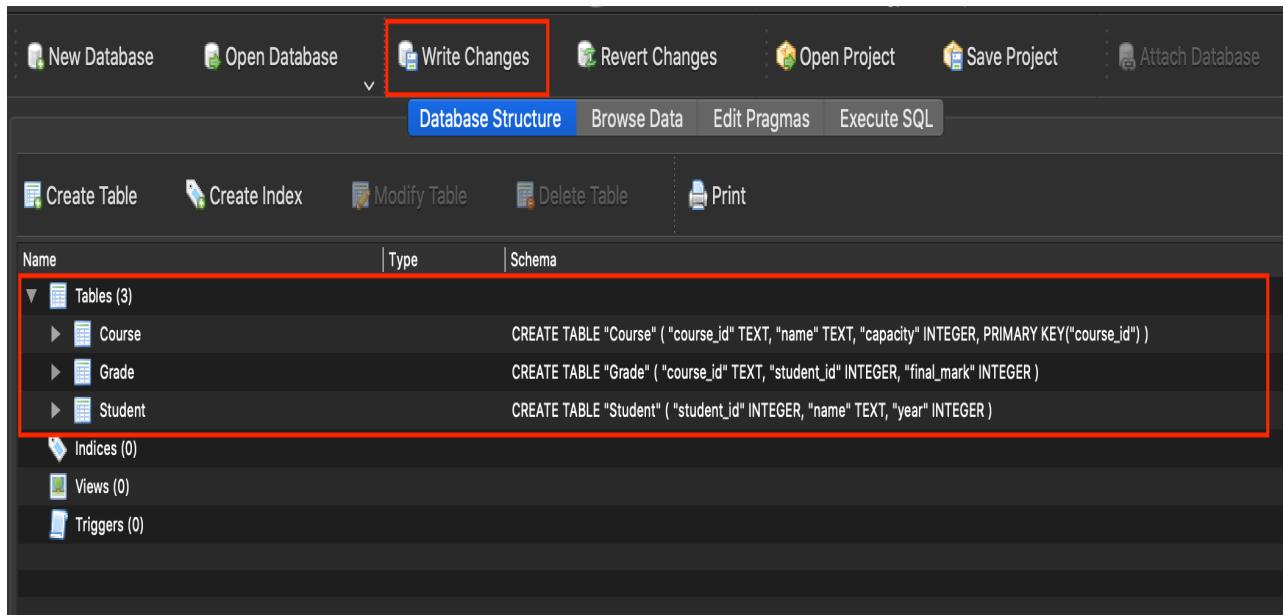
1. Click File -> Import.
2. Select the one of the CSV files student.csv, grade.csv and course.csv.
3. Check:
  - o Table name: in this tutorial we capitalize the first letter in table names.
  - o Field separator: set as ,.
  - o tick Column names in first row.
  - o Click OK if everything is fine.
4. Repeat the process for the other two CSV files.
5. Click Write Changes



Screenshot on how to add tables to the database from an existing CSV file



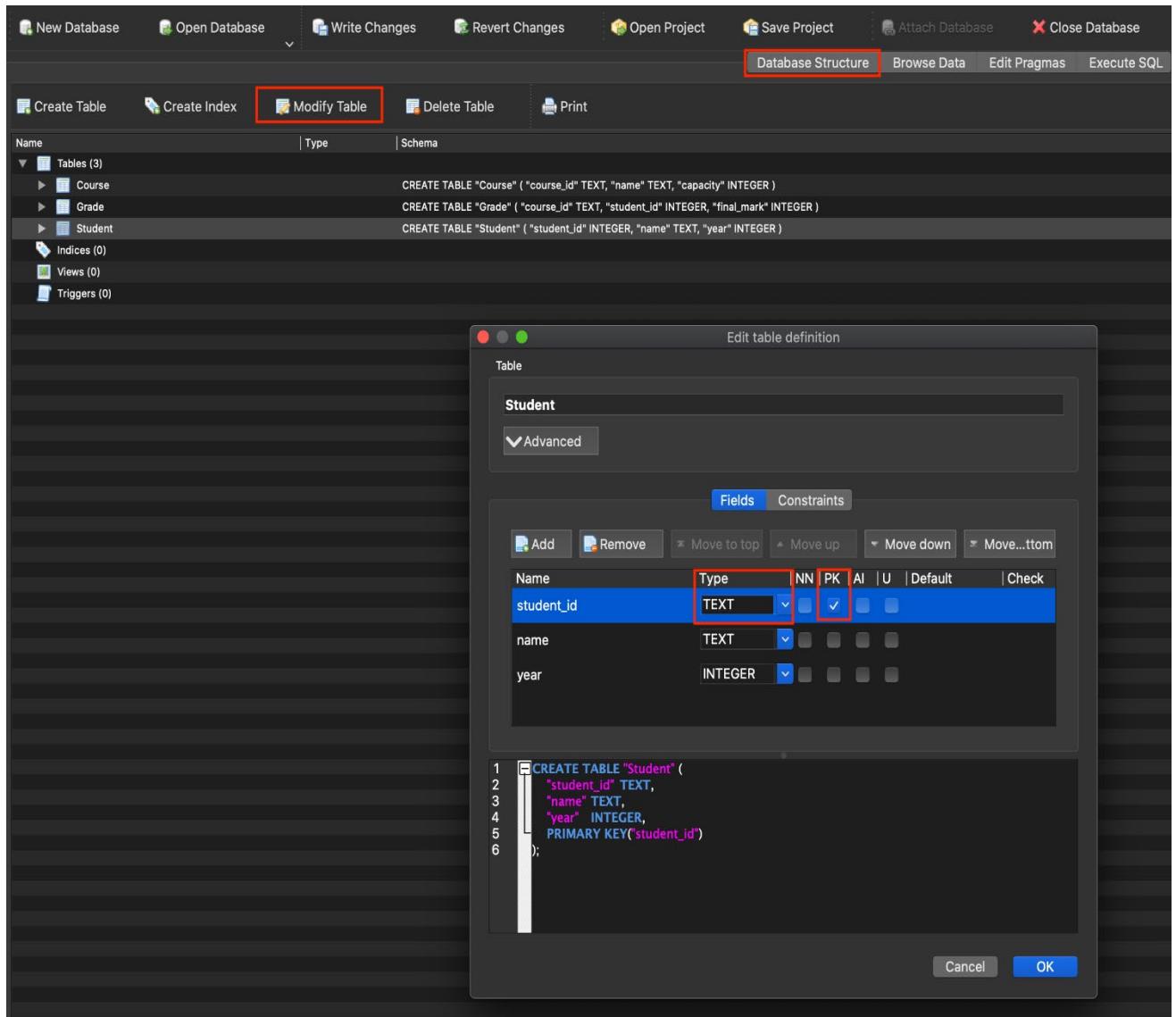
Screenshot on how to add tables to the database from an existing CSV file



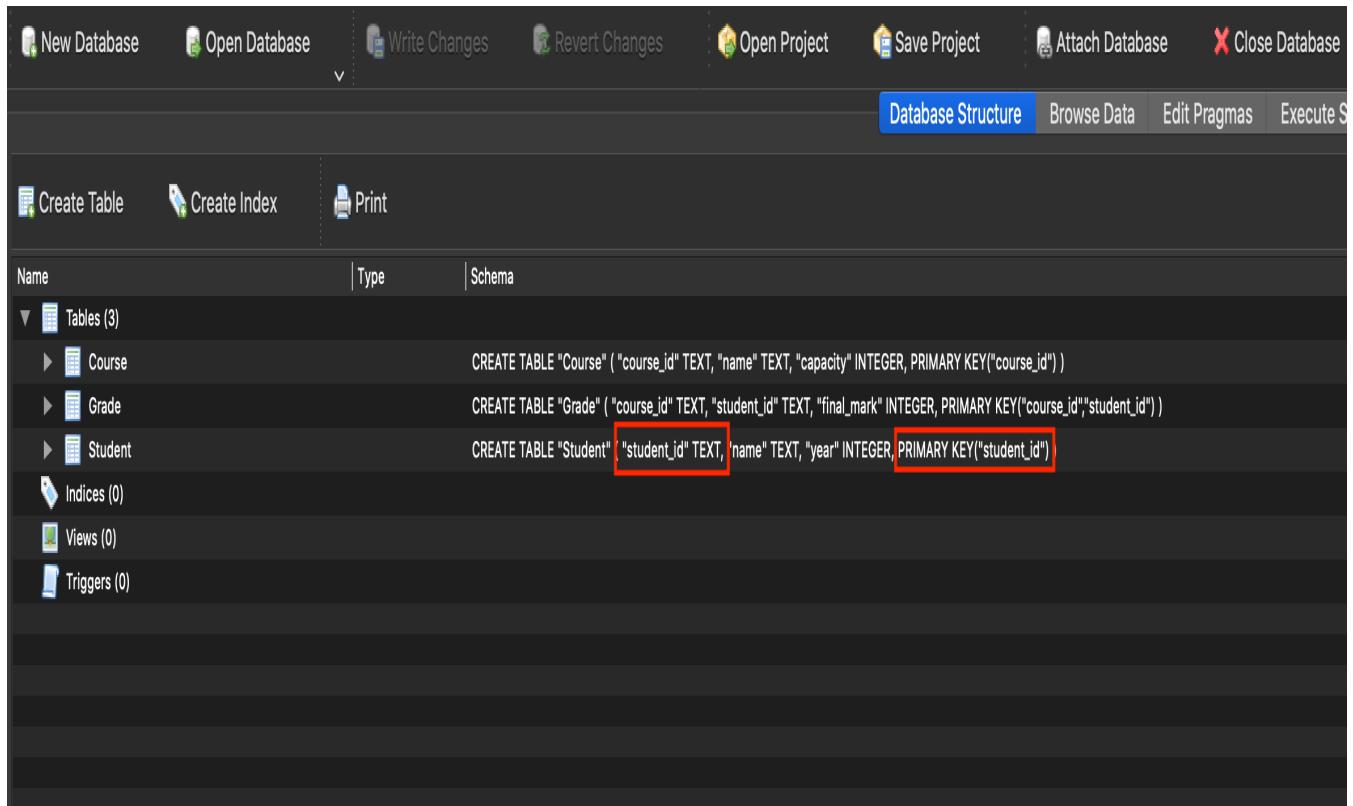
Screenshot on how to add tables to the database from an existing CSV file

## Update the Schema of Tables

1. Click Database Structure.
2. Select Student and click Modify Table.
3. Set student\_id with Type TEXT and select PK (primary key).
4. Click ok.
5. Similarly for Course table, set course\_id as primary key. For Grade table, set student\_id with Type TEXT and BOTH course\_id and student\_id as the primary key.
6. Click Write Changes.



Screenshot on how to update the schema of the tables



Screenshot of the updated schemas

## Browse Data

Click **Browse Data** to browse the data. When clicking on the list from **Table**, you should see three tables **Student**, **Grade**, and **Course**. Select the table that you want to browse.

The screenshot shows the 'Browse Data' tab selected in the top navigation bar. The 'Table' dropdown menu is open, showing 'Course', 'Grade', and 'Student'. The 'Student' option is selected and highlighted with a red box. Below the table list is a toolbar with various icons. The main area displays the data for the 'Student' table:

student_id	name	year
1	Ava Smith	2
2	Ben Johnson	3
3	Charlie Jones	1
4	Dan Norris	1
5	Emily Wood	1
6	Freddie Harris	2
7	Grace Clarke	2

Screenshot on how to browse data

# Manipulating Databases

## Add a New Table

Add a new table Teacher:

1. Click Database Structure.
2. Click Create Table.
3. Write Teacher in the first blank for the name of the table.
4. Under the Field, click Add to create a new attribute. Type staff\_id as Name and TEXT as Type. Select PK.
5. Under the Field, click Add to create a new attribute. Type name as Name and TEXT as Type.
6. Click ok.
7. Click Write Changes.

Screenshot of a database management software interface showing the creation of a 'Teacher' table.

The top menu bar includes:

- New Database
- Open Database
- Write Changes (highlighted with a red box)
- Revert Changes
- Open Project
- Save Project
- Attach Database
- Close Database

The main toolbar includes:

- Create Table (highlighted with a red box)
- Create Index
- Print

The Database Structure tab is selected.

The left sidebar shows the database structure:

- Tables (3):
  - Course
  - Grade
  - Student
- Indices (0)
- Views (0)
- Triggers (0)

The central area displays the 'Edit table definition' dialog for the 'Teacher' table.

The 'Table' section shows the table name: **Teacher**.

The 'Fields' tab is selected.

The table structure is defined as follows:

Name	Type	NN	PK	AI	U	Default	Check
staff_id	TEXT						
name	TEXT						

The 'Add' button is highlighted with a red box.

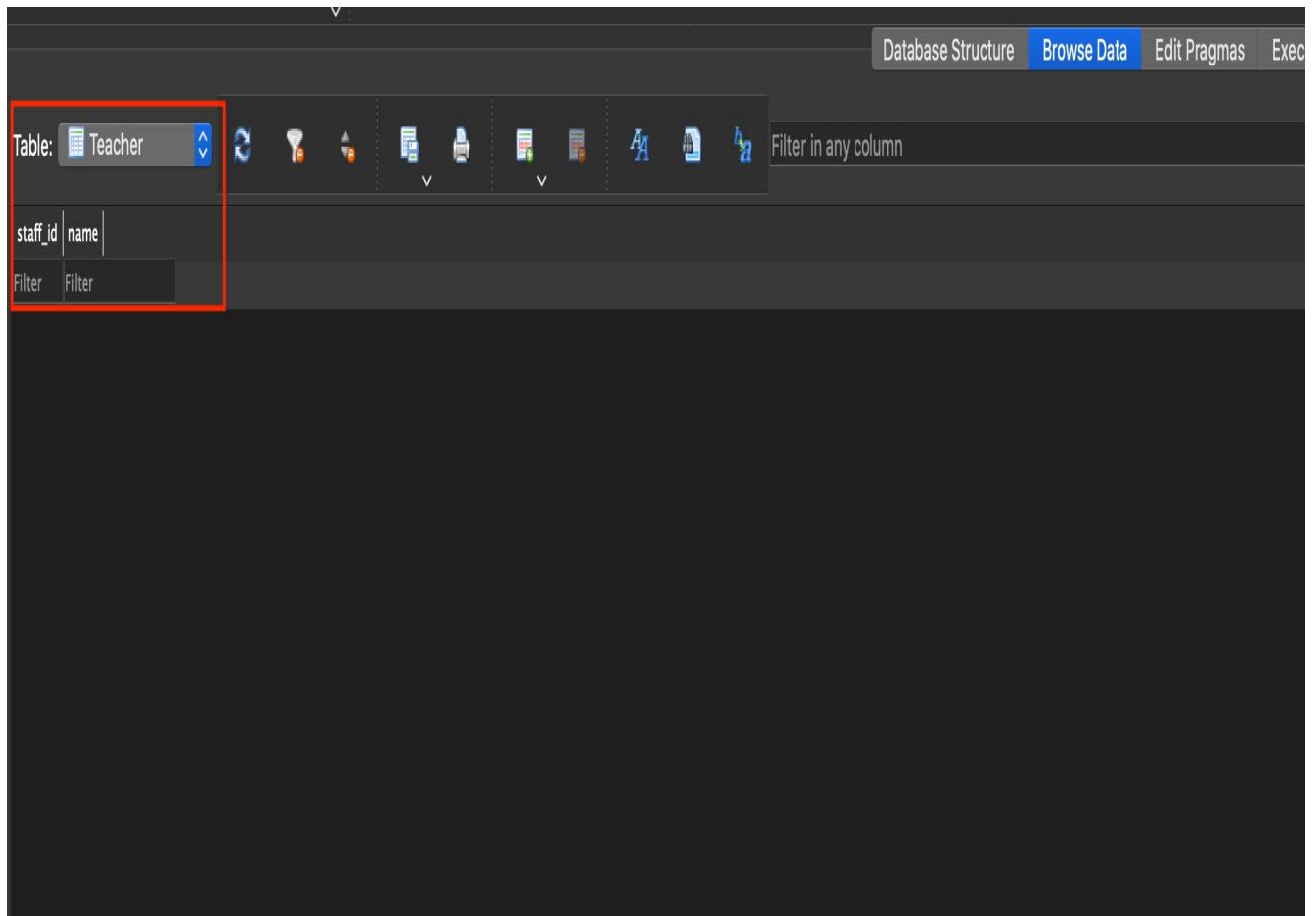
The bottom pane shows the generated SQL code:

```
CREATE TABLE "Teacher" (
    "staff_id" TEXT,
    "name" TEXT,
    PRIMARY KEY("staff_id")
);
```

Buttons at the bottom right are: Cancel and OK.

Screenshot on how to add a new table

Click `Browse Data` to browse the data. When click on the list from Table, you should see four tables. Select the table `Teacher` and it will display an empty table.

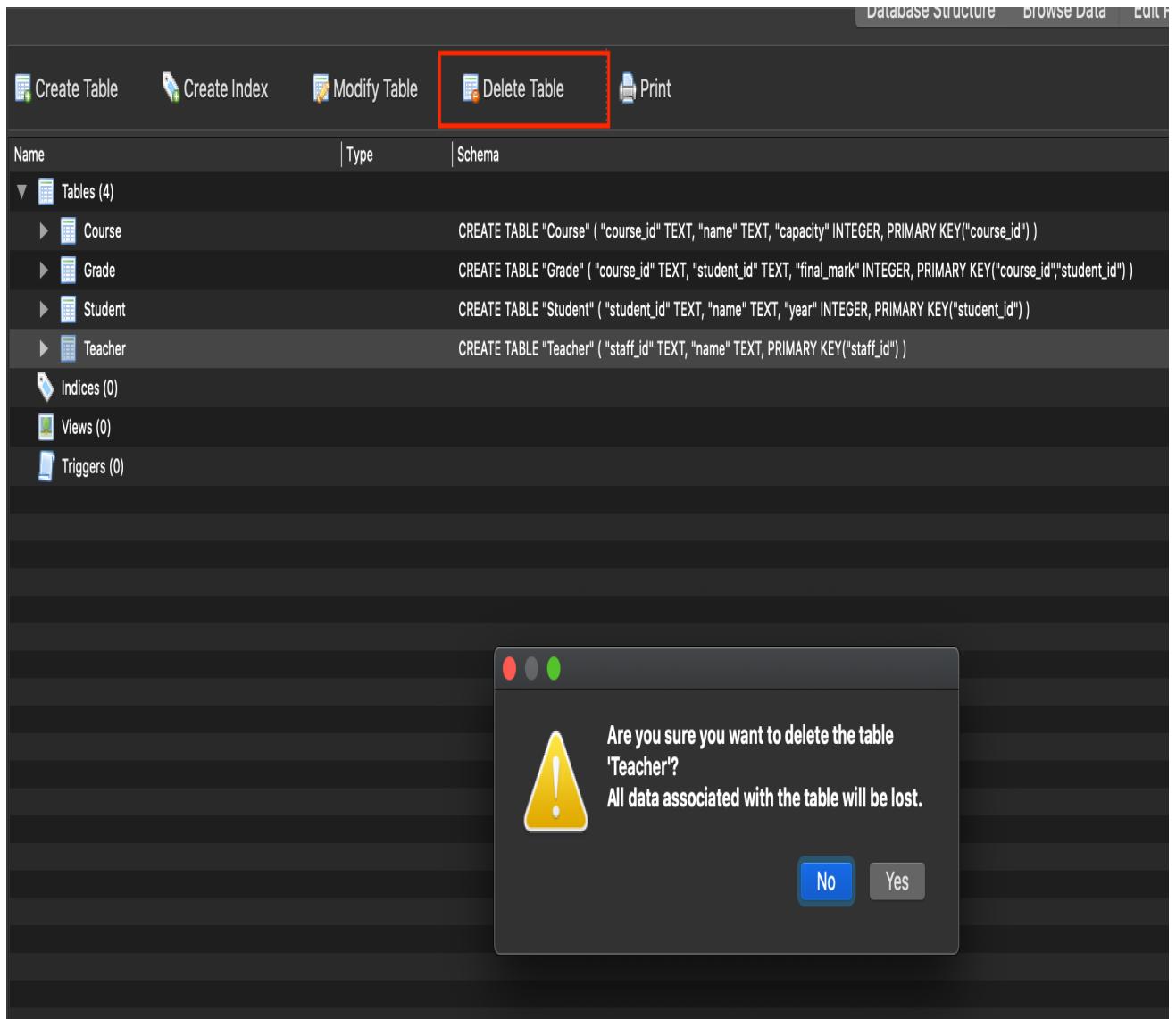


Screenshot of the newly created empty table

## Delete a Table

We can remove the `Teacher` table in the following way:

1. Click `Database Structure`.
2. Click on the `Teacher` table.
3. Click `Delete Table`.
4. Click `Write Changes`.



Screenshot on how to delete a table

Browse and now there are only three tables on the list from Table.

Name	Type	Schema
Tables (3)		
Course		CREATE TABLE "Course" ( "course_id" TEXT, "name" TEXT, "capacity" INTEGER, PRIMARY KEY("course_id") )
Grade		CREATE TABLE "Grade" ( "course_id" TEXT, "student_id" TEXT, "final_mark" INTEGER, PRIMARY KEY("course_id","student_id") )
Student		CREATE TABLE "Student" ( "student_id" TEXT, "name" TEXT, "year" INTEGER, PRIMARY KEY("student_id") )
Indices (0)		
Views (0)		
Triggers (0)		

Screenshot of list of tables

## Insert Tuples/Rows

Insert the year 1 student Harper Taylor with student ID 202029744 to Student:

1. Click Browse data.
2. Select Student from the list from Table.
3. Click the button the add an empty row.
4. Manually put the information into the new row.
5. Click Write Changes.

The screenshot shows a database management interface with the following details:

- Top Bar:** Includes buttons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Attach Database, and Close Database.
- Toolbar:** Shows icons for Undo, Redo, Cut, Copy, Paste, Find, and Delete.
- Table Selection:** Displays "Table: Student" with up and down arrows.
- Filtering:** Includes a "Filter in any column" search bar.
- Data Grid:** A table with columns student\_id, name, and year, containing 8 rows of data. Row 8 is highlighted with a red box.

student_id	name	year
1	Ava Smith	2
2	Ben Johnson	3
3	Charlie Jones	1
4	Dan Norris	1
5	Emily Wood	1
6	Freddie Harris	2
7	Grace Clarke	2
8	NULL	N...

Screenshot on how to insert a row

Table: **Student**

	student_id	name	year
	Filter	Filter	Filter
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	201933222	Freddie Harris	2
7	201875940	Grace Clarke	2
8	202029744	Harper Taylor	1

Screenshot on how to insert a row

Browse the table `Student` and makes sure the new student information is in the table.

## Update Tuples/Rows

Update the student ID of student Harper Taylor to 201929744:

1. Click `Browse` data.
2. Select `Student` from the list from `Table`.
3. Click on `student_id` cell on the row with `name` “Harper Taylor.”
4. In the window `Edit Database Cell`, change the value to `201929744`.
5. Click `Apply`.
6. Click `Write Changes`.

The screenshot shows the DBHub.io application interface. At the top, there is a navigation bar with icons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Attach Database, and Close Database. Below the navigation bar, there is a toolbar with buttons for Database Structure, Browse Data (which is highlighted with a red box), Edit Pragmas, and Execute SQL. The main area is divided into two panes. The left pane displays a table named 'Student' with columns 'student\_id', 'name', and 'year'. The data in the table is:

	student_id	name	year
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	20193222	Freddie Harris	2
7	201875940	Grace Clarke	2
8	202029744	Harper Taylor	1

The row with student\_id 8 is selected and highlighted with a red box. The right pane shows a detailed view of the selected row, with the 'student\_id' value '202029744' highlighted with a red box. Below this, there is a status message: 'Type of data currently in cell: Text / Numeric' and '9 characters' with an 'Apply' button. The bottom right pane shows a file browser interface with tabs for 'DBHub.io', 'Local', and 'Current Database', and a list of files.

Screenshot on how to update a row

Browse the table `student` and make sure the student information is updated in the table.

The screenshot shows the SQLite Database Browser interface. The top menu bar includes 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', and 'Save Project'. Below the menu is a toolbar with icons for database structure, browse data (selected), edit pragmas, and execute SQL. A sub-toolbar for the 'Student' table includes icons for sorting, filtering, and deleting. The main area displays the 'Student' table with the following data:

	student_id	name	year
	Filter	Filter	Filter
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	201933222	Freddie Harris	2
7	201875940	Grace Clarke	2
8	201929744	Harper Taylor	1

Screenshot of the table with a updated row

## Delete Tuples/Rows

Delete the record for the student Harper Taylor from table `Student`:

1. Click `Browse data`.
2. Select `Student` from the list from `Table`.
3. Click on the row with the name “Harper Taylor.”
4. Click the button to remove the row.
5. Click `Write Changes`.

The screenshot shows the DB Browser for SQLite interface. The top menu bar includes 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', and 'Attach Database'. Below the menu is a navigation bar with tabs: 'Database Structure', 'Browse Data' (which is highlighted with a red box), 'Edit Pragmas', and 'Execute SQL'. The main area is titled 'Table: Student' with a dropdown arrow. The toolbar below the table name contains several icons, with the delete icon (a bin) highlighted by a red box. A filter input field 'Filter in any column' is also visible. The table itself has columns 'student\_id', 'name', and 'year'. The data rows are:

	student_id	name	year
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	201933222	Freddie Harris	2
7	201875940	Grace Clarke	2

Screenshot on how to delete a row

Browse the table `Student` and make sure the row is deleted.

## Useful Links and Resources

- [What is SQLite](#): the official pages of SQLite
- [DB Browser for SQLite](#): the official pages of DB Browser for SQLite
- From Wikipedia
  - [SQL](#)
  - [SQLite](#)

# Basic SQL/SQLite Syntax and Queries

## SQL Recap

Structured query language (SQL) is one of the most widely used database languages, designed for managing data held in a relational database management system (RDBMS). It is used to:

- Create (database/tables)
- Manipulate (insert, update and delete tables/tuples)
- Query

a relational database. SQL:

- Is easy to learn: SQL commands are like English sentences, which makes it easy to read and write SQL queries
- Has well-defined standards: American National Standards Institute (ANSI) and International Organization for Standardization (ISO) have officially adopted the standard “Database Language SQL” language definition.

## Manipulating Databases

We have already seen how to use the **DB Browser for SQLite** graphical interface to add and delete tables and rows in a database. Here, we show how to do the same operations via SQL commands. We can run SQL code directly in **DB Browser for SQLite** in the following way:

1. Click on `Execute SQL` and type your SQL code in the top text box.
2. Click the run button to execute the SQL code.

Below is a screenshot of a SQL command executed in **DB Browser for SQLite**

The screenshot shows the DB Browser for SQLite interface. The toolbar at the top includes 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', 'Attach Database', and 'Close Database'. Below the toolbar, a menu bar has tabs for 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL', with 'Execute SQL' highlighted by a red box. A toolbar below the menu bar contains various icons, with the play button icon highlighted by a red box. The main window has a large red box around the SQL editor area. The SQL editor contains the following code:

```
1 CREATE TABLE Teacher(
2   staff_id TEXT,
3   name TEXT)
```

At the bottom of the SQL editor, there is a blue button labeled 'S...'. The results pane at the bottom of the interface displays the following output, also enclosed in a red box:

```
Execution finished without errors.
Result: query executed successfully. Took 0ms
At line 1:
CREATE TABLE Teacher (
  staff_id TEXT,
  name TEXT)
```

In what follows we will work with the `University` database that we have created previously using **DB Browser for SQLite**.

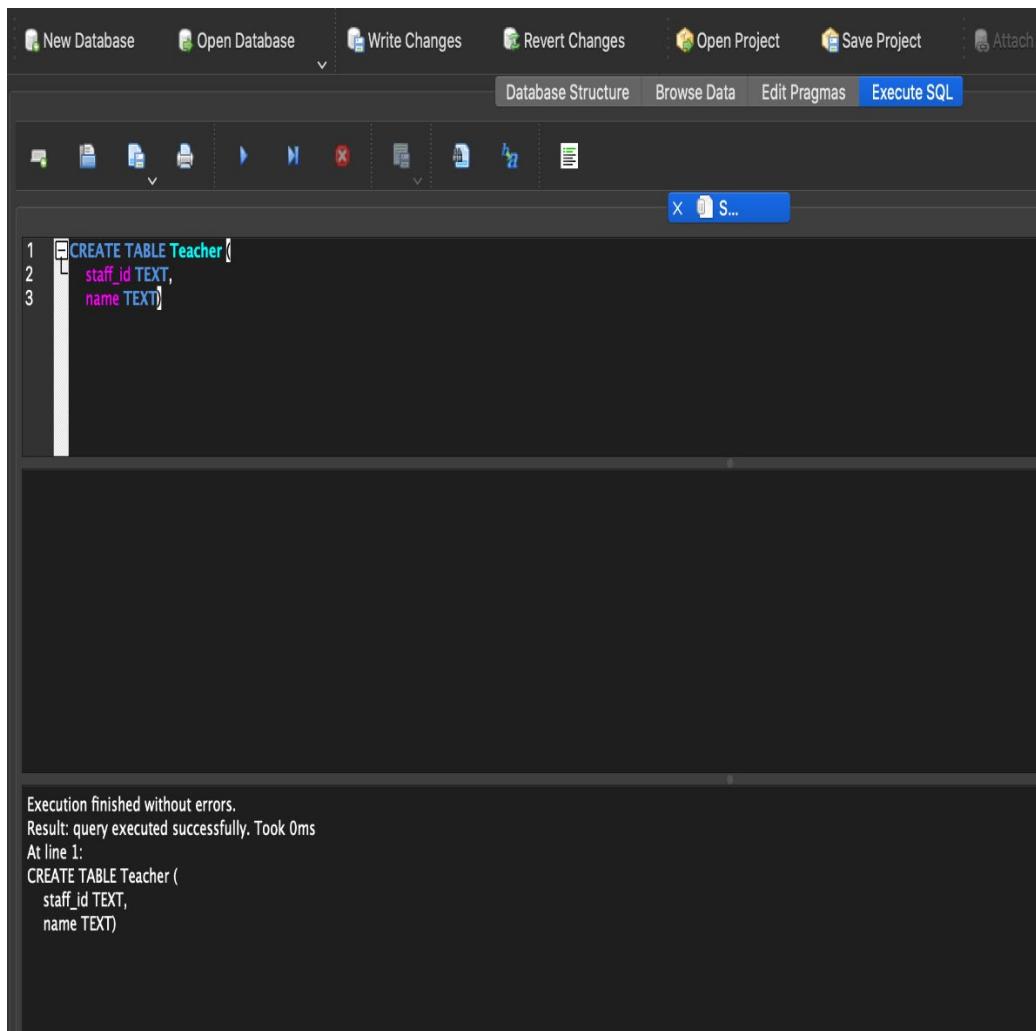
We open the database `University` in **DB Browse for SQLite** by clicking on `Open Database` and selecting the file `University.db`.

## Add a Table

Add a new table `Teacher` by using the `CREATE TABLE` command:

```
CREATE TABLE Teacher (
    staff_id TEXT,
    name TEXT)
```

and hitting the execute button above the text box.



Screenshot on how to add a table using SQL

Click `Write Changes`.

Now, click `Browse Data` to browse the database. When you click on the list `Table`, you should get four tables. Browse the table `Teacher` and it will display an empty table with attributes “`staff_id`” and “`name`.”

A screenshot of a database management software interface. The top menu bar includes 'Database Structure', 'Browse Data' (which is highlighted in blue), 'Edit Pragmas', and 'Exec'. Below the menu is a toolbar with various icons. A red box highlights the 'Table' dropdown menu and the 'Teacher' table name. The main area shows a table structure with two columns: 'staff\_id' and 'name'. Both columns have a 'Filter' button at the bottom. The table is currently empty.

staff_id	name
Filter	Filter

Screenshot of the newly created empty table

## Delete a Table

We can remove the Teacher table by the `DROP TABLE` command:

```
DROP TABLE Teacher
```

The screenshot shows the SQLite Database Browser interface. At the top, there are tabs for "Database Structure", "Browse Data", "Edit Pragmas", and a blue highlighted "Execute SQL". Below the tabs is a toolbar with various icons for database management. The main area contains a single line of SQL code: "1 | DROP TABLE Teacher". In the bottom right corner of the main window, there is a status message: "Execution finished without errors. Result: query executed successfully. Took 0ms At line 1: DROP TABLE Teacher".

Screenshot on how to delete a table

and click Write Changes

If you browse now, you will see that there are only three tables left.

The screenshot shows the MySQL Workbench interface with the 'Database Structure' tab selected. The main pane displays a list of database objects:

Name	Type	Schema
Tables (3)		
Course		CREATE TABLE "Course" ( "course_id" TEXT, "name" TEXT, "capacity" INTEGER, PRIMARY KEY("course_id") )
Grade		CREATE TABLE "Grade" ( "course_id" TEXT, "student_id" TEXT, "final_mark" INTEGER, PRIMARY KEY("course_id","student_id") )
Student		CREATE TABLE "Student" ( "student_id" TEXT, "name" TEXT, "year" INTEGER, PRIMARY KEY("student_id") )
Indices (0)		
Views (0)		
Triggers (0)		

Screenshot of list of tables

## Insert Tuples/Rows

Insert the year 1 student “Harper Taylor” with student ID 202029744 to Student by using the `INSERT INTO` command:

```
INSERT INTO Student VALUES(202029744, "Harper Taylor", 1)
```

The screenshot shows the SQLite Database Browser interface. The toolbar at the top includes 'New Database', 'Open Database', 'Write Changes' (which is highlighted in blue), 'Revert Changes', 'Open Project', and 'Save Project'. Below the toolbar are tabs for 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL' (also highlighted in blue). The main window contains a single line of SQL code: '1 | INSERT INTO Student VALUES(202029744, "Harper Taylor", 1)'. At the bottom of the window, the output of the query is displayed: 'Execution finished without errors.', 'Result: query executed successfully. Took 0ms, 1 rows affected', and 'At line 1: INSERT INTO Student VALUES(202029744, "Harper Taylor", 1)'.

Screenshot on how to insert a row

Click Write Changes.

Browse the table `Student` and make sure the new student information is in the table.

The screenshot shows a database interface with a toolbar at the top containing icons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Database Structure, Browse Data (which is selected), Edit Pragmas, and Execute SQL. Below the toolbar is a table header for 'Student' with columns 'student\_id', 'name', and 'year'. There are filter buttons for each column. The table body contains 8 rows of data. A new row has been added at the bottom with student\_id 8, name 'Harper Taylor', and year 1.

	student_id	name	year
	Filter	Filter	Filter
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	201933222	Freddie Harris	2
7	201875940	Grace Clarke	2
8	202029744	Harper Taylor	1

Screenshot of the table with a newly added row

## Update Tuples/Rows

Update the student ID of “Harper Taylor” to 201929744 by using the UPDATE command:

```
UPDATE Student
SET student_id = "201929744"
WHERE name = "Harper Taylor"
```

The screenshot shows a dark-themed SQLite database interface. At the top, there's a toolbar with icons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Attach. Below the toolbar, a navigation bar includes Database Structure, Browse Data, Edit Pragmas, and Execute SQL, with Execute SQL being the active tab. A toolbar below the navigation bar contains icons for file operations like New, Open, Save, and Print. The main area is a code editor with a blue status bar at the bottom right showing 'X S...'. The code in the editor is:

```
1 UPDATE Student
2 SET student_id = "201929744"
3 WHERE name = "Harper Taylor"
```

Below the code editor, a message window displays the results of the query execution:

```
Execution finished without errors.
Result: query executed successfully. Took 0ms, 1 rows affected
At line 1:
UPDATE Student
SET student_id = "201929744"
WHERE name = "Harper Taylor"
```

Screenshot on how to update a row

Click Write Changes.

Browse the table `Student` and make sure the new student information is in the table.

The screenshot shows a database interface with a toolbar at the top containing 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', and 'Save Project'. Below the toolbar is a menu bar with 'Database Structure', 'Browse Data' (which is selected and highlighted in blue), 'Edit Pragmas', and 'Execute SQL'. A sub-menu for 'Table' is open, showing 'Student' as the current selection. The main area displays the 'Student' table with the following data:

	student_id	name	year
1	201921323	Ava Smith	2
2	201832220	Ben Johnson	3
3	202003219	Charlie Jones	1
4	202045234	Dan Norris	1
5	201985603	Emily Wood	1
6	201933222	Freddie Harris	2
7	201875940	Grace Clarke	2
8	201929744	Harper Taylor	1

Screenshot of the table with an updated row

## Delete Tuples/Rows

Now, delete the record for the student “Harper Taylor” from table `Student`:

```
DELETE FROM Student  
WHERE name = "Harper Taylor"
```

Click `Write Changes`

The screenshot shows a dark-themed database management tool window. At the top, there are tabs: "Database Structure", "Browse Data", "Edit Pragmas", and "Execute SQL". The "Execute SQL" tab is active, highlighted in blue. Below the tabs is a toolbar with various icons. A status bar at the bottom of the toolbar displays the text "Execute all/selected SQL [⌘F, F5, ⌘R]" and a small "S..." button. The main area contains two lines of SQL code:

```
1 DELETE FROM Student  
2 WHERE name = "Harper Taylor"
```

Below the code, the output window shows the results of the execution:

```
Execution finished without errors.  
Result: query executed successfully. Took 0ms, 1 rows affected  
At line 1:  
DELETE FROM Student  
WHERE name = "Harper Taylor"
```

Screenshot on how to delete a row

Browse the table `Student` and make sure that there is no row for “Harper Taylor.”

## SQL Queries

SQL queries are used to select data from a database. The basic SQL query has the form:

```
SELECT A_1,A_2,...,A_n  
FROM R_1,R_2, ...,R_m  
WHERE conditions
```

- `SELECT` line: Specify the attribute(s) to retain in the result.
- `FROM` line: Specify the table(s) to query from.
- `WHERE` line: Determine which tuple(s) to select.

The SQL keywords `SELECT`, `FROM`, `WHERE` are case-insensitive, however the common convention is to write them in capital letters.

## Example 1: Conditions

We would like to return the student ID of all students who are in ST101:

```
SELECT student_id  
FROM Grade  
WHERE course_id = 'ST101'
```

The screenshot shows a SQLite database interface with the following details:

- Toolbar:** Includes buttons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Database Structure, Browse Data, Edit Pragmas, and Execute SQL (which is highlighted).
- SQL Editor:** A code editor containing the SQL query:

```
1 SELECT student_id  
2 FROM Grade  
3 WHERE course_id = 'ST101'
```
- Result Table:** A table showing the results of the query:

student_id
1 201921323
2 201985603
3 202003219
- Log Area:** Displays the execution log:

```
Execution finished without errors.  
Result: 3 rows returned in 27ms  
At line 1:  
SELECT student_id  
FROM Grade  
WHERE course_id = 'ST101'
```

Screenshot of the query result

If we want to show all attributes (`course_id`, `student_id`, `final_mark`), then we can use the symbol `*`, which means that all attributes should be selected:

```
SELECT *
FROM Grade
WHERE course_id = 'ST101'
```

The screenshot shows a dark-themed database management application window. At the top, there are several icons for database operations: New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Execute SQL (which is highlighted in blue). Below the toolbar is a row of small icons representing different database functions. The main area contains a code editor with the following SQL query:

```
1 | SELECT *
2 | FROM Grade
3 | WHERE course_id = 'ST101'
```

Below the code editor is a table displaying the results of the query:

	course_id	student_id	final_mark
1	ST101	201921323	78
2	ST101	201985603	60
3	ST101	202003219	47

At the bottom of the interface, a message box displays the execution results:

```
Execution finished without errors.
Result: 3 rows returned in 40ms
At line 1:
SELECT *
FROM Grade
WHERE course_id = 'ST101'
```

Screenshot of the query result

## Example 2: Several Tables

Suppose we want to get the names of the students who took the course with course ID `ST101`. Note the student name information is in the `Student` table whereas the information about which course the student took is in `Grade`. In order to perform the query we need to combine information from the `Student` and `Grade` tables.

```
SELECT Student.name
```

```
FROM Grade, Student  
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
```

The screenshot shows a SQLite database interface with the following details:

- Toolbar:** Includes "New Database", "Open Database", "Write Changes", "Revert Changes", "Open Project", "Save Project", and "Attach D".
- Tab Bar:** Shows "Database Structure", "Browse Data", "Edit Pragmas", and "Execute SQL" (which is selected).
- Query Editor:** Displays the SQL query:

```
1 SELECT Student.name  
2 FROM Grade, Student  
3 WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
```
- Result Table:** A table showing the results of the query:

	name
1	Ava Smith
2	Emily Wood
3	Charlie Jones
- Log Area:** Shows the execution log:

```
Execution finished without errors.  
Result: 3 rows returned in 35ms  
At line 1:  
SELECT Student.name  
FROM Grade, Student  
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
```

Screenshot of the query result

Note:

- On the WHERE line we use AND to specify that both conditions Grade.course\_id = 'ST101' and Student.student\_id = Grade.student\_id need to be satisfied.
- In this query we use Student.student\_id and Grade.student\_id instead of student\_id to specify which table the attribute is from. This is because the attribute student\_id exists in both Student and Grade, and we need to indicate which table the attribute is from to eliminate ambiguity.
- We have Student.name instead of name and Grade.course\_id instead of course\_id to specify which table the attribute is from. However, this is not necessary here; it is perfectly fine to instead write:

```
SELECT name
FROM Grade, Student
WHERE course_id = 'ST101' AND Student.student_id = Grade.student_id
```

The screenshot shows a database interface with a toolbar at the top containing 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', and 'Attachment' buttons. Below the toolbar is a navigation bar with tabs: 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL' (which is highlighted in blue). The main area contains a code editor with the following SQL query:

```
1 | SELECT name
2 | FROM Grade, Student
3 | WHERE course_id = 'ST101' AND Student.student_id = Grade.student_id
```

Below the code editor is a table result set:

	name
1	Ava Smith
2	Emily Wood
3	Charlie Jones

At the bottom of the interface, the message 'Execution finished without errors.' is displayed, followed by the results: 'Result: 3 rows returned in 38ms' and the query text again.

Screenshot of the query result

The reason we can do this is because the attribute `name` is only in the table `Student` and the attribute `course_id` is only in the table `Grade`.

If we want to order the result, we can use the `ORDER BY` clause. For example, if we want to order the names of the students in alphabetical order:

```
SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
```

The screenshot shows a SQLite database interface with the following details:

- Toolbar:** Includes "New Database", "Open Database", "Write Changes", "Revert Changes", "Open Project", "Save Project", and "Attach".
- Menu Bar:** Shows "Database Structure", "Browse Data", "Edit Pragmas", and "Execute SQL" (which is highlighted).
- Query Editor:** Displays the following SQL query:

```
1 SELECT Student.name
2 FROM Grade, Student
3 WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
4 ORDER BY Student.name
```
- Result Table:** A table titled "name" showing three rows of student names:

	name
1	Ava Smith
2	Charlie Jones
3	Emily Wood
- Log Area:** Shows the execution results:

```
Execution finished without errors.
Result: 3 rows returned in 28ms
At line 1:
SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
```

Screenshot of the query result

### Example 3: Multiple Conditions

We would like to get the name of the courses taken by the student Ava Smith or Freddie Harris:

```
SELECT Course.name
FROM Student, Grade, Course
WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id =
Grade.course_id
```

The screenshot shows the SQLite Manager interface with the following details:

- Toolbar:** New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Attach Database.
- Tab Bar:** Database Structure, Browse Data, Edit Pragmas, Execute SQL (selected).
- Toolbar Buttons:** Undo, Redo, Cut, Copy, Paste, Find, Replace, Delete, Select All, Sort Ascending, Sort Descending.
- Query Editor:** Contains the following SQL code:

```
1 | SELECT Course.name
2 | FROM Student, Grade, Course
3 | WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris') AND Student.student_id = Grade.student_id AND Course.course_id = Grade.course_id
```
- Result Table:** A table titled "name" showing the results of the query.

	name
1	programming for data science
2	Managing and Visualising Data
3	Managing and Visualising Data
4	Databases
- Message Area:** Execution finished without errors. Result: 4 rows returned in 28ms.
- Query History:** At line 1:  
SELECT Course.name  
FROM Student, Grade, Course  
WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris') AND Student.student\_id = Grade.student\_id AND Course.course\_id = Grade.course\_id

Screenshot of the query result

Note:

- **OR** is used because only one of the conditions `Student.name = 'Ava Smith'` and `Student.name = 'Freddie Harris'` is needed for `department`.
- If you look at the output closely, there are a few duplicate rows, which we can remove using `DISTINCT`:

```
SELECT DISTINCT Course.name
FROM Student, Grade, Course
```

```
WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id =
Grade.course_id
```

The screenshot shows a database management interface with the following components:

- Toolbar:** Includes buttons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Attach Database.
- Tab Bar:** Shows Database Structure, Browse Data, Edit Pragmas, and Execute SQL. The Execute SQL tab is selected.
- SQL Editor:** Displays the following SQL query:

```
1 SELECT DISTINCT Course.name
2 FROM Student, Grade, Course
3 WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris') AND Student.student_id = Grade.student_id AND Course.course_id = Grade.course_id
```
- Result Table:** A table showing the results of the query:

	name
1	programming for data science
2	Managing and Visualising Data
3	Databases
- Output Area:** Shows the execution status and results:

```
Execution finished without errors.
Result: 3 rows returned in 35ms
At line 1:
SELECT DISTINCT Course.name
FROM Student, Grade, Course
WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris') AND Student.student_id = Grade.student_id AND Course.course_id = Grade.course_id
```

Screenshot of the query result

## Example 4: Aggregation

We would like to calculate the average mark for each course according to the value of `course_id`:

```
SELECT course_id, AVG(final_mark)
FROM Grade
GROUP BY course_id
```

The screenshot shows the SQLite Database Browser interface. The toolbar at the top includes buttons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Database Structure, Browse Data, Edit Pragmas, and Execute SQL. The Execute SQL button is highlighted in blue. Below the toolbar is a toolbar with icons for file operations and database navigation. The main area contains a code editor window with the following SQL query:

```
1 | SELECT course_id, AVG(final_mark)
2 | FROM Grade
3 | GROUP BY course_id
```

Below the code editor is a table displaying the results of the query:

	course_id	AVG(final_mark)
1	ST101	61.66666666666667
2	ST115	82.33333333333333
3	ST207	66.5

At the bottom of the interface, a message box displays the execution status:

```
Execution finished without errors.
Result: 3 rows returned in 41ms
At line 1:
SELECT course_id, AVG(final_mark)
FROM Grade
GROUP BY course_id
```

Screenshot of the query result

The attribute name for the average mark looks different from other attributes. We can rename it using the `AS` clause:

```
SELECT course_id, AVG(final_mark) as avg_mark
FROM Grade
GROUP BY course_id
```

The screenshot shows a dark-themed database management application window. At the top, there's a toolbar with icons for 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', and 'Attach Data'. Below the toolbar is a navigation bar with tabs: 'Database Structure' (selected), 'Browse Data', 'Edit Pragmas', and 'Execute SQL'. The 'Execute SQL' tab is highlighted with a blue background. In the main area, there's a code editor containing the following SQL query:

```
1 | SELECT course_id, AVG(final_mark) as avg_mark
2 | FROM Grade
3 | GROUP BY course_id
```

Below the code editor is a table showing the results of the query:

	course_id	avg_mark
1	ST101	61.66666666666667
2	ST115	82.33333333333333
3	ST207	66.5

At the bottom of the window, a message indicates the execution status:

Execution finished without errors.  
Result: 3 rows returned in 40ms  
At line 1:  
SELECT course\_id, AVG(final\_mark) as avg\_mark  
FROM Grade  
GROUP BY course\_id

Screenshot of the query result

`GROUP BY` groups rows that have the same attribute value. It is often used to group rows before applying aggregation functions to an attribute of the group. Common aggregation functions are:

- `AVG`
- `COUNT`
- `SUM`

- AVG
- MAX
- MIN

In our example, we group the rows based on the `course_id`, and summarize the grouped rows by the average `final_mark` for each group.

## SQL Joins

In the previous examples, we have used conditions like `Student.student_id = Grade.student_id` in the WHERE to make use of multiple tables for query. The same queries can be written more concisely using JOIN clauses.

A JOIN clause combines rows from two or more tables based on related column(s) between them. The SQL language offers many different types of joins. Note that not all types of JOIN are implemented by the database engines.

- INNER JOIN: Select rows that have matching values in *both* tables based on the given columns
- NATURAL JOIN: Similar to INNER JOIN except that there is no need to specify which columns are used for matching values
- OUTER JOIN: Unlike INNER JOIN, unmatched rows in one or both tables can be returned. There are LEFT, RIGHT and FULL OUTER JOIN. SQLite only supports LEFT OUTER JOIN. For LEFT OUTER JOIN, all the records from the left table are included in the result.
- CROSS JOIN: Return the Cartesian product of the two joined tables, by matching all the values from the left table with all the values from the right table.

### **INNER JOIN**

As in Example 2, below we get the records about students who took the course with course ID ST101, but we also sort the student names in alphabetical order:

```
SELECT *
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
```

Database Structure Browse Data Edit Pragmas

Print text from current SQL Editor tab [⌘P] X S...

```
1 | SELECT *
2 | FROM Grade, Student
3 | WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
4 | ORDER BY Student.name
```

	course_id	student_id	final_mark	student_id	name	year
1	ST101	201921323	78	201921323	Ava Smith	2
2	ST101	202003219	47	202003219	Charlie Jones	1
3	ST101	201985603	60	201985603	Emily Wood	1

Execution finished without errors.  
Result: 3 rows returned in 39ms  
At line 1:  
SELECT \*  
FROM Grade, Student  
WHERE Grade.course\_id = 'ST101' AND Student.student\_id = Grade.student\_id  
ORDER BY Student.name

Screenshot of the query result

We can rewrite the above query using INNER JOIN:

```
SELECT *
FROM Student JOIN Grade ON Student.student_id = Grade.student_id
WHERE course_id = 'ST101'
```

```
ORDER BY Student.name
```

The screenshot shows a SQLite database application window. At the top, there are several menu icons: New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and others. Below the menu bar, there is a toolbar with various icons for database management. The main area has tabs for Database Structure, Browse Data, Edit Pragmas, and Execute SQL. The Execute SQL tab is currently selected. A blue button labeled "S..." is visible in the toolbar. The code input field contains the following SQL query:

```
1 | SELECT *
2 | FROM Student JOIN Grade ON Student.student_id = Grade.student_id
3 | WHERE course_id = 'ST101'
4 | ORDER BY Student.name
```

Below the code, the results are displayed in a table:

	student_id	name	year	course_id	student_id	final_mark
1	201921323	Ava Smith	2	ST101	201921323	78
2	202003219	Charlie Jones	1	ST101	202003219	47
3	201985603	Emily Wood	1	ST101	201985603	60

At the bottom of the interface, a message indicates the execution status:

```
Execution finished without errors.  
Result: 3 rows returned in 36ms  
At line 1:  
SELECT *  
FROM Student JOIN Grade ON Student.student_id = Grade.student_id  
WHERE course_id = 'ST101'  
ORDER BY Student.name
```

Screenshot of the query result

Note that

- We write `JOIN` instead of `INNER JOIN` as by default `INNER JOIN` is used when you do not specify the join type.
- The `ON` keyword specifies on what condition you want to join the tables.
- If you look at the result, `student_id` appears twice—this is because *all* the columns from both tables are returned.
- By using the `JOIN` clause, we separate the logic of combining the tables (`Student.student_id = Grade.student_id`) and the other condition (`course_id = 'ST101'`), which makes the SQL query more readable.

Instead of joining using `ON`, we can use `USING` with `JOIN` if the columns that we are joining have the same name. For example:

```
SELECT *
FROM Student JOIN Grade USING(student_id)
WHERE course_id = 'ST101'
ORDER BY Student.name
```

The screenshot shows a dark-themed SQLite database management interface. At the top, there's a toolbar with icons for 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', and 'Attach Database'. Below the toolbar, a menu bar has tabs for 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL'. The 'Execute SQL' tab is currently selected, indicated by a blue background. A toolbar below the menu bar contains icons for creating tables, inserting data, updating data, deleting data, and other database operations. To the right of the toolbar is a search bar with a magnifying glass icon and a placeholder 'S...'. The main area contains a code editor with the following SQL query:

```
1 SELECT *
2 FROM Student JOIN Grade USING(student_id)
3 WHERE course_id = 'ST101'
4 ORDER BY Student.name
```

Below the code editor is a table displaying the results of the query. The table has columns: student\_id, name, year, course\_id, and final\_mark. The data is as follows:

	student_id	name	year	course_id	final_mark
1	201921323	Ava Smith	2	ST101	78
2	202003219	Charlie Jones	1	ST101	47
3	201985603	Emily Wood	1	ST101	60

At the bottom of the interface, a message states 'Execution finished without errors.' followed by 'Result: 3 rows returned in 30ms'. The query text is also repeated at the bottom.

Screenshot of the query result

Note that

- The `USING` keyword specifies which column is used to select rows that have matching values in both tables.
- If you look at the result, `student_id` appears only once now.

#### **NATURAL JOIN**

```
SELECT *
FROM Student NATURAL JOIN Grade
WHERE course_id = 'ST101'
ORDER BY Student.name
```

The screenshot shows a dark-themed SQLite database management interface. At the top, there are several menu items: New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Attach Database. Below the menu is a toolbar with icons for creating tables, inserting data, selecting data, updating data, deleting data, and other database operations. A navigation bar includes Database Structure, Browse Data, Edit Pragmas, and Execute SQL, with Execute SQL being the active tab.

In the main area, a SQL query is entered:

```
1 SELECT *
2 FROM Student NATURAL JOIN Grade
3 WHERE course_id = 'ST101'
4 ORDER BY Student.name
```

Below the query, the results are displayed in a table:

	student_id	name	year	course_id	final_mark
1	201921323	Ava Smith	2	ST101	78
2	202003219	Charlie Jones	1	ST101	47
3	201985603	Emily Wood	1	ST101	60

At the bottom of the interface, the following message is displayed:

Execution finished without errors.  
Result: 3 rows returned in 33ms  
At line 1:  
SELECT \*  
FROM Student NATURAL JOIN Grade  
WHERE course\_id = 'ST101'  
ORDER BY Student.name

Screenshot of the query result

Note that:

- We do not specify how to join the two tables. The join condition is automatically identified.
- If you look at the result, `student_id` appears only once.

#### **LEFT JOIN**

When we run the following SQL commands to use `INNER JOIN` to combine the tables `Student` and `Grade`

```
SELECT *
FROM Student INNER JOIN Grade USING (student_id)
ORDER BY Student.name
```

New Database Open Database Write Changes Revert Changes Open Project Save Project Attach Database

Database Structure Browse Data Edit Pragmas Execute SQL

X S...

```
1 | SELECT *
2 | FROM Student INNER JOIN Grade USING (student_id)
3 | ORDER BY Student.name
```

	student_id	name	year	course_id	final_mark
1	201921323	Ava Smith	2	ST101	78
2	201921323	Ava Smith	2	ST115	92
3	202003219	Charlie Jones	1	ST101	47
4	202003219	Charlie Jones	1	ST115	67
5	201985603	Emily Wood	1	ST101	60
6	201933222	Freddie Harris	2	ST115	88
7	201933222	Freddie Harris	2	ST207	73
8	201875940	Grace Clarke	2	ST207	60

Execution finished without errors.  
Result: 8 rows returned in 48ms  
At line 1:  
SELECT \*  
FROM Student INNER JOIN Grade USING (student\_id)  
ORDER BY Student.name

### Screenshot of the query result

the record of students Ben Johnson and Dan Norris are not shown, because there are not corresponding records for these two students in the table Grade. If we instead use LEFT OUTER JOIN to combine the tables Student and Grade:

```
SELECT *
FROM Student LEFT JOIN Grade USING (student_id)
ORDER BY Student.name
```

The screenshot shows a database management application window. At the top, there is a toolbar with the following icons and labels from left to right: New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Attach Database. Below the toolbar is a navigation bar with tabs: Database Structure, Browse Data, Edit Pragmas, and Execute SQL. The Execute SQL tab is highlighted with a blue background. Underneath the navigation bar is a toolbar with several small icons, including a magnifying glass, a file folder, and a refresh symbol. To the right of this toolbar is a blue button with a trash can icon and the text "X S...".

In the main area, there is a code editor window containing the following SQL query:

```
1 | SELECT *
2 | FROM Student LEFT JOIN Grade USING (student_id)
3 | ORDER BY Student.name
```

Below the code editor is a results grid displaying the output of the query. The grid has columns labeled student\_id, name, year, course\_id, and final\_mark. The data is as follows:

	student_id	name	year	course_id	final_mark
1	201921323	Ava Smith	2	ST101	78
2	201921323	Ava Smith	2	ST115	92
3	201832220	Ben Johnson	3	NULL	NULL
4	202003219	Charlie Jones	1	ST101	47
5	202003219	Charlie Jones	1	ST115	67
6	202045234	Dan Norris	1	NULL	NULL
7	201985603	Emily Wood	1	ST101	60
8	201933222	Freddie Harris	2	ST115	88
9	201933222	Freddie Harris	2	ST207	73
10	201875940	Grace Clarke	2	ST207	60

At the bottom of the application window, there is a message area showing the execution status:

```
Execution finished without errors.
Result: 10 rows returned in 31ms
At line 1:
SELECT *
FROM Student LEFT JOIN Grade USING (student_id)
ORDER BY Student.name
```

Screenshot of the query result

we get a result where:

- All the students from the left table `Student` are included (including Ben Johnson and Dan Norris)
- The students with no corresponding record in the right table `Grade`, have `NULL` value in attributes `course_id` and `final_mark` from `Grade`.

#### CROSS JOIN

When we use `CROSS JOIN`:

```
SELECT *
FROM Student CROSS JOIN Grade
ORDER BY Student.name
```

The screenshot shows a dark-themed database management application window. At the top, there's a toolbar with icons for 'New Database', 'Open Database', 'Write Changes', 'Revert Changes', 'Open Project', 'Save Project', and 'Attach'. Below the toolbar, a navigation bar includes tabs for 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL', with 'Execute SQL' being the active tab. A set of small icons is located just below the navigation bar. The main area contains a code editor with the following SQL query:

```
1 SELECT *
2 FROM Student CROSS JOIN Grade
3 ORDER BY Student.name
```

	student_id	name	year	course_id	student_id	final_mark
1	201921323	Ava Smith	2	ST101	201921323	78
2	201921323	Ava Smith	2	ST101	201985603	60
3	201921323	Ava Smith	2	ST101	202003219	47
4	201921323	Ava Smith	2	ST115	201921323	92
5	201921323	Ava Smith	2	ST115	202003219	67
6	201921323	Ava Smith	2	ST115	201933222	88
7	201921323	Ava Smith	2	ST207	201933222	73
8	201921323	Ava Smith	2	ST207	201875940	60
9	201832220	Ben Johnson	3	ST101	201921323	78
10	201832220	Ben Johnson	3	ST101	201985603	60
11	201832220	Ben Johnson	3	ST101	202003219	47
12	201832220	Ben Johnson	3	ST115	201921323	92
13	201832220	Ben Johnson	3	ST115	202003219	67
14	201832220	Ben Johnson	3	ST115	201933222	88
15	201832220	Ben Johnson	3	ST207	201933222	73
16	201832220	Ben Johnson	3	ST207	201875940	60
17	202003219	Charlie Jones	1	ST101	201921323	78
18	202003219	Charlie Jones	1	ST101	201985603	60
19	202003219	Charlie Jones	1	ST101	202003219	47
20	202003219	Charlie Jones	1	ST115	201921323	92
21	202003219	Charlie Jones	1	ST115	202003219	67

Screenshot of the query result

we get 56 rows, which is number of rows in `student` (7) times number of rows in `Grade` (8).

## Useful Links and Resources

- [SQLite join](#) to learn more about how to join tables via SQLite.
- [SQLite select](#) to learn more about how to query via SQLite.
- [SQL](#) from Wikipedia.

# Creating and Manipulating Databases in R Using DBI



## Using Databases With R

So far, we have seen how to create, update and query a database using **DB Browser for SQLite**. Often the process does not stop there—we may want to do further analysis on the queried data, or we may want to store the data analyzed using R and Python back to the database. For data analysis, we could export the table / query result into a file (say a CSV file) and then read the file into R and Python. For storing the processed data, we could again export the data from R and Python to a file and then import the data into the database using **DB Browser for SQLite**.

Another (and probably a better) way is to do it all in R and Python—we can connect R and Python to the database to create, update and query as appropriate. We then can work on the query result directly without the need of saving and loading the query result to an external file. Similarly, we can directly store the analyzed data back to the database.

In these pages we will focus on how to interact with databases in R and Python. Here we will continue to use SQLite as the database engine, and we will use the **RSQLite** and **DBI** R packages to show that what we have been done so far on **DB Browser for SQLite** can also be done within in R. To highlight this point, we will use the same examples we used before. In this notebook, we will show how to connect and create databases and tables using R.

## Connecting to Databases

We load the library **DBI** and use the function `dbConnect()` to create an object, `conn`, to connect to the SQLite driver to manipulate the database `university.db`. If the database `university.db` exists in your working directory, the following code chunk will remove it.

```
# install.packages("RSQLite")
library(DBI)

if (file.exists("university.db"))
  file.remove("university.db")
[1] TRUE
conn <- dbConnect(RSQLite::SQLite(), "university.db")
```

We can list all tables in `university.db` using the function `dbListTable()` from **DBI**.

```
# list all tables
dbListTables(conn)
character(0)
```

Nothing is returned because we have not created any tables in the database yet.

## Creating Tables

Now we are going to create some tables to the database `university.db`. Like before, we will create the tables using data saved in the CSV files. We first read the CSV files into `data.frame` in R:

```
course <- read.csv("course.csv", header = TRUE)
student <- read.csv("Student.csv", header = TRUE)
grade <- read.csv("grade.csv", header = TRUE)
```

We then copy the data frames `student`, `grade` and `course` to tables in the database `university.db` using **DBI**'s `dbWriteTable()` function:

```
dbWriteTable(conn, "Course", course)
dbWriteTable(conn, "Student", student)
dbWriteTable(conn, "Grade", grade)
```

Now we can see there are three tables in the database:

```
dbListTables(conn)
[1] "Course"   "Grade"    "Student"
```

We can also browse any table in the database using the function `dbReadTable()` from **DBI**. For example, we can browse the `Student` table by:

```
dbReadTable(conn, "Student")
  student_id      name year
1  201921323     Ava Smith    2
2  201832220     Ben Johnson   3
3  202003219   Charlie Jones   1
4  202045234     Dan Norris    1
5  201985603    Emily Wood    1
6  201933222 Freddie Harris    2
7  201875940   Grace Clarke    2
```

Or we can see the attributes of `Student` by:

```
dbListFields(conn, "Student")
[1] "student_id" "name"       "year"
```

We can also check if the database has been properly created by opening the database in **DB Browser for SQLite** and browse the tables from there.

# Manipulating Databases

The simplest way to manipulate databases is to use the `dbExecute()` function. This function executes SQL statements and returns the number of rows affected. This allows us to leverage SQL commands to manipulate databases from within R. There are also some other functions in **DBI** that are specialized to perform certain tasks.

## Adding a New Table

We can add a new table by using the function `dbCreateTable()`. Alternatively, you can use `dbExecute()` to run the SQL command to create a new table.

```
dbCreateTable(conn, "Teacher", c(staff_id = "TEXT", name = "TEXT"))

# Alternative:
# dbExecute(conn,
#   "CREATE TABLE Teacher (
#     staff_id TEXT PRIMARY KEY,
#     name TEXT)")
```

When we list the tables, we can see four tables.

```
dbListTables(conn)
[1] "Course"    "Grade"     "Student"   "Teacher"
```

The table `Teacher` has two attributes with no rows.

```
dbListFields(conn, "Teacher")
[1] "staff_id" "name"
dbReadTable(conn, "Teacher")
[1] staff_id name
<0 rows> (or 0-length row.names)
```

## Deleting a Table

We can remove a table by using the function `dbRemoveTable()`:

```
dbRemoveTable(conn, "Teacher")

# Alternative:
# dbExecute(conn,
#   "DROP TABLE Teacher")
```

When we list the tables, we can now see three tables.

```
dbListTables(conn)
[1] "Course"    "Grade"     "Student"
```

## Inserting Tuples/Rows

Below we insert the year 1 student “Harper Taylor” with student ID 202029744 to `Student` by using the function `dbAppendTable()`:

```
dbAppendTable(conn, "Student", data.frame(student_id = "202029744",
                                         name = "Harper Taylor",
                                         year = 1))

[1] 1
# Alternative:
# dbExecute(conn,
# "INSERT INTO Student VALUES(202029744, 'Harper Taylor', 1)")
```

When we browse the table, we can see the new row has been added.

```
dbReadTable(conn, "Student")
student_id      name year
1  201921323    Ava Smith   2
2  201832220    Ben Johnson 3
3  202003219   Charlie Jones 1
4  202045234    Dan Norris   1
5  201985603    Emily Wood   1
6  201933222  Freddie Harris 2
7  201875940    Grace Clarke 2
8  202029744  Harper Taylor 1
```

## Updating Tuples/Rows

Below, we update the student ID of student Harper Taylor to 201929744 by `dbExecute()`. There is no specific function for updating a row in **DBI**.

```
dbExecute(conn,
"Update Student
SET student_id = '201929744'
WHERE name = 'Harper Taylor'")
[1] 1
```

When we browse the table, we can see the row has changed.

```
dbReadTable(conn, "Student")
student_id      name year
1  201921323    Ava Smith   2
2  201832220    Ben Johnson 3
3  202003219   Charlie Jones 1
4  202045234    Dan Norris   1
5  201985603    Emily Wood   1
6  201933222  Freddie Harris 2
7  201875940    Grace Clarke 2
8  201929744  Harper Taylor 1
```

## Deleting Tuples/Rows

Below, we delete the record for the student Harper Taylor from table `Student` using `dbExecute()`. There is no specific function for deleting a row in **DBI**.

```
dbExecute(conn,
"DELETE FROM Student
WHERE name = 'Harper Taylor'")
[1] 1
```

When we browse the table, we can see the row has been removed.

```
dbReadTable(conn, "Student")
  student_id      name year
1  201921323    Ava Smith   2
2  201832220    Ben Johnson  3
3  202003219  Charlie Jones  1
4  202045234    Dan Norris   1
5  201985603   Emily Wood   1
6  201933222 Freddie Harris  2
7  201875940  Grace Clarke  2
```

## Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()` from **DBI**:

```
dbDisconnect(conn)
```

## Useful Links and Resources

- [Using DBI](#): A guide on how to use **DBI** from RStudio
- [DBI reference manual](#)
- [RSQLite vignettes](#)

# Querying Databases in R Using DBI



## Using Databases With R

In the previous section, we have seen how to use R to connect, create and manipulate a database. Here, we will use R to query `university.db`. We first connect to the database `university.db` and list all the tables.

```
library(DBI)
conn <- dbConnect(RSQLite::SQLite(), "university.db")
dbListTables(conn)
[1] "Course"   "Grade"     "Student"
```

We should see all the three tables `Student`, `Course`, and `Grade` that we have created before.

## Querying Databases Using DBI

Once we formulate the query into an SQL SELECT statement, we can get the query result in R using the function `dbGetQuery()` or `dbSendQuery()` from **DBI**. The following examples show how we can run queries from within R.

### Getting Grades of a Course With `course_id "ST101"`

```
q1 <- dbGetQuery(conn,
"SELECT final_mark
FROM Grade
WHERE course_id = 'ST101'")
q1
final_mark
1          78
2          60
3          47
```

The second argument in `dbGetQuery()` is the SQL query statement we used in previous Sections, when queries were sent using DB Browser for SQLite.

```
class(q1)
[1] "data.frame"
```

Note that the `dbGetQuery()` returns a `data.frame`.

Alternatively, we can use `dbSendQuery()` and `dbFetch()`:

```

q1 <- dbSendQuery(conn,
"SELECT final_mark
FROM Grade
WHERE course_id = 'ST101'")
q1
<SQLiteResult>
  SQL  SELECT final_mark
  FROM Grade
 WHERE course_id = 'ST101'
  ROWS Fetched: 0 [incomplete]
    Changed: 0

```

Note `dbSendQuery()` only sends and executes the SQL query to the database engine. It does not extract any records.

```

dbFetch(q1)
  final_mark
1      78
2      60
3      47

```

When we run `dbFetch()`, the executed query result will then be fetched.

## Getting Names of Students in Alphabetical Order

```

dbGetQuery(conn,
"SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name")
Warning: Closing open result set, pending rows
          name
1      Ava Smith
2 Charlie Jones
3      Emily Wood

```

Or we can do it with NATURAL JOIN:

```

dbGetQuery(conn,
"SELECT Student.name
FROM Student NATURAL JOIN Grade
WHERE course_id = 'ST101'
ORDER BY Student.name")
          name
1      Ava Smith
2 Charlie Jones
3      Emily Wood

```

## Getting Courses Taken by Students Ava Smith or Freddie Harris

```

dbGetQuery(conn,
"SELECT DISTINCT Course.name
FROM Student, Grade, Course"

```

```

WHERE (Student.name ='Ava Smith' or Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id =
Grade.course_id")
          name
1  programming for data science
2 Managing and Visualising Data
3             Databases

```

Or we can do it using JOIN:

```

dbGetQuery(conn,
"SELECT DISTINCT Course.name
FROM (Student NATURAL JOIN Grade) S JOIN Course on Course.course_id =
S.course_id
WHERE S.name ='Ava Smith' or S.name ='Freddie Harris'")
          name
1  programming for data science
2 Managing and Visualising Data
3             Databases

```

## Calculating Average Mark for Each Course

```

dbGetQuery(conn,
"SELECT course_id, AVG(final_mark) as avg_mark
FROM Grade
GROUP BY course_id")
  course_id avg_mark
1      ST101 61.66667
2      ST115 82.33333
3      ST207 66.50000

```

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()`:

```
dbDisconnect(conn)
```

## Useful Links and Resources

- [Using DBI](#): A guide on how to use **DBI** from RStudio
- [DBI reference manual](#)
- [RSQLite vignettes](#)

# Querying Databases in R Using dplyr

## Introduction to dplyr

**dplyr** is an R package for data manipulation. It provides a set of functions, named as verbs, that can be used to carry out a wide range of data manipulation operations:

- The `mutate()` function adds new variables that are transformations of existing variables.
- The `select()` function picks variables based on their names.
- The `filter()` function picks cases based on their values.
- The `summarize()` function reduces multiple values down to a single summary.
- The `arrange()` function changes the ordering of the rows.

**dplyr** verbs operate on data frames, but they also, almost seamlessly apply to database tables! In particular **dplyr** allows you to use database tables as if they are data frames, by internally converting **dplyr** code into SQL commands using **dbplyr**. The following table compares the syntax used in SQL with **dplyr** syntax:

action	SQL	dplyr
select a column	SELECT	<code>select</code>
select a row	WHERE	<code>filter</code>
sort	ORDER BY	<code>arrange</code>
group	GROUP BY	<code>group_by</code>
aggregation	aggregation functions (e.g. <code>AVG()</code> ) in SELECT	<code>summarize</code>

See the [SQL Translation](#) article in **dbplyr**'s pages for more information.

## Pipe Operator

All of the **dplyr** functions take a data frame (or a [`tibble`](#)) as the first argument. In this way, the `%>%` operator from the **magrittr** R package can be used, so that user does not need to save intermediate objects or nest verbs. For example, the statement `x %>% f(y)` is equivalent to `f(x, y)`, and the result from one step is “piped” into the next step. You can think of the pipe operator as “then.”

# Querying Databases Using dplyr

## Connecting to the Database

Again, we first connect to the database `university.db` and list all the tables. We should see all the three tables `student`, `course` and `grade` that we have created before.

```
library(DBI)
conn <- dbConnect(RSQLite::SQLite(), "university.db")
dbListTables(conn)
[1] "Course"   "Grade"    "Student"
```

## Creating a Reference to Table

```
library(dplyr)
student_db <- tbl(conn, "Student")
grade_db <- tbl(conn, "Grade")
course_db <- tbl(conn, "Course")
```

By creating references to the tables as done above, we can treat `student_db`, `grade_db`, and `course_db` as data frames, and use **dplyr** functionality to query the database.

## Getting Grades of the Course With `course_id "ST101"`

```
q1 <- grade_db %>% filter(course_id == "ST101")
q1
# Source:  lazy query [?? x 3]
# Database: sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
# course_id student_id final_mark
# <chr>      <int>        <int>
1 ST101      201921323      78
2 ST101      201985603      60
3 ST101      202003219      47
```

The function `filter()` selects the rows in the `grade_db` which satisfy the condition `course_id == "ST101"`.

We can use the function `show_query()` to show the SQL query that **dbplyr** produced, when we run the code above:

```
show_query(q1)
<SQL>
SELECT *
FROM `Grade`
WHERE (`course_id` = 'ST101')
```

## Getting Names of Students in Alphabetic Order

If we want to work on more than one table in `dplyr`, we can use join or set operations. In this example we show how to use `inner_join()`. `arrange()` is then used to order the query result.

```
q2 <- inner_join(student_db, grade_db) %>%
  filter(course_id == "ST101") %>%
  select(name) %>%
  arrange(name)

q2
# Source:      lazy query [?? x 1]
# Database:   sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
# Ordered by: name
#   name
#   <chr>
1 Ava Smith
2 Charlie Jones
3 Emily Wood
```

The corresponding SQL query is:

```
show_query(q2)
<SQL>
SELECT `name`
FROM (SELECT `LHS`.`student_id` AS `student_id`, `name`, `year`,
`course_id`, `final_mark`
FROM `Student` AS `LHS`
INNER JOIN `Grade` AS `RHS`
ON (`LHS`.`student_id` = `RHS`.`student_id`)
)
WHERE (`course_id` = 'ST101')
ORDER BY `name`
```

## Getting Courses Taken by Ava Smith or Freddie Harris

Here we use `inner_join` to specify which attribute should be used to join by, in this case `course_id`. As both `student_db` and the `course_db` have the attribute `name`, we use the argument `suffix` to rename the attribute `name` to `name.student` and `name.course`, correspondingly. In this way we eliminate ambiguity.

```
q3 <- inner_join(student_db, grade_db, by = "student_id") %>%
  inner_join(course_db, by = "course_id", suffix = c(".student",
".course")) %>%
  filter(name.student == 'Ava Smith' | name.student == 'Freddie Harris') %>%
  select(name.course) %>%
  distinct()

q3
# Source:      lazy query [?? x 1]
# Database:   sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
#   name.course
#   <chr>
1 programming for data science
```

2 Managing and Visualising Data  
3 Databases

The corresponding SQL query is:

```
show_query(q3)
<SQL>
SELECT DISTINCT `name.course`
FROM (SELECT `student_id`, `LHS`.`name` AS `name.student`, `year`,
`LHS`.`course_id` AS `course_id`, `final_mark`, `RHS`.`name` AS
`name.course`, `capacity`
FROM (SELECT `LHS`.`student_id` AS `student_id`, `name`, `year`,
`course_id`, `final_mark`
FROM `Student` AS `LHS`
INNER JOIN `Grade` AS `RHS`
ON (`LHS`.`student_id` = `RHS`.`student_id`)
) AS `LHS`
INNER JOIN `Course` AS `RHS`
ON (`LHS`.`course_id` = `RHS`.`course_id`)
)
WHERE (`name.student` = 'Ava Smith' OR `name.student` = 'Freddie Harris')
```

Note the computer-generated SQL code that **dplyr** created internally is more complicated than the SQL code we wrote for the same query.

## Calculating Average Mark for Each Course

The combination of the verbs `group_by()` and `summarize()` are used to calculate the average `final_mark` for each `course_id`.

```
q4 <- grade_db %>%
  group_by(course_id) %>%
  summarize(avg_mark = mean(final_mark, na.rm = TRUE))
q4
# Source:  lazy query [?? x 2]
# Database: sqlite 3.34.0 [/Users/yiannis/Dropbox/ST2195_DB/Content
#   Development/H1/university.db]
  course_id avg_mark
  <chr>      <dbl>
1 ST101       61.7
2 ST115       82.3
3 ST207       66.5
```

The corresponding SQL query is:

```
show_query(q4)
<SQL>
SELECT `course_id`, AVG(`final_mark`) AS `avg_mark`
FROM `Grade`
GROUP BY `course_id`
```

## Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()` from `DBI`:

```
dbDisconnect(conn)
```

## Useful Links and Resources

- [Using dplyr with databases](#): A guide on how to use `dplyr` with databases from RStudio
- [dplyr vignettes](#): A introduction to `dplyr`
- [dbplyr SQL Translation](#)

# Creating and Manipulating Databases in Python



\*\* Note: The code chunks below should be run in the following order \*\*

## Using Databases With Python

We have shown how to create, update and query a database using DB Browser for SQLite and in R. Now we will illustrate how the same thing can be done in Python. Again we will continue using the university example.

## Connecting to Databases Using Python

We import the module `sqlite3` and use the function `connect()` to create an object, `conn`, to connect to the SQLite driver to manipulate the database `University.db`. If the database `University.db` exists in your working directory, the following code chunk will remove it.

```
# This makes sure you can run this notebook multiple times without errors
import os

try:
    os.remove('University.db')
except OSError:
    pass

import sqlite3
conn = sqlite3.connect('University.db')
```

## Creating Tables Using Python

Now we are going to create some tables to the database `University.db`. Like before, we will create the tables using the data saved in the CSV files. We first load the CSV files into `DataFrame` in Python:

```
import pandas as pd

student = pd.read_csv("student.csv")
course = pd.read_csv("course.csv")
grade = pd.read_csv("grade.csv")
```

We then write record stored in `DataFrames` `student`, `grade` and `course` as tables to the database `University.db` using the `DataFrame` method `to_sql()`.

```
# index = False to ensure the DataFrame row index is not written into the SQL table
s

student.to_sql('Student', con = conn, index = False)
course.to_sql('Course', con = conn, index = False)
grade.to_sql('Grade', con = conn, index = False)
```

Again, we can check if the database is created properly by opening the database in DB Browser for SQLite and browse the tables.

## Manipulate Databases Using Python

We can manipulate databases in Python by the `execute()` and `fetchall()` methods from the `sqlite3` module which performs SQL commands. This allows us to leverage the SQL commands we have learned to manipulate the databases in Python. We first need to create a cursor object `c`:

```
c = conn.cursor()
```

After that we can execute the SQL commands we learned before using the function `execute()` and `fetchall()`. For example, if we want to get all the tables in the database, we can run:

```
c.execute('''
SELECT name
FROM sqlite_master
WHERE type='table'
''')
<sqlite3.Cursor object at 0x1447c1f80>
```

The result is not returned until we run `fetchall`:

```
c.fetchall()
[('Student',), ('Course',), ('Grade',)]
```

We can see there are three tables in the database. If we want to browse the table `student` we can run (here we display the results as `pandas DataFrame`):

```
import pandas as pd
q = c.execute("SELECT * FROM Student").fetchall()
pd.DataFrame(q)

      0           1   2
0  201921323    Ava Smith  2
1  201832220    Ben Johnson  3
2  202003219  Charlie Jones  1
3  202045234    Dan Norris  1
```

```
4 201985603      Emily Wood  1
5 201933222    Freddie Harris 2
6 201875940      Grace Clarke 2
```

Note here we combine the use of `execute()` and `fetchall()` in one line.

## Add a New Table

We can add a new table by running the SQL command through `execute()`:

```
c.execute('''
CREATE TABLE Teacher (staff_id TEXT PRIMARY KEY,
name TEXT)
''')
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit() # save (commit) the changes
```

When we list the tables, we can see four tables.

```
c.execute('''
SELECT name
FROM sqlite_master
WHERE type='table'
''').fetchall()
[('Student',), ('Course',), ('Grade',), ('Teacher',)]
```

## Delete a Table

We can delete a table by running the SQL command through `execute()`:

```
c.execute("DROP TABLE Teacher")
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit()
```

When we list the tables, we can see three tables.

```
c.execute('''
SELECT name
FROM sqlite_master
WHERE type='table'
''').fetchall()
[('Student',), ('Course',), ('Grade',)]
```

## Insert Tuples/Rows

Insert the year 1 student Harper Taylor with student ID 202029744 to Student:

```
c.execute("INSERT INTO Student VALUES(202029744, 'Harper Taylor', 1)")

<sqlite3.Cursor object at 0x1447c1f80>

conn.commit()
```

When we browse the table, we can see the new row is added.

```
q = c.execute("SELECT * FROM Student").fetchall()

pd.DataFrame(q)

   0           1    2
0  201921323  Ava Smith  2
1  201832220  Ben Johnson  3
2  202003219  Charlie Jones  1
3  202045234  Dan Norris  1
4  201985603  Emily Wood  1
5  201933222 Freddie Harris  2
6  201875940 Grace Clarke  2
7  202029744 Harper Taylor  1
```

## Update Tuples/Rows

Update the student ID of student Harper Taylor to 201929744:

```
c.execute('''

UPDATE Student

SET student_id = "201929744"
WHERE name = "Harper Taylor"

'''')

<sqlite3.Cursor object at 0x1447c1f80>

conn.commit()
```

When we browse the table, we can see the row has changed.

```
q = c.execute("SELECT * FROM Student").fetchall()

pd.DataFrame(q)

   0           1    2
0  201921323  Ava Smith  2
1  201832220  Ben Johnson  3
2  202003219  Charlie Jones  1
3  202045234  Dan Norris  1
4  201985603  Emily Wood  1
5  201933222 Freddie Harris  2
```

```
6 201875940 Grace Clarke 2
7 201929744 Harper Taylor 1
```

## Delete Tuples/Rows

Delete the record for the student Harper Taylor from table `Student`:

```
c.execute('''
DELETE FROM Student
where name = "Harper Taylor"
''')
<sqlite3.Cursor object at 0x1447c1f80>
conn.commit()
```

When we browse the table, we can see the row has been removed.

```
q = c.execute("SELECT * FROM Student").fetchall()
pd.DataFrame(q)

      0           1   2
0  201921323    Ava Smith  2
1  201832220    Ben Johnson  3
2  202003219  Charlie Jones  1
3  202045234    Dan Norris  1
4  201985603   Emily Wood  1
5  201933222  Freddie Harris  2
6  201875940  Grace Clarke  2
```

## Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the method `close()` on `conn`:

```
conn.close()
```

## Useful Links and Resources

- [sqlite3](#)

# Querying Databases in Python

\*\* Note: The code chunks below should be run in the following order \*\*

## Using Databases With Python

In the first notebook we have seen how to use Python to connect, create and manipulate a database. Now we will use Python to query databases using the database `University.db` that we have created in the first notebook.

## Connecting to Databases Using Python

We first connect to the database `University.db` and list all the tables. We should see all the three tables `Student`, `Course`, and `Grade` that we have created in the first notebook.

```
import sqlite3
conn = sqlite3.connect('University.db')
c = conn.cursor()
c.execute("SELECT name FROM sqlite_master WHERE type='table'").fetchall()
[('Student',), ('Course',), ('Grade',)]
```

## Querying Databases Using Python

We can query databases in Python by the `execute()` and `fetchall()` methods from the `sqlite3` module which performs SQL commands. Here we display the results as a `Pandas` data frame. The SQL commands used here have been discussed in the previous notebooks.

### Example 1: Get Grades of the Course `course_id` ST101

```
q1 = c.execute('''
SELECT final_mark
FROM Grade
WHERE course_id = 'ST101'
''').fetchall()

import pandas as pd
pd.DataFrame(q1)

0    78
1    60
```

## Example 2: Get Names of Students in Alphabetical Order

```
q2 = c.execute('''
SELECT Student.name
FROM Grade, Student
WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
ORDER BY Student.name
''').fetchall()

pd.DataFrame(q2)
   0
0    Ava Smith
1  Charlie Jones
2    Emily Wood
```

## Example 3: Get Courses Taken by Ava Smith or Freddie Harris

```
q3 = c.execute('''
SELECT DISTINCT Course.name
FROM Student, Grade, Course
WHERE (Student.name ='Ava Smith' OR Student.name = 'Freddie Harris') AND
Student.student_id = Grade.student_id AND Course.course_id = Grade.course_id
''').fetchall()

pd.DataFrame(q3)
   0
0  programming for data science
1  Managing and Visualising Data
2            Databases
```

## Example 4: Calculate Average Mark for Each Course

```
q4 = c.execute('''
SELECT course_id, AVG(final_mark) as avg_mark
FROM Grade
GROUP BY course_id
```

```
''' .fetchall()

pd.DataFrame(q4)

   0           1
0  ST101  61.666667
1  ST115  82.333333
2  ST207  66.500000
```

## Disconnecting From Databases

After we finish manipulating the database, we can close the connection using the method `close` on `conn`:

```
conn.close()
```

## Useful Links and Resources

- [sqlite3](#)

# ST2195 Programming for Data Science

## Block 04 Programming Concepts

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:  
<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-4>

### Computer Programming

Computer programming is the process of writing instructions that a computer can follow in order to perform a particular task. Programming is very much like writing down a cooking recipe: the aim is to provide precise instructions, which, if followed to the letter by someone else, the prepared meal will have the same taste and look as what you had in mind. So, as with cooking recipes, the first and most critical stage of the programming process is to be absolutely clear on what is the goal of the program or what is the problem that the program solves. In other words, to define what the program does. For example, a spreadsheet application should enable the organization and analysis of rectangular data, a word processor should make it easy to edit and format text, optical character recognition (OCR) software should be able to identify the characters in an image and extract them in a text file, and so on.

The second stage is to plan the steps that the program should take to fulfil its purpose. Typically, the set of steps takes some *inputs* (analogous to ingredients and cooking equipment) and produces predetermined *outputs* (analogous to the prepared meal). For example, the input of an OCR program is the image of a document and the output is the recognized text, possible in a text file. This sequence of steps is called a programming *algorithm*. Note here that there may be more than one algorithms that produce the same output starting from the same input, each having its own advantages and disadvantages. Typically, the aim is to produce algorithms that are fast and reliable, and require predictable time and resources to run.

Of course, the more we want the computer to do, the more instructions we need to provide. Let us come back to the recipe analogy. It is not so difficult to write a recipe for making pasta. Just boil water, throw in the pasta for 10 minutes, add salt, drain, and serve. But how about cooking curry or making a birthday cake? It definitely requires more steps and ingredients.

# Avoid Spaghetti Programming

It is very important to spend some time to carefully plan your program, otherwise it is likely to create something that is difficult to understand, maintain or extend. *Spaghetti program* is a phrase commonly used to describe poorly planned or structured programs. Below is an example of a spaghetti program written in the **BASIC language** (one of the early programming languages) that first prints “This line prints first”, then “This line prints second” and then ends.

```
10 GOTO 50
20 PRINT "This line prints second"
30 END
40 GOTO 20
50 PRINT "This line prints first"
60 GOTO 40
```

Clearly, this program is more convoluted than it should be. Line 10 (the first line) sends the computer to line 50 (the fifth line). Then line 50 gets the computer to print `This line prints first`. Afterwards the computer follows the command in line 60 that sends it to line 40, which in turn sends it to line 20. There it tells the computer to print `This line prints second`. After that, the computer automatically follows the command on the next line, which is line 30 and tells it to end. Despite that the program consists of six lines, the `GOTO` command gets the computer to move between lines, making it very difficult for a human to understand the flow and the purpose of this program. In cases of several hundreds lines of code this kind of coding will create programs that are almost impossible to understand or maintain, and should be avoided.

# Structured Programming

It is essential to impose some structure to keep programs organized. Programs are usually divided into three distinct parts:

- Sequences
- Branches or conditional statements
- Iterations or loops

Branches and loops are examples of *control flow structures*.

Dividing a program into sequences, branches, and loops is very helpful in terms of isolating and organising groups of related commands into discrete chunks of code that can be modified without affecting the rest of the program.

## Sequences

A *sequence* is a set of steps that the computer should follow one after another. An example is the following running calorie calculator that takes as inputs the user's age, weight, average heart rate and time the user ran and outputs the calories burned.

1. Calculate  $[(Age \times 0.2017) + (Weight \times 0.09036) + (Heart\ Rate \times 0.6309) - 55.0969] \times Time / 4.184$
2. Store the result from the calculation in 1 in `Calories Burned`
3. Print the value of `Calories Burned`

## Branches

Branches consist of two or more options each of which leads to potentially different outputs. The program determines which group of commands should be followed according to whether a particular condition is satisfied or not.

For example, at the end a video game asks you, “Do you want to play again (Yes or No)?” If you choose “Yes”, the program lets you play the video game again. If you choose “No”, the program stops running, as shown in the figure below.



Branch flowchart; image taken from Wang (2008)

## Iterations

Sometimes you may want the computer to run the same commands over and over again. For example, a program might ask the user for a password. If the user types an invalid password, the program displays an error message and asks the user to type the password again. If you wanted your program to ask the user for a password three times, you could write the same group of commands to ask for the password three times, but that would be wasteful. Not only would this force you to type the same commands multiple times, but if you wanted to modify these commands, you'd have to modify them in three different locations as well. Loops are basically a shortcut to writing one or more commands multiple times.

An *iteration*, or loop, consists of two parts:

- The group of commands that the loop repeats
- A command that defines how many times the loop should run

# Top Down Programming

For small programs, organizing a program into sequences, branches, and loops works well. But the larger your program gets, the harder it can be to view and understand the whole thing. So a second feature of structured programming involves breaking a large program into smaller parts where each part performs one specific task. This is also known as top-down programming; i.e. designing your program by identifying the main (top) task that you want your program to solve.

For example, if you wanted to write a program that predicts the next winning lottery numbers, that is a top task of your program. Then, smaller tasks may involve

- Identifying the lottery numbers that tend to appear often.
- Picking the six numbers that have appeared most often and displaying those as the potential future winning numbers.

The idea is that writing a large program may be tough, but writing a small program is easier. So if you keep dividing the tasks of your program into smaller and smaller parts, eventually you can write a small, simple program that can solve that task. Then you can put these small programs together like building blocks, and you'll have a well-organized big program. Also, if you need to modify part of the large program, you just need to find the small program that needs changing, modify it, and plug it back into the larger program. Ideally, each small program should be small enough to fit on a single sheet of paper. This makes each small program easy to read, understand, and modify. When you divide a large program into smaller programs, each small program is a *subprogram*.

# Documenting Programs and Comments

Despite the fact that most programming languages often attempt to use self-explanatory commands such as `print, read.csv` etc, it is often very difficult to understand what different parts of code are doing. For this reason, programmers usually add explanations directly into the code by using *comments*. A comment is nothing more than text embedded in the source code. To keep the compiler from thinking a comment is an actual command, special syntax is used to signify a comment. In R and Python all text that follows the character `#` in a line contains comments. Comments are typically short explanations that describe what the code does. Can you understand what the following sequence of R commands does and why?

```
A <- 2  
B <- 3  
C <- sqrt(A * A + B * B);
```

`A` and `B` are multiplied by themselves, the results are added together, and then the square root of the sum is stored in `C`. But this does not tell us why this code is doing this. Below we see the same set of commands but with some comments added to the code:

```
A <- 2  
B <- 3  
# Calculates the hypotenuse of a triangle (C) ' using the Pythagoras theorem: C2 <-  
A2 + B2  
C <- sqrt(A * A + B * B)
```

Even if one does not know (or care) about Pythagoras' theorem, the comments are helpful in understanding what the purpose of this line is. Another example with more comments and information is given below

```
# This formula uses the epidemic model to calculate the spread of a flu epidemic as
# a function of time. P is the current population of a city, t is time measured in weeks,
# c is the number of people in contact with an infected person and B is constant
# value that can be determined by the initial parameters of the flu epidemic

P <- 10000
B <- 1
c <- 2000
t <- 2
F <- P / (1 + (B * exp(-c * t)))
```

Comments are therefore useful to explain both what code does and how it works. But having too many comments in a computer program is usually a sign that the code is too complicated. So, instead of writing lengthy comments, a good programmer aims to write code which is as self-explanatory as possible and uses comments only if necessary.

A program can be documented using comments that explain the purpose of one or more lines of code or of entire subprograms. Comments can also be used to provide information such as identifying the original programmer, their contact details, the license of the program, etc.

## Useful Links and Resources

- [Wikipedia's Beginners' All-purpose Symbolic Instruction Code \(BASIC\) language page](#)
- [Wikipedia's Spaghetti Code page](#)

## References

Wang, W. (2008). *Beginning programming all-in-one desk reference for dummies*. Wiley.

# Variables, Control Flow Structures and Functions



\*\* Note: The code chunks below should be run in the following order \*\*

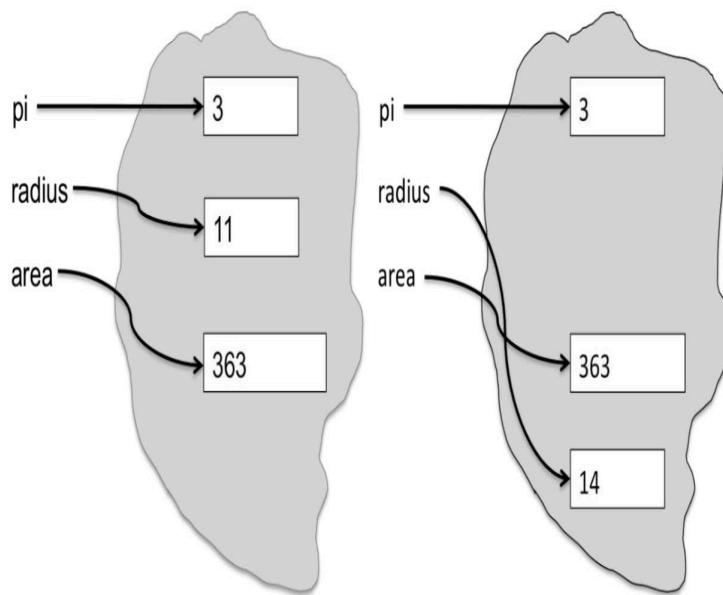
## Variables

Variables provide a way to associate names with storage points in the computer that the programs can access and manipulate if needed. Variables can have different data types such as `float`, `int`, `string`, `vector`, `array`, `matrix` etc. Perhaps you are familiar with the term variable in mathematics. Programming variables are similar. A variable is a way of saving a piece of information with a specific name that we can easily reuse several times in our code. In line with the idea of variability, a programming variable allows us to easily change a value throughout our code.

Consider the following chunk of R code based on an example from Guttag (2013):

```
pi <- 3
radius <- 11
area <- pi * (radius^2)
radius <- 14
```

First, we assign the names `pi` and `radius` to different storage points that contain data of type `int` in this case. Then we assign the name `area` to a third storage point containing the value of the product between the square of `radius` and `pi`. This is represented graphically in the left panel of the figure below:



## Variables assignment; image taken from Guttag (2013)

The last line of code then changes `radius` to another value `radius <- 14` as shown in the right panel of the figure above. Note that this assignment has no effect on the value of the variable `area` which remains `3*(11**2)`. By the way, if you believe that the actual value of `pi`, reflecting the famous number  $\pi$  is not 3, you are right. This was done for illustration purposes only. A value of `3.14159` would offer a more accurate answer.

At this point it may be relevant to clarify the difference between variables and data. Variables reflect the assignment to storage points that contain data. So in that sense and if you are familiar with the term *random variable* in probability and statistics and the notation  $(X)$  vs  $(x)$ , a programming variable refers to the mathematical function denoted by  $(X)$  whereas data relates the realised value of a random variable  $(x)$ .

Note also that the names assigned are important as programs are also read by humans. It is not always easy to write programs that work correctly and programmers often devote a lot of time to understand why programs do not behave in the way they should. A good choice of variable names can provide substantial help in making computer programs easier to read. Consider the two code chunks:

```
a = 3.14159
b = 11.2
c = a * (b**2)
```

and

```
pi = 3.14159
diameter = 11.2
area = pi * (diameter**2)
```

As far as R or Python is concerned, the two code chunks do exactly the same thing. Looking at the first chunk, we see nothing wrong with the code. But a quick glance at the code in the second chunk immediately raises suspicions as to whether the program really does what the programmer wanted to; either the variable `diameter` should have been named `radius`, or `diameter` should have been divided by 2 before squaring it in the calculation of the area.

In R and Python, variable names can contain upper-case and lower-case letters and digits (but they cannot start with a digit), and the dot or underline characters. Variable names are case-sensitive.

## Control Flow Structures

There are two primary tools of control flow: conditional statements (or branches) and iterations (or loops). Conditional statements allow you to run different code depending on the evaluation of a condition. Standard examples are the `if else` and `switch()` statements in R. Iterations, such as the `for` and the `while` loops, allow to repeatedly run code, potentially by changing the inputs in a particular way.

In this section we have adapted some material from chapter 5 of Wickham (2019). We strongly recommend taking a look at Chapter 5 of this resource, which is freely available online [here](#).

### Conditional Statements

It is possible to create programs that can make decisions based on the inputs you provide or the value that a variable has during run time.

With conditional statements we can write programs that determine whether a particular part of the code should be executed instead of another.

We start with the `if` statement. The basic forms of `if` statements encountered in R are

```
if (condition) true_action  
if (condition) true_action else false_action
```

If the `condition` in the above code evaluates to `TRUE`, the `true_action` is performed; if the `condition` evaluates to `FALSE`, the `false_action` is performed. Of course, specifying a `false_action` is optional. If the actions are compound statements of more than one command they need to be contained within brackets `{ }`

```
x <- 105  
  
if (x > 100) {  
  print("A")  
  print("B")  
}  
  
[1] "A"  
[1] "B"
```

We can also have nested `if` statements. For example, the following code chunk will print “A” if  $(x > 90)$ , “B” if  $(80 < x \leq 90)$ , and “C” otherwise (if  $(x \leq 80)$ ).

```
x <- 80  
  
if (x > 90) {  
  print("A")  
} else if (x > 80) {  
  print("B")  
} else
```

```
print("C")
[1] "C"
```

Try running the above with different values for `x` to confirm that the nested if statements do what we want them to.

Note that the `condition` should be or evaluate to Boolean type, i.e. either `TRUE` or `FALSE`. Most other inputs will generate an error.

A similar conditional statement in R is the `switch()` statement. It lets us replace code like:

```
x <- "c"

if (x == "a") {
  "A"
} else if (x == "b") {
  "B"
} else if (x == "c") {
  "C"
} else {
  stop("Invalid `x` value")
}

[1] "C"
```

with the more compact

```
x <- "b"

switch(x,
  "a" = "A",
  "b" = "B",
  "c" = "C",
  stop("Invalid `x` value")
)
[1] "B"
```

## Iterations

Iterations (also known as loops) are programming structures that repeat a sequence of instructions until a specific condition is met. Programmers use iterations extensively to cycle through values, compute sums of numbers, repeat actions, and many other things. Oftentimes, a certain operation needs to be repeated many times with little variation in order to achieve a goal. Rather than copying and pasting the commands over and over again, a loop can be used instead. In addition to simplifying writing programs, loops offer another advantage. Imagine there is an error in the code you copied and pasted several times. You will then have to go and correct this error at every instance of this code. But if a loop had been used, the correction would only need to be made once!

There are three main types of iterations: the `for` loop, the `while` loop and the `repeat` loop. The `for` loops are used to iterate over items in a vector. They have the following basic form:

```
for (item in vector) perform_action
```

For each `item` in `vector`, `perform_action` is called once; then the loop moves to the next `item` of in the `vector`, and so on. For example, the following code prints all the numbers between `1` and `10`:

```
for (i in 1:10) {  
  print(i)  
}  
  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10
```

To illustrate the points made earlier consider the naive alternative code for printing all the numbers between `1` and `10` as above.

```
print(1);print(2);print(3);print(4);print(5);print(6);print(7);print(8);print(9);pr  
int(10)  
  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10
```

Now imagine what would happen if we wanted to print all numbers up to `10000`; or if we accidentally copied and pasted `prin` instead of `print`. A handy rule of thumb in programming is to try to avoid copying and pasting more than twice.

It is possible to also terminate a `for` loop early. There are two ways to do it:

- `next` stops the current iteration and moves to the next `item` in `vector`.
- `break` stops the loop.

```

for (i in 1:10) {
  if (i < 3)
    next
  print(i)
  if (i >= 5)
    break
}
[1] 3
[1] 4
[1] 5

```

It is a good idea to pre-define the objects that will hold the outputs of the loop, otherwise the loop can be very slow. In the code below we begin by defining the empty vector `k` as the vector that will hold the results from each iteration of the `for` loop. Note that it has dimension `3` which the same as that of the vector we iterate over.

```

k <- numeric(3)
for (i in 1:3) {
  k[i] = i^4
}
print(k)
[1] 1 16 81

```

The `for` loops are useful if you know in advance the set of values that you want to iterate over. If you don't know, there are two related tools with more flexible specifications:

- `while (condition) action`: Performs `action` while `condition` is or evaluates to `TRUE`.
- `repeat (action)`: Repeats `action` forever (i.e. until it encounters `break`).

We can rewrite any `for` loop and use `while` instead. We can also rewrite any `while` loop and use `repeat` instead. But the converses are not true. More specifically, `while` is more flexible than `for`, and `repeat` is more flexible than `while`. It is good practice, however, to use the simplest solution to a problem, so it is usually recommended to use `for` if it suits the purpose of the program and only consider alternatives if they are really needed.

## Functions

Functions are essential in splitting the program into smaller parts and allow to automate common tasks in a more powerful and general way than copying and pasting; recall the rule of avoiding copy and paste more than twice. Writing a function has three big advantages over using copy and paste:

- You can give a function a name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making accidental mistakes when you copy and paste (e.g. updating a variable name in one place, but not in another).

# Writing a Function

A function can be defined with the following code:

```
func_name <- function(arguments) {  
  body  
}
```

The reserved word `function` is used to declare a function in R. The statements within the curly brackets form the `body` of the function. As with loops, these brackets are optional if the body contains only a single statement. Also, a function has inputs that are termed as `arguments`. Finally, the function above is given the name `func_name`.

As an example let us create a function called `raise()`. It takes two arguments, computes the first argument raised to the power specified in the second argument, and prints the result accompanied with a few words:

```
raise <- function(x, y) {  
  # prints x to the power of y  
  result <- x^y  
  print(paste(x, "to the power of", y, "is", result))  
}
```

The above uses the base R function `paste()` to concatenate string variables. It is good practice to check the function after writing it by calling it a few times using different arguments, and testing that the output is as expected.

```
raise(8, 2)  
[1] "8 to the power of 2 is 64"  
8^2  
[1] 64  
raise(2, 8)  
[1] "2 to the power of 8 is 256"  
2^8  
[1] 256
```

## Default Values for Arguments

It is also possible to assign default values to the arguments in a R function. For example, in the previous function we can set the default value for `y` to be `2`.

```
raise <- function(x, y = 2) {  
  # prints x to the power y  
  result <- x^y  
  print(paste(x, "to the power", y, "is", result))  
}
```

```
raise(3)
[1] "3 to the power 2 is 9"
raise(3, 1)
[1] "3 to the power 1 is 3"
```

Note that the default value of `y` is used only when no information for `y` is provided.

## Composing and Nesting Functions

Now let us look at another function that converts from centimetres to inches, `cm_to_inches`:

```
cm_to_inches <- function(cm) {
  inches <- cm/2.54
  return(inches)
}

# check that the function gives 1 inch for 2.54 centimetres
cm_to_inches(2.54)
[1] 1
```

Note the in R, it is not necessary to include the `return` statement. R automatically returns whichever variable is on the last line of the body of the function.

In a similar manner to getting inches from centimetres, we can transform inches into yards:

```
inches_to_yards <- function(inches) {
  yards <- inches/36
  yards
}

# check that the function gives 1 yard for 36 centimetres
inches_to_yards(36)
[1] 1
```

What about converting centimetres to yards? In addition to the previous approach, we can also do it by *composing* the two functions we have already created:

```
cm_to_yards <- function(cm) {
  inches <- cm_to_inches(cm)
  yards <- inches_to_yards(inches)
  yards
}

# check that 91.44 centimetres correspond to 1 yard
cm_to_yards(91.44)
[1] 1
```

The above can also be achieved with a single line of code by *nesting* functions:

```
inches_to_yards(cm_to_inches(91.44))  
[1] 1
```

This is a demonstration of how larger programs are built: basic operations are defined and then combined.

## Naming Functions

As with variables it is important to remember that functions are not only read by computers but also by humans; hence their name or the comments they contain are important. Below are some good practice recommendations:

- Aim for short function names as long as they are informative of what the function does. If this is hard being informative is more important than being short, as IDEs such as RStudio typically offer autocomplete, thus making it easy to type long names.
- A frequently used convention is to use verbs for function names and nouns for their arguments although there are exceptions, e.g. `mean()` is more convenient than `compute_mean()`.

## Useful Links and Resources

- [Chapter 19 from “R for data science”](#)
- [Chapter 21 from “R for data science”](#)
- [Chapter 5 from “Advanced R”](#)
- [Chapter 6 from “Advanced R”](#)

## References

- Guttag, J. V. (2013). *Introduction to computation and programming using python*. MIT Press.
- Wickham, H. (2019). *Advanced R*. Taylor & Francis Inc. CRC Press Inc.
- Wickham, H., & Grolemund, G. (2017). *R for data science: Import, tidy, transform, visualize, and model data* (1st ed.). O'Reilly Media.

# Exceptions, Error Handling and Debugging in R

## Outline

This page focuses on what happens when there are errors in our code. It has adapted material from chapters 8 and 22 of Wickham (2019). The aim is to present actions for addressing the errors and the tools that are available to help. We will cover ways to fix unanticipated problems (*debugging*), show how functions can be designed to communicate problems and what actions can be taken based on those communications (*condition handling*). Finally, we will go over strategies to avoid common problems before they occur (*defensive programming*). We will illustrate concepts in R; the material of next week will focus on how to perform these actions in Python.

Debugging is the task of fixing unexpected problems in your code. Although there are some guidelines on performing it, it is usually considered an art. Of course, not all errors are unexpected. Some common problems, for example, include a non-existent file or the wrong type of input. Using *conditions*, such as *errors*, *warnings*, and *messages*, when writing functions offers a great help:

- Errors are raised by `stop()` and force all execution in a function to terminate. They are used when there is no way for a function to continue.
- Warnings are generated by `warning()` and are used to display potential problems, e.g. `log(-1)`.
- Messages are generated by `message()` and are used to give informative output in a way that can easily be suppressed by the user (see `?suppressMessages`).

Condition handling tools, like `withCallingHandlers()`, `tryCatch()`, and `try()` allow users to take specific actions when a condition occurs. For example, when many models are being fitted, it may be desirable to continue fitting the others even if one fails to converge.

We conclude with a discussion of “defensive” programming and suggest strategies to avoid common errors before they occur. Quite often this involves spending a bit more time when writing the code but this pays off in the long run by reducing debugging time substantially. The basic principle of defensive programming is to raise an error as soon as something goes wrong. This takes three particular forms: checking that inputs are correct, avoiding non-standard evaluation, and avoiding functions that can return different types of output.

# Debugging Techniques

Debugging code is challenging. Most bugs are subtle and hard to find otherwise we would probably have avoided them in the first place. Here, we discuss some useful tools, provided by R and RStudio, and outline a general procedure for debugging. While the procedure below is by no means foolproof, it is helpful in organizing your thoughts when debugging. There are five steps:

1. **Realize that you have a bug:** You cannot fix a bug until you know it exists. Hence, it is always a good idea to test your code before you finish. Automated testing is a more involved option in that direction.
2. **Make it repeatable:** Once you have determined you have a bug in your code, you need to be able to reproduce it on demand. Without this, it becomes extremely difficult to isolate its cause and to confirm that you have successfully fixed it. Generally, you will start with a big block of code that you know causes the error and then slowly whittle it down to get to the smallest possible snippet that causes the error. Binary search is particularly useful for this. To do a binary search, you repeatedly remove half of the code until you find the bug. This is fast because, with each step, you reduce the amount of code to look through by half. As you work on creating a minimal example, you will also discover similar inputs that do not trigger the bug. Make sure you take note of them as they will be helpful when diagnosing the cause of the bug.
3. **Figure out where it is:** If you are lucky, one of the tools in the following section will help you to quickly identify the line of code that is causing the bug. Usually, however, you will have to think a bit more about the problem. It is a good idea to adopt the scientific method. Generate hypotheses, design experiments to test them, and record your results. This may seem like a lot of work, but a systematic approach will end up saving you time.
4. **Fix it and test it:** Once you have found the bug, you need to figure out how to fix it and to check that the fix actually worked. Again, it is very useful to have automated tests in place. Not only does this help to ensure that you have actually fixed the bug, it also helps to ensure you have not introduced any new bugs in the process. In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.
5. **Internet search:** In some cases, it may be a good idea to do an internet search on the error message. There is a chance that this is a common error with a known solution.

# Debugging Tools

There are three key debugging tools in R:

- RStudio’s error inspector and `traceback()`, which list the sequence of calls that lead to the error.
- RStudio’s “Rerun with Debug” tool and `options(error = browser)`, which open an interactive session where the error occurred.
- RStudio’s breakpoints and the `browser()` function, which open an interactive session at an arbitrary location in the code.

## Determining the Sequence of Calls

The first tool is the call stack, the sequence of calls that lead up to an error. Let us look at the following example: you can see that `f()` calls `g()` calls `h()` calls `i()`, which adds together a number and a string creating a error:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

When we run this code in RStudio we see:

```
> f(10)
```

Error in "a" + d : non-numeric argument to binary operator

[Show Traceback](#)

[Rerun with Debug](#)

Two options appear to the right of the error message: “Show Traceback” and “Rerun with Debug.” If you click “Show Traceback” you see:

```
> f(10)
```

Error in "a" + d : non-numeric argument to binary operator

[Hide Traceback](#)

[Rerun with Debug](#)

4. i(c)  
3. h(b)  
2. g(a)  
1. f(10)

Sometimes this is enough information to let you track down the error and fix it. However, sometimes it is not.  `traceback()` shows where the error occurred, but not why. The next useful tool is the interactive debugger, which allows you to pause execution of a function and interactively explore its state.

## Browsing on Error

The easiest way to enter the interactive debugger is through RStudio’s “Rerun with Debug” tool. This reruns the command that created the error, pausing execution where the error occurred. You are now in an interactive state inside the function, and you can interact with any object defined there. You will see the corresponding code in the editor (with the statement that will be run next highlighted), objects in the current environment in the “Environment” pane, the call stack in a “Traceback” pane, and you can run arbitrary R code in the console. There are a few special commands you can use in debug mode. You can access them either with the RStudio toolbar or with the keyboard:

- *Next*: Executes the next step in the function. Be careful if you have a variable named `n`; to print it you will need to do `print(n)`.
- *Step into*: Works like Next, but if the next step is a function, it will step into that function so you can work through each of that functions lines.
- *Finish*: Finishes execution of the current loop or function.
- *Continue*: Leaves interactive debugging and continues regular execution of the function. This is useful if you have fixed the bad state and want to check that the function proceeds correctly.
- *Stop*: Stops debugging, terminates the function, and returns to the global workspace. Use this once you have figured out where the problem is, and you are ready to fix it and reload the code.

As well as entering an interactive console on error, you can enter it at an arbitrary code location by using either an RStudio breakpoint or `browser()`.

## Condition (Error) Handling

Unexpected errors require interactive debugging to figure out what went wrong. Some errors, however, are expected, and you want to handle them automatically and in some cases ignore them. For example, when you are fitting a model to many datasets you may prefer to try and fit it into as many datasets as possible, even if some fits fail, and perform diagnostics in the end.

In R, there are three tools for handling conditions (including errors):

- `try()` gives you the ability to continue execution even when an error occurs.
- `tryCatch()` lets you specify handler functions that control what happens when a condition is signalled.
- `withCallingHandlers()` is a variant of `tryCatch()` that establishes local handlers, whereas `tryCatch()` registers exiting handlers.

## Ignoring Errors with `try()`

`try()` allows execution to continue even after an error has occurred. For example, normally if you run a function that throws an error, it terminates immediately and does not return a value. The below code chunk

```
f1 <- function(x) {  
  log(x)  
  10  
}  
f1("x")
```

stops with the error message “Error in `log(x)`: non-numeric argument to mathematical function.” However, if you wrap the statement that creates the error in `try()`, the error message will be printed but execution will continue:

```
f2 <- function(x) {  
  try(log(x))  
  10  
}  
f2("a")  
Error in log(x) : non-numeric argument to mathematical function  
[1] 10
```

## Handling Conditions with `tryCatch()`

`tryCatch()` is a general tool for handling conditions: in addition to errors, you can take different actions for warnings, messages, and interrupts. With `tryCatch()` you map conditions to handlers, which are named functions that are called with the condition as an input. If a condition is signalled, `tryCatch()` will call the first handler whose name matches one of the classes of the condition. A handler function can do anything, but typically it will either return a value or create a more informative error message. For example, the `show_condition()` function below sets up handlers that return the type of condition signalled:

```
show_condition <- function(code) {  
  tryCatch(code,  
    error = function(c) "error",  
    warning = function(c) "warning",  
    message = function(c) "message"  
  )  
}  
show_condition(stop("!"))  
[1] "error"  
show_condition(warning("?!"))  
[1] "warning"  
show_condition(message("?"))  
[1] "message"  
# If no condition is captured, tryCatch returns the  
# value of the input  
show_condition(10)  
[1] 10
```

As an example, consider a simple function that checks if an argument is an even number. You might write the following

```
is_even <- function(n) {
```

```

n %% 2 == 0
}
is_even(768)
is_even("two")

```

You can see that providing a string causes this function to raise an error. You could imagine though that you want to use this function across a list of different data types, and you only want to know which elements of that list are even numbers. You might think to write the following:

```

is_even_error <- function(n) {
  tryCatch(n %% 2 == 0,
           error = function(e) {
             FALSE
           })
}
is_even_error(714)
[1] TRUE
is_even_error("eight")
[1] FALSE

```

## Defensive Programming

Defensive programming is the strategy of making code fail in a well-defined manner even when something unexpected occurs. A key principle of defensive programming is to signal an error as soon as something wrong is discovered. In R, the “fail fast” principle is implemented in three ways:

- Be strict about what you accept. For example, if your function is not vectorized in its inputs, but uses functions that are, make sure to check that the inputs are scalars.
- Avoid functions that use non-standard evaluation, like `subset`, `transform`, and `with`. These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages.
- Avoid functions that return different types of output depending on their input. The two biggest offenders are `[` and `sapply()`.

## Useful Links and Resources

- [A prototype of a condition system for R](#): by Robert Gentleman and Luke Tierney
- [Beyond exception handling: conditions and restarts](#): by Peter Seibel, translated from Lisp to R
- [Automated Testing](#)

## References

Wickham, H. (2019). *Advanced R*. Taylor & Francis Inc. CRC Press Inc.

# ST2195 Programming for Data Science

## Block 5 Variables, Mutability and Aliasing in Python and R

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:  
<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-5>



\*\* Note: The code chunks below should be run in the following order \*\*

### Outline

In this page we focus on presenting some programming concepts in Python. Before doing this, we first return to the definitions of data types we have learned previously in the notes on Python data structures, and describe their association with objects. We then talk about mutability and aliasing in Python. Finally, we compare how objects in Python and R behave differently.

### Object and Type

You can consider everything in Python as an *object*. Each object has a *type* and the type dictates the behaviour of the object. We have seen in the page “Python and Data Structures” about different data type and structure, and here we revisit what we have learned.

The table below lists some of the built-in types (i.e. the types you can use straight away) in Python:

Type	Example
int	1, 123
float	3.14, 1.0
bool	True, False
None	None
string	"hello", "1"

Note that 1 is int but 1.0 is float and "1" is string.

# Type Checking and Casting

We can check the type of an object by the function `type()`. For example

```
type(1)
<class 'int'>
type(1.0)
<class 'float'>
type("1")
<class 'str'>
```

We can change the type of an object to another type by casting. For example:

```
type("1")
<class 'str'>
type(int("1"))
<class 'int'>
```

The first object `"1"` has the type `string` whereas the second object `int("1")` has the type `int` as the function `int()` converts the argument value to an integer (i.e. a object with type `int`).

What do you expect the outcome of the following code?

```
int("a")
```

## Relations Between Type and the Behaviour of the Objects

What we can do on objects depends on their type. For example, when we use the `+` operator on two objects with numeric type (`int` here in the example below):

```
1 + 2
3
```

we get another object with `int` type and value `3`, which is the sum of `1` and `2`.

If we do:

```
1 + "a"
```

we will get the error message: `TypeError: unsupported operand type(s) for +: 'int' and 'str'`, as we are not allow to “add” `int` and `string` together. The type of the objects decides what you can do and what you cannot do on them.

The type of an object also decides the behaviour of the operation. If we apply the summation operator on two `strings`:

```
"hello"+"world"
'helloworld'
```

we are not “summing” the `string`s in the numerical sense but we *concatenate* the `string`s to get a new `string`. The type therefore dictates the behaviour of the objects.

What if we run the same code in R?

```
## R code  
"hello"+"world"
```

Does it work?

## Variables and Assignment

In the above examples we created an object with type `int` and value `3` by the expression `1+2`. We also created another object with type `string helloworld` by concatenating two `string`s together. However, these objects are created and lost. We are not able to use them again (e.g. for further calculation). In order to be able to use them later, we need to assign them to *variables*. We can do it by using the assignment operator `=`. For example:

```
pi = 3.14
```

which we assign the object with type `float` and value `3.14` to the variable with name `pi`. So now we can retrieve the value by using the variable `pi`:

```
pi  
3.14
```

The variable will have the same type as the object that assigned to it:

```
print(type(pi))  
<class 'float'>  
print(type(3.14))  
<class 'float'>
```

Similar to R, we can assign value to a variable using an expression, see the third line of the code:

```
pi = 3.14  
radius = 11  
area = pi * (radius**2)  
area  
379.94
```

We can rebind the variable to another object by simply assigning another object to it. For example:

```
pi = "hello"  
pi  
'hello'
```

Now `pi` is bound to the object with value `"hello"`.

The type of a variable depends on the object assigned to it. Now the type of `pi` is `string`, as it binds to the object `"hello"`.

```
type(pi)  
<class 'str'>
```

Note that this assignment has no effect on the value of the variable `area`, which remains as  $3*(11**2) = 379.94$

```
area  
379.94
```

## Naming a Variable

Similar to R, variable names can contain upper-case and lower-case letters, digits (but they cannot start with a digit) and underline characters, and variable names are case-sensitive. Unlike R, and you **cannot** use dots as part of the variable name. We will see later that dot is used to get attributes from an object.

In Python, some of the words are reserved for specific purposes (known as *keywords*) and you are not allowed to use them as variable names. Example of the keywords are `if`, `else`, `for`, `def`, etc. If you run the following:

```
if = 3
```

you will get an error with the error message: `SyntaxError: invalid syntax`.

## Mutability

In the page “Python and Data Structures” we have seen that we can modify objects like `list`. For objects that can be modified, we say they are *mutable*. Examples of type of mutable objects are `list` and `dict`.

For example, we can modify a `list`:

```
colours = ["red", "blue", "green"]  
colours[0] = "orange"  
colours  
['orange', 'blue', 'green']
```

Remember in Python the indices start from `0` (in R indices start from `1`).

On the other hand, we cannot modify *immutable* objects like `tuple` or `string`. For example if we try to modify an element in a `tuple`:

```
colours = ("red", "blue", "green")  
colours[0] = "orange"
```

It will give the error message `TypeError: 'tuple' object does not support item assignment`.

# Aliasing

Consider the following code:

```
colours = ["red", "blue", "green"]
colours_2 = colours
colours_2[0] = "orange"
print(colours_2)
['orange', 'blue', 'green']
print(colours)
['orange', 'blue', 'green']
```

While you probably expected `colours_2` has "orange" as the first element, you probably did not expect that `colours` has "orange" as the first element too! What is wrong? After all if we write similar code in R:

```
# R code
r_colours <- c("red", "blue", "green")
r_colours_2 <- r_colours
r_colours_2[1] <- "orange"
print(r_colours_2)
print(r_colours)
[1] "orange" "blue"   "green"
[1] "red"    "blue"   "green"
```

`r_colours` would not be changed! Why similar blocks of code behave differently in R and Python?

To understand further, it is worth taking a closer look what assignment of an object to a variable is actually doing behind the scene in Python. Below we use [Python Tutor](#) to visualise how the first 3 lines of the Python code bind the object (the `list`) with variable names `colours` and `colours_2`, and change the object through `colours_2`. Please press the `next` button to see what actually is happening for each line of the code.

```
colours = ["red", "blue", "green"]
colours_2 = colours
colours_2[0] = "orange"
```

As we can see in the visualisation above, when we run `colours_2 = colours`, what we are copying is the `reference` to the `list` but not the `list` itself. In other words, what we are doing is assigning the `list` that the variable `colours` referred to to `colours_2`. Effectively now the variables `colours` and `colours_2` are binding to the same `list`. You can think `colours` and `colours_2` are just two names of the same object. The `list` itself is not copied. When we run the third line, we modify the `list` through the variable name `colours_2`. As the variable `colours` refers to the same `list` as `colours_2`, when we retrieve the `list` by the variable name `colours`, we will see the first element is "orange" instead of "red".

We can verify that that the variables `colours` and `colours_2` refer to the same object by using the function `id()`:

```
id(colours)
5441935496
id(colours_2)
5441935496
```

The function `id()` returns a unique identify of an object and we can see that `id(colours)` and `id(colours_2)` return the same value.

If we want to actually make a copy of the `list` (the object), we need to do it explicitly by using the `copy()` function:

```
colours_3 = colours.copy()
colours_3[0] = "black"
print(colours_3)
['black', 'blue', 'green']
print(colours)
['orange', 'blue', 'green']
```

When working on Python, remember that a container can be modified by any variables binding to them. If you want to make sure no other variables can modify the container, explicitly create a copy when assigning to another variable, as shown in the code block above.

## Should We Worry About Aliasing with Immutable Objects?

The short answer is no because *immutable* objects cannot be changed. Consider the following code:

```
pi = 3.14
print(pi)
3.14
pi = "hello"
print(pi)
hello
```

Are we not changing the object in the third line? While we indeed “update” the value associate with the variable `pi` by the second assignment, object created from the first assignment is not changed. What is actually happening in the second assignment is that a *new* object associated with the value `"hello"` is created and is bound to the variable `pi`. We can confirm this by printing the identity of the object bound to `pi`:

```
pi = 3.14
print(id(pi))
5441320784
pi = "hello"
```

```
print(id(pi))  
5441941320
```

We can see that the identity has changed, showing that `pi` on line 1 and 3 are binding to different objects. We did not *change* the immutable object, instead we created a new immutable object.

Therefore, we do not worry about the aliasing issue, as shown in the example below:

```
pi = 3.14  
a = pi  
print(id(pi))  
5441320904  
print(id(a))  
5441320904  
pi = "hello"  
print(a)  
3.14  
print(id(pi))  
5441416304  
print(id(a))  
5441320904
```

## Why Does the R Code Behave Differently?

The *short* answer is that R uses the *copy-on-modify* strategy. When you want to modify the object that a variable is pointing to, R will make a copy of the object, make the change and assign it to the variable. If there are other variables pointing to the same object, they will remain pointing to the old, unmodified object. In reality it is more complicated than this and we will explain further below.

If you find it difficult to understand even after reading the explanation above, just remember the following: in this course you do not need to worry about the aliasing issue in R. When you modify a variable in R, in this course you can *assume* other variables will not be modified.

## Is the R `vector` a Mutable Object?

We can modify `vector` in R in a similar way as `list` in Python:

```
# R code  
r_colours <- c("red", "blue", "green")  
r_colours[1] <- "orange"  
r_colours  
[1] "orange" "blue"    "green"
```

It feels like R `vector` is mutable as we can change its element.

# Does R Copy the Reference Like Python?

In R, when we assign `r_colours_2 <- r_colours`, similar to Python only the reference is copied. We can verify it using the function `tracemem()`. `tracemem()` shows you the memory that the variable is pointing to, and will give a message when the corresponding object is copied.

```
# R code
r_colours <- c("red", "blue", "green")
tracemem(r_colours)
r_colours_2 <- r_colours
tracemem(r_colours_2)
[1] "<0x7fbb2ee79008>"
[1] "<0x7fbb2ee79008>"
```

Note that no message is shown as no object is copied. Both `r_colours` and `r_colours_2` have the same memory address.

However, when we change `r_colours`,

```
# R code
r_colours[1] <- "orange"
tracemem[0x7fbb2ee79008 -> 0x7fbb2f304308]: eval eval withVisible withCallingHandle
rs handle timing_fn evaluate_call <Anonymous> evaluate in_dir block_exec call_block
process_group.block process_group withCallingHandlers process_file <Anonymous> <Anonymous>
```

We can see the message from the `tracemem()` function printing out `tracemem[old_address -> new_address]`, which tells us that `r_colours` is now pointing to a *new* object. When there are more than one variables pointing to an object and we want to modify one of the variables (e.g. `r_colours[1] <- "orange"`), R will make a copy of the object, make a change and then assign to the variable which trigger the change. For other objects, they are still pointing to the old object.

We can verify that `r_colours` is pointing to a new object and `r_colours_2` is pointing to the old object by the `tracemem()` function:

```
# R code
tracemem(r_colours)
tracemem(r_colours_2)
[1] "<0x7fbb2f304308>"
[1] "<0x7fbb2ee79008>"
```

Note that the address for `r_colours` has changed, but the address for `r_colours_2` has not changed. It is still pointing the old object.

Therefore, the first element of `r_colours` has changed to "orange" but not `r_colours_2`:

```
# R code
print(r_colours)
print(r_colours_2)
[1] "orange" "blue"   "green"
[1] "red"    "blue"   "green"
```

## Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Section 2.1.2.
- [Python tutor: A tool to visualise your Python code](#)
- [Official Python tutorial on list](#)

# Control Flow Structures in Python



\*\* Note: The code chunks below should be run in the following order \*\*

## Conditional Statements

We start with the `if` statement. Its basic form follows:

```
if condition:  
    true_action_1  
    true_action_2  
    ...  
  
if condition:  
    true_action_1  
    true_action_2  
    ...  
else:  
    false_action_1  
    false_action_2  
    ...
```

For example:

```
mark = 80  
if mark >= 50:  
    print("pass")  
pass
```

While the control statements between R and Python are quite similar, there are several differences between them:

1. No parentheses are needed to enclose the condition
2. `:` is needed after the `if` and `else` keywords
3. **Indentation is required for the block after the condition statement.** You have to indent following the way demonstrated above. While any number of spaces will do as long as the spacing is consistent, the standard is to use four spaces for one level indentation. The following code will return error with error message `IndentationError: expected an indented block`:

```
mark = 80
if mark >= 50:
    print("pass")
```

## More on Indentation

In Python, each line of code in the the block must be indented by the same amount. This is in contrast to many other languages (like R) where indentation is optional, and curly brackets {} are used to define blocks.

In Python, different indentation may provide different outcome, and some indentation may result in errors. For example:

```
mark = 30
if mark >= 50:
    print("pass")
    print("congrats!")
```

and

```
mark = 30
if mark >= 50:
    print("pass")
print("congrats!")
congrats!
```

provide different results. Why is this the case? Note that the line `print("congrats!")` is not part of the control flow in the second example. Therefore, it will be executed no matter whether the condition `mark >= 50` is satisfied.

The code below will trigger the error message `IndentationError: unexpected indent`, as the `if` statement should not be indented.

```
mark = 80
    if mark >= 50:
        print("pass")
```

Similarly, the code below results in the same error message, as the indentation should be the same for both print statements.

```
if mark >= 50:
    print("pass")
    print("congrats")
```

The code below results in the same error message as well, as the `if` and the `else` statement should be at the same level.

```
if mark >= 50:  
    print("pass")  
else:  
    print("fail")
```

Again, the code below results in the same error message. Why is it the case?

```
if mark >= 50:  
    print("pass")  
print("congrats!")  
else:  
    print("fail")
```

## Indentation in R vs Python

In R the use of indentation is optional. You have the freedom to choose whether to indent and how to indent your code. Often indentation is used in R to make the code easier to be read. If we rewrite the examples which result in error above in R syntax they will run fine in R:

```
## R code  
mark = 80  
if (mark >= 50)  
print("pass")  
[1] "pass"  
## R code  
if (mark >= 50)  
print("pass")  
[1] "pass"  
## R code  
if (mark >= 50) {  
print("pass")  
    print("congrats")  
}  
[1] "pass"  
[1] "congrats"
```

While the above R code is valid, it is not recommended to write the code with non-standard indentation as it makes it more difficult to understand the code and likely to cause confusion.

## Nested `if`

As in R, we can nest an `if` statement within an `if` statement. In Python we have the `elif` keyword which is short for 'else if'. The following two codes will give the same result:

```
mark = 65
if mark >= 70:
    print("distinction")
elif mark >= 60:
    print("merit")
else:
    print("pass")
merit
if mark >= 70:
    print("distinction")
else:
    if mark >= 60:
        print("merit")
    else:
        print("pass")
merit
```

# Iterations

There are two main types of iterations in Python, the `for` and `while` loops.

## `for` Loop

The `for` loops are used to iterate a collection of objects. They have the following basic form:

```
for item in container:
    perform_action
```

Can you see how the syntax in Python is different from R?

For each `item` in a `container`, `perform_action` is called once. For example the following code prints all the numbers between `1` and `5`:

```
for i in [1,2,3,4,5]:  
    print(i)  
1  
2  
3  
4  
5
```

Similar to R, it is possible to also terminate a `for` loop early. There are two ways to do it:

- `continue` exits the current iteration.
- `break` exits the entire for loop.

```
for i in [1,2,3,4,5]:  
    if i < 2:  
        continue  
        # for i < 2 the code below will not be executed  
    print(i)  
    if i >= 4:  
        break # the loop stops here  
2  
3  
4
```

## while Loop

The `for` loops are useful if you know in advance the set of values that you want to iterate over. If you don't know, you can use `while` loop:

```
while condition:  
    action
```

For example, it is more appropriate to use `while` loop for the following example:

```
word = input("Give me a 4-letter word:")  
while len(word) != 4:  
    print("Wrong input!")  
    word = input("Give me a 4-letter word:")  
print(word)
```

As we do not know how many times we need to wait until the user gives us a required input.

# Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Section 3.2.
- [Official Python tutorial on control flow](#)

# Function and Scope in Python

\*\* Note: The code chunks below should be run in the following order \*\*

## Writing a Function

A function can be defined with the following code:

```
def func_nam(arg1, arg2, arg3):
    """
    explanation of the function
    ...
    line_1
    line_2
    ...
    return value
```

The keyword `def` tells Python that the following block of code defines a function. Again indentation is important. The explanation in the beginning of the block tells users what the function does. The `return` statement can be omitted if the function does not return any value.

```
def circle_area(radius):
    """
    This function calculates the area of a cirle given the radius
    input: radius
    output: area = pi * radius ^2
    ...
    pi = 3.14
    return (radius**2)*pi
```

We can call the function by using the name of the function with `()`:

```
circle_area(2)
12.56
```

In contrast to R, if you want to return a value from a function, you have to use the `return`. If you write the function `circle_area` in the following way:

```
def circle_area(radius):
    """
    This function calculates the area of a circle given the radius
    input: radius
    output: area = pi * radius ^2
    """
    pi = 3.14
    (radius**2)*pi
```

When you call the function nothing is returned or, in other words, the value `None` is returned.

```
circle_area(2)
```

## Scope

In Python, variables are only available within the corresponding region it is created. Such region is called a *scope*.

### Global and Local Scope

A variable created within the main body in Python is in the *global scope*. Their values are available anywhere. Variables created in the function belong to the *local scope*. They are only available within the corresponding function but not elsewhere. Note the local scope is created when the function is *called* but not when it is defined.

### Variables From the Global Scope is Available Everywhere

Consider the following code (Example 1):

```
# Example 1
def add_a(x):
    x = x+a
    return x

a = 8
z = add_a(10)
print(z)
18
```

Does the following code work, despite not defining `a` in the function `add_a`? Yes it works because `a` is global and can be used anywhere (including inside a function).

Press `next` to visualise the code and understand what is happening when we execute each line of the code.

As we can see, we first have the function `add_a()` defined in the *global frame*. By running the fifth line we have `a = 8` in the global frame. When we call the function `add_a()` in the sixth line, we created a *local frame add\_a*. Note that it is created when the function `add_a()` is *called* but *not* when it is defined. The object `x` is in this local frame with value `10` and it then is updated to `18` after executing the line `x = x + a`. `a` is available within the function as `a` is global so it is available everywhere. The function `add_a` hits the return statement with the return value `18`. The return value is then assigned to `z` in the global frame and the local frame `add_a` is gone.

## Variables From the Local Scope is Not Available Elsewhere

Consider the following code (Example 2):

```
# Example 2

def add_b(x):
    b = 8
    x = x + b
    return x

z = add_b(10)
print(b)
```

If you run the code above, you will get the error `NameError: name 'b' is not defined`. Why is it the case? Press `next` to visualise the code and understand what is happening when we execute each line of the code.

Like the previous example, we first have the function `add_b()` defined in the *global frame*. When we call the function `add_b()` in the sixth line, we created a *local frame add\_b*. This time the object `b` is in this *local frame* with value `8`. The function `add_b` hits the return statement. The local frame `add_b` is gone and so is the variable `b`. Therefore, we get an error when we try to `print(b)` in the main body.

## LEGB Rule

Previous, we saw that if we are trying to get a variable in the local frame but it is not defined there, the global one will be used if it is available. What if the same variable name is available for the local and global scope? Consider the following code (Example 3):

```
# Example 3

def add_1(x):
    x = x + 1
    return x

x = 10
z = add_1(12)
print(z)
13
```

What is the value of `z`? What is the value of `x`? Let us look at the visualisation of the above code. Please press [next](#) to see how each line works behind the scene.

As we can see, we first have the function `add_1` defined in the *global frame*. By running the fifth line we have `x = 10` in the global frame. When we call the function `add_1()` in the sixth line, we first created a *local frame* `add_1`. The `x` in this local frame with value `12` is different from the `x` with value `10` in the global frame. They are two different objects. Therefore when we do `x = x+1` in the function `add_1`, we see that `x` in the `add_1` frame has changed to `13` but `x` in the global frame stays as `10`. The function `add_1` hits the return statement with the return value `13`. The return value is then assigned to `z` in the global frame and the local frame `add_1` (together with the local `x`) is gone.

The *LEGB rule* can be thought of as a name look up procedure. LEGB stands for:

- L: Local scope
- E: Enclosing scope, this exists for nested functions. The outer function is the enclosing scope for the inner function.
- G: Global scope
- B: Built-in scope, a scope that is available and automatically loaded whenever you run Python. It contains names of built-in functions, keywords, exceptions, etc. Examples are:
  - Keywords: `if`, `def`, etc.
  - Functions: `print`, `len`, etc.

Python code will find the name sequentially in the local scope ("L"), the enclosing scope ("E"), the global scope ("G"), the built-in scope ("B") based on where the executed line is in. If nothing can be found after searching all these four scopes, Python will throw an error. We can use the LEGB rule to explain the Examples 1 to 3 above:

- Example 1: We search for the variable `a` when we are in the local scope. As the variable name `a` is only available in the global scope, the `a` in the global scope is used.
- Example 2: We search for the variable `b` when we are in global scope. As the variable name `b` is not available (`b` was in the local scope but it is not available in global scope and was not retrievable after the function has returned), Python cannot find the variable `b`.
- Example 3: The variable `x` is available in both local and global scope. Following the LEGB rule, when running the line `x = x + 1`, we first search it in the local scope for `x`. As `x` is found in the local scope, the value of `x` in the global scope will not be used inside the function.

In R, a similar rule applies. See the same examples above implemented in R:

```
# Example 1
add_a <- function(x) {
  x <- x+a
  return (x)
}

a <- 8
z <- add_a(10)
print(z)
[1] 18

# Example 2
add_b <- function(x) {
  b <- 8
  x <- x + b
  return (x)
}

z <- add_b(10)
print(b)
Error in print(b): object 'b' not found

# Example 3
add_1 <- function(x) {
  x <- x + 1
  return (x)
}

x <- 10
z <- add_1(12)
print(z)
[1] 13
```

## Functions With Mutable Object

Consider the following code:

```
def add_one_number(seq, num):
    seq.append(num)
    return (seq)

nums = [1,3,2,4]
new_nums = add_one_number(nums, 5)
print(new_nums)
[1, 3, 2, 4, 5]
print(nums)
[1, 3, 2, 4, 5]
```

Given what you have learned about mutable objects, aliasing and scoping, are you surprised by the result? If you are confused, take a look of the following visualisation:

When dealing with mutable objects in a function, you need to be careful if there are any unintentional side-effects raised. For the example above, if you do not wish to change the original `list`, the function should be written as:

```
def add_one_number(seq, num):
    new_seq = seq.copy()
    new_seq.append(num)
    return (new_seq)

nums = [1,3,2,4]
new_nums = add_one_number(nums, 5)
print(new_nums)
[1, 3, 2, 4, 5]
print(nums)
[1, 3, 2, 4]
```

## Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Chapter 4.
- [Python Tutor: A tool to visualise your Python code](#)
- [An official Python tutorial on functions](#)

# Exceptions and Error Handling in Python

\*\* Note: The code chunks below should be run in the following order \*\*

## Exceptions in Python

Until now, you probably have seen different error messages when you write a Python program. Reading the error messages is often the first step to figuring out what is wrong in your code. There are two main types of error:

- **SyntaxError**: When your syntax is not correct and Python cannot parse your program. **IndentationError** is one example of **SyntaxError**. It occurs when you do not indent blocks of code appropriately.
- **Exceptions**: Something else is wrong even your syntax is correct. There are many different types of exceptions and below we list some of the common ones:
  - **NameError**: Attempt to retrieve a variable that is not defined in the local or global scope.
  - **TypeError**: Providing arguments with some inappropriate types.
  - **ValueError**: Type of the argument is correct but the value is not.
  - **IndexError**: When you try to access an element with the index that is out of range.

There are many more types of exceptions, and you can learn more from [here](#).

The following blocks of code trigger different error messages in Python:

```
# SyntaxError  
print "hello"
```

which gives the error message **SyntaxError: Missing parentheses in call to 'print'.**  
**Did you mean print("hello")?**

```
# IndentationError  
def print_hello():  
    print("hello")
```

which gives the error message **IndentationError: expected an indented block.**

```
# NameError  
hello  
Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'hello'  
is not defined  
  
Detailed traceback:
```

```

  File "<string>", line 1, in <module>
    # TypeError
    1+'2'

Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported o
perand type(s) for +: 'int' and 'str'

Detailed traceback:
  File "<string>", line 1, in <module>
    # ValueError
    int('a')

Error in py_call_impl(callable, dots$args, dots$keywords): ValueError: invalid lite
ral for int() with base 10: 'a'

Detailed traceback:
  File "<string>", line 1, in <module>
    # IndexError
    num = [1,2,3]
    num[3]

Error in py_call_impl(callable, dots$args, dots$keywords): IndexError: list index o
ut of range

Detailed traceback:
  File "<string>", line 1, in <module>

```

## Handling Exceptions

Until now, whenever we encounter an exception, we let the program crash and we go back to the code to try out the problem and to fix it. It is not ideal and for some situations, it is not even a feasible strategy. For example, consider the following example:

```

nomin_str = input("Enter an integer: ")
nomin = int(nomin_str)
denom_str = input("Enter another integer: ")
denom = int(denom_str)
num = nomin / denom
print(f"Dividing {nomin} by {denom} gives {num}.")

```

As the input is from the users, we cannot guarantee that they will always provide a valid input. If they accidentally give "a" as the input, then you will get the error `ValueError: invalid literal for int() with base 10: 'a'` and the program will crash. Even if numbers are provided as the input, if the second number provided is 0, then we will get a `ZeroDivisionError`.

## Using `try-except` to Handle Exceptions

A better approach is to handle the exception by the `try` and `except` blocks:

```
try:  
    nomin_str = input("Enter a number: ")  
    nomin = int(nomin_str)  
    denom_str = input("Enter another number: ")  
    denom = int(denom_str)  
    num = nomin / denom  
    print(f"Dividing {nomin} by {denom} gives {num}.")  
except:  
    print("Wrong input.")
```

If an exception is raised in the `try` clause during execution, the rest of the `try` clause will be skipped. The exception will be handled by the `except` statement and the `except` clause will be executed. Now the program does not crash even if the user provides a wrong input, only the printout "Wrong input." is given.

We can use separate `except` clauses to handle different types of exceptions:

```
try:  
    nomin_str = input("Enter an integer: ")  
    nomin = int(nomin_str)  
    denom_str = input("Enter another integer: ")  
    denom = int(denom_str)  
    num = nomin / denom  
    print(f"Dividing {nomin} by {denom} gives {num}.")  
except ValueError:  
    print("Input provided cannot be converted into a number.")  
except ZeroDivisionError:  
    print("Zero cannot be the denominator.")  
except:  
    print("Some other error.")
```

The `ValueError` handles the case when users provide non-numerical input.

`ZeroDivisionError` handles when the second input is zero. The last `except` handles all other possible exceptions. Try to run the code above with different inputs, and see what printout you will get.

We can also use the following with the `try` and `except` clauses:

- `else`: The clause will be executed if *no* exception is raised
- `finally`: The clause will *always* be executed no matter there is an exception or not. It will be executed even if there is a `break`, `continue` or `return`.

Try to run the following code and test your understanding on how `except`, `else` and `finally` work.

```
count = 0

while True:
    try:
        nomin_str = input("Enter an integer: ")
        nomin = int(nomin_str)
        denom_str = input("Enter another integer: ")
        denom = int(denom_str)
        num = nomin / denom
    except ValueError:
        print("Input provided cannot be converted into an integer.")
    except ZeroDivisionError:
        print("Zero cannot be the denominator.")
    except:
        print("Some other error.")
    else:
        print(f"Dividing {nomin} by {denom} gives {num}.")
        break
    finally:
        count += 1
        print(f"Attempt {count}.")
```

## try-except or if-else Statement?

The code above can be rewritten by `if-else` instead of the `try-except`:

```
nomin_str = input("Enter an integer: ")
denom_str = input("Enter another integer: ")

if nomin_str.isdigit() and nomin_str.isdigit():
    nomin = int(nomin_str)
    denom = int(denom_str)

    if denom != 0:
        num = nomin / denom
        print(f"Dividing {nomin} by {denom} gives {num}.")
    else:
        print("Zero cannot be the denominator.")
else:
    print("Input provided cannot be converted into a number.")
```

Which is a better way? In Python, EAFP (easier to ask for forgiveness than permission) is the common coding style. We first do what we expect to do by assuming the input is fine, and if the assumption proves to be wrong we catch the exceptions by the `try-except` clauses. This contrasts to “LBYL” (look before you leap) style commonly used in many other languages (e.g. C), which try to avoid the exceptions (by the `if-else` statement) instead of catching them. Therefore in Python `try-except` is better to be used instead of `if-else` when something might be wrong. If you compare both codes, the `try-except` version is probably easier to be read.

## Raise Exceptions

Above we have shown that we can avoid raising exceptions by catching them. Sometimes we want to *raise* our own exceptions to give more precise information about what is wrong.

Consider the following code:

```
try:
    nomin_str = input("Enter an integer: ")
    nomin = int(nomin_str)
    denom_str = input("Enter another integer: ")
    denom = int(denom_str)
    num = nomin / denom
    print(f"Dividing {nomin} by {denom} gives {num}.")
except ValueError:
    raise ValueError("Input provided cannot be converted into a number.")
except ZeroDivisionError:
    raise ZeroDivisionError("Zero cannot be the denominator.")
except:
    raise ValueError("Wrong input.")
```

If you give "a" as the first input, you will get the error `ValueError: Input provided cannot be converted into a number.`. This error message gives additional information to the generic error message `ValueError: invalid literal for int() with base 10: 'a'`.

## Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Chapter 7.
- [Official Python tutorial on errors and exceptions](#)

# Debugging in Python

\*\* Note: The code chunks below should be run in the following order \*\*

## Debugging Tools in Python

Below are the main debugging tools in Python:

- Spyder
- [Python Tutor](#)
- Put `print()` into where you suspect there may be a bug in the program

Below we will illustrate how to use Python Tutor and function `print()` for debugging. For Spyder, we will show it in the live section. Let us consider the following code, with the function `get_sorted_numbers()` aims to return a sorted list. The list returned from the function does not seem to be sorted:

```
def get_sorted_numbers(nums):
    """
    input: a list of numbers
    output: a list of sorted numbers
    """
    sorted(nums)
    return nums

nums = [1,3,2,1]
sorted_num = get_sorted_numbers(nums)
print(sorted_num)
[1, 3, 2, 1]
```

What is wrong? Let us try to examine the code using Python Tutor:

We can see visually from Python Tutor that the function `sorted()` does not change the order of the list `num`. Such visualisation power is very helpful for us to understand each line of the code and see if any of the lines does not work as we have expected.

Next, we use the `print()` function to see if `nums` is actually sorted by the function `sorted()`. We put the `print()` function before and after the line `sorted(nums)` to see if the line has actually changed the list `nums`:

```

def get_sorted_numbers(nums):
    """
    input: a list of numbers
    output: a list of sorted numbers
    """

    print("before sort", nums)
    sorted_nums
    print("after sort", nums)
    return nums

nums = [1,3,2,1]
sorted_num = get_sorted_numbers(nums)
before sort [1, 3, 2, 1]
after sort [1, 3, 2, 1]
print(sorted_num)
[1, 3, 2, 1]

```

We can see that the list `nums` is the same before and after `sorted()`, showing us that something must be wrong at the line `sorted(nums)`. Now we can proceed and correct the problem using something like:

```

def get_sorted_numbers(nums):
    """
    input: a list of numbers
    output: a list of sorted numbers
    """

    return sorted(nums)

nums = [1,3,2,1]
sorted_num = get_sorted_numbers(nums)
print(sorted_num)
[1, 1, 2, 3]

```

## Assertions

Previously in “Exceptions, Error Handling and Debugging in R”, we talked about the “defensive programming” strategy. One of the ways to implement this strategy is to use `assert` statement to check if the program is running as expected. If it is not, `AssertionError` exception is raised. Assertions can be used to:

- Halt the program as soon as some unexpected conditions occur. This makes it easier to discover and locate the bugs.
- Check if the output is appropriate before returning the bad value and causing unexpected consequences.

The following function aims to sort the list of numbers:

```
def get_sorted_numbers(nums):
    sorted_nums
    assert(len(nums)<2 or all([nums[i] <= nums[i+1] for i in range(len(nums)-1)])), 
"The list " + str(nums) + " is not sorted"
    return nums
```

The `assert` statement check if the next number is at larger or equal to the previous number. If the condition is not satisfied, the `AssertionError` is raised and the error message `The list ... is not sorted` is shown. This tells us that there must be a bug in this function.

## Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Section 6.2 and Chapter 7.

# Classes and Programming Paradigms

\*\* Note: The code chunks below should be run in the following order \*\*

## Classes

In the previous sections, we have defined our own functions. Here we will define our own *type* of objects by creating a new *class*. Classes allow us to bundle data and functionality together (*encapsulation*).

## Problem-Solving Using Functions

Consider that we have data in the form of two dimension points, e.g. (4, 3), (1.1, 2.42) etc. We want to store the data and be able to do some calculations on them (e.g. calculate the distance between two points). One possibility is to use tuples to store the data points (x and y coordinate value) and functions to do calculation, as shown below:

```
def point_dist(point_1, point_2):
    """
    return the distance of the two given points.
    point_1 and point_2 should be in the form of (x1, y1), (x2, y2) with x1,x2,y1,y2 are numbers.
    """
    assert len(point_1) == 2 and len(point_2) == 2, "points must be 2 dimensional"
    try:
        return ((point_1[0] - point_2[0]) ** 2 + (point_1[1] - point_2[1]) ** 2) ** (1/2)
    except:
        raise ValueError("Inputs are not in the form of (x1, y1), (x2, y2) with x1, x2, y1, y2 are numbers.")

def dist_from_origin(point):
    """
    return the distance of a point from the origin.
    point should be in the form of (x, y) with x, y are numbers.
    """
    assert len(point) == 2, "points must be 2 dimensional"
    try:
        return (point[0] ** 2 + point[1] ** 2) ** (1/2)
    except:
```

```

        raise ValueError("Input is not in the form of (x, y), with x,y are numbers.
")

from numbers import Number as numeric

def is_same_point(point_1, point_2):
    """
    check if the given two points are the same.
    point_1 and point_2 should be in the form of (x1, y1), (x2, y2) with x1,x2,y1,y
    2 are numbers.
    """

    assert len(point_1) == 2 and len(point_2) == 2, "points must be 2 dimensional"
    assert all(isinstance(element, numeric) for element in point_1), "point_1 non-n
    umeric"
    assert all(isinstance(element, numeric) for element in point_2), "point_2 non-n
    umeric"

    try:
        return point_1[0] == point_2[0] and point_1[1] == point_2[1]
    except:
        raise ValueError("Inputs are not in the form of (x1, y1), (x2, y2) with x1,
        x2,y1,y2 are numbers.")


def mid_point(point_1, point_2):
    """
    return a mid point from the given two points.
    point_1 and point_2 should be in the form of (x1, y1), (x2, y2) with x1,x2,y1,y2
    are numbers.
    """

    assert len(point_1) == 2 and len(point_2) == 2, "points must be 2 dimensional"
    try:
        x = (point_1[0] + point_2[0]) / 2
        y = (point_1[1] + point_2[1]) / 2
        return (x, y)
    except:
        raise ValueError("Inputs are not in the form of (x1, y1), (x2, y2) with x1,
        x2,y1,y2 are numbers.")

```

The above code chunks only defined the functions required for the calculations. We have to *call* the functions to execute them, for example:

```

point = (3,4)
dist_from_origin(point)
5.0

```

With the implementation above, the calculations and data are separated. Also note that any other objects not in the form of a point can also use the functions, although it is likely that an error will be raised. This implementation is in the “style” of *functional programming*, for which we solve a problem by functions:

```
point_1->[dist_from_origin]-> distance
```

Also, the function does not have any side effect (e.g. does not mutate the input, modify the global variables or have any print out), and the output of the function only depends on the input.

This is very similar to how we solve the mathematics problem  $|y = f(x)|$ . For the example above, we have `x = (3, 4)` (a point), `f = dist_from_origin` (function to calculate the distance from the origin) and `y = 5` (distance).

## Problem-Solving Using Classes

Alternatively, we can define our own class to do achieve what we have done above. Have a look at the code below before we explain what it is doing.

```
class Point:  
    """  
        The Point class represents a point in 2 dimensions.  
    """  
  
    def __init__(self, x, y):  
        if isinstance(x, numeric) & isinstance(y, numeric):  
            self.x = x  
            self.y = y  
        else:  
            raise ValueError('x and y must be numbers')  
  
    def dist(self, pt):  
        """  
            return the distance between the point and a given point.  
        """  
        assert isinstance(pt, Point), "the argument must be a point"  
        return ((self.x - pt.x) ** 2 + (self.y - pt.y) ** 2) ** (1/2)  
  
    def dist_from_origin(self):  
        """  
            return the distance of the point from the origin.  
        """  
        return (self.x ** 2 + self.y ** 2) ** (1/2)  
  
    def mid_point(self, pt):  
        """
```

```

        return a mid point from the point and a given point.

'''

assert isinstance(pt, Point), "the argument must be a point"

x = (self.x + pt.x) / 2
y = (self.y + pt.y) / 2
return (x, y)

def is_same(self, pt):
    '''

    check if the point and the given point is the same.

    '''

    assert isinstance(pt, Point), "the argument must be a point"
    return self.x == pt.x and self.y == pt.y

```

The class `Point` defined above bundles the data (x, y coordinates) and the functionalities (calculate the distance between two points, find the midpoint, etc.) of points together:

- Data: `self.x` and `self.y` store the x and y coordinate information of a given point. The values are assigned in the `__init__()` method.
- Functionalities: four different methods are defined. The method `distance_from_origin`, for example, calculate the distance of the point from the origin, using the point data stored in `self.x` and `self.y`.

The code in the above code chunk is written in a *object-oriented programming* (OOP) style, for which we solve the problem by creating classes.

## Creating an Object With Type `Point`

As we have seen with functions, the above code chunk only *defines* the class. To use it, we need to create an instance of the object with the type `Point`, for example:

```
pt = Point(3,4)
```

This creates an object with type `Point` representing a point with x and y coordinates 3 and 4.

What does Python do when we call `pt = Point(3,4)`? Python first creates an *instance* of the class `Point` (which is `self`) and the special initialising function `__init__()` is called automatically when an object of the class is created. `__init__()` allows the class to initialise the attributes of the instance `self`, and in our case we assign the value 3 to `self.x` and 4 to `self.y`. The instance of `Point` created is then bound to the variable name `pt`. As the instance of `Point` has the attributes `x` and `y`, we can retrieve them by using the dot notation:

```

print(pt.x)
3
print(pt.y)
4

```

## Calling the Methods in `Point`

*Methods* are functions of a class. Every method (including `__init__()`) requires `self` to be the first argument. To use a method, we again use the dot notation. For example:

```
pt.dist_from_origin()  
5.0
```

Note that while the method `dist_from_origin()` requires an argument `self`, we do not need to provide such argument when calling it, as `self` is automatically provided to the method. In terms of the result, calling `pt.dist_from_origin()` is the same as calling:

```
Point.dist_from_origin(pt)  
5.0
```

And `pt` is given as `self` for the method. Note that you should always use `object.method()` (e.g. `pt.dist_from_origin()`) instead of `Class.method(object)` (e.g. `Point.dist_from_origin(pt)`), although they should in principle provide the same result.

## Encapsulation

### Encapsulation With Classes

*Encapsulation* is a concept of bundling data and methods together and restricting the direct access to data of the object. We have seen that the use of classes allows us to bundle data and methods together. We now show how we can restrict the direct access of the data with the example below:

```
class BankAccount:  
    ...  
  
    BankAccount represents the bank account, with methods to deposit and withdraw m  
oney from it  
    ...  
  
    def __init__(self, balance = 0):  
        assert balance >= 0, "balance has to be non-negative."  
        self._balance = balance  
  
    def deposit(self, amount):  
        ...  
  
        Add the deposit amount to the account balance.  
        Deposit amount has to be non-negative.  
        No return value.  
        ...  
  
        if amount < 0:  
            print("Deposit fail: deposit amount has to be non-negative.")
```

```

        return 0

    else:
        self._balance += amount
        return amount

def withdraw(self, amount):
    """
    Deduct the withdraw amount from the account balance.
    Withdraw amount has to be non-negative and not greater than the balance.
    Return the value of the withdrawn amount
    """
    if amount < 0:
        print("Withdraw fail: withdraw amount has to be non-negative.")
        return 0
    elif self._balance >= amount:
        self._balance -= amount
        return amount
    else:
        print("Withdraw fail: withdraw amount is more than the balance.")
        return 0

def get_balance(self):
    """
    Return current balance
    """
    return self._balance

```

One thing you may have noticed is that we defined the class attribute for the account balance as `_balance`. Any variable name with one underscore (`_`) indicates that such variable is intended for internal use. Pressing `tab` after typing `object.` in a Python terminal lists all the methods and data that are visible to users. For example for the `Point` object `pt`, we can see all the methods and data that we have defined:

```

>>> pt = Point(3,4)
>>> pt.
pt.dist(           pt.dist_from_origin(  pt.is_same(           pt.mid_point(
pt.x             pt.y

```

However, we do not see `_balance` for an instance of `BankAccount`:

```

>>> account_a = BankAccount(100)
>>> account_a.
account_a.balance(   account_a.deposit(   account_a.withdraw(

```

Why do we want to hide the account balance? It is because we do not want users to work on the *internal* data directly. For example, users may change the balance to negative, which is not a valid account balance. Instead, we provide a *getter* method `get_balance()` to allow users to see the account balance. If they want to change the amount, they should use the `withdraw()` or `deposit()` functions, as the functions check the withdraw and deposit amount to make sure the balance is always non-negative. The code below illustrates how the balance amount is updated with `withdraw()` and `deposit()`:

```
account_a = BankAccount(100)
account_a.deposit(20)
20
print(account_a.get_balance())
120
money = account_a.withdraw(200)
Withdraw fail: withdraw amount is more than the balance.
print(account_a.get_balance())
120
money = account_a.withdraw(50)
print(account_a.get_balance())
70
```

By encapsulation and data hiding with class, we prevent users from manipulating the data in an arbitrary way and ensure data integrity.

One thing to notice is that while `_balance` is marked for internal use, Python still allows you to change the value:

```
account_a._balance = -100
print(account_a.get_balance())
-100
```

You *should not* do it in this way (even you are able to do it).

## Functions

If we want to rewrite the solution with functions instead of creating a new class, we can do it in this way:

```
def deposit(balance, deposit_amount):
    """
    Return the balance with the deposit amount added to the account balance.
    Deposit amount has to be non-negative.
    """
    if deposit_amount < 0:
        print("Deposit fail: deposit amount has to be non-negative.")
    return balance
```

```

    else:
        return balance + deposit_amount

def withdraw(balance, withdraw_amount):
    """
    Return the balance with the withdraw amount deducted from the account balance.
    Deposit amount has to be non-negative and smaller than the balance
    """
    if withdraw_amount < 0:
        print("Withdraw fail: withdraw amount has to be non-negative.")
        return balance
    elif balance >= withdraw_amount:
        return balance - withdraw_amount
    else:
        print("Withdraw fail: withdraw amount is more than the balance.")
        return balance

balance = 100
balance = deposit(balance, 20)
print(balance)
120
balance = withdraw(balance, 200)
Withdraw fail: withdraw amount is more than the balance.
print(balance)
120
balance = withdraw(balance, 50)
print(balance)
70

```

Note data can be directly accessed, and functions and data are separated.

## Inheritance and Polymorphism

### Classes

*Inheritance* occurs when we define a new class (call the *child* class) that is based on an existing class (the *parent* class). Let us have a look of the following code:

```

class OverdraftAccount(BankAccount):
    """
    OverdraftAccount represents a bank account with overdraft limit

```

```

"""
def __init__(self, balance, overdraft_limit):
    assert overdraft_limit > 0, "overdraft limit has to be non-negative."
    assert balance > -overdraft_limit, "balance exceeds overdraft limit"
    self._balance = balance
    self._overdraft_limit = overdraft_limit

def withdraw(self, amount):
    """
    Deduct the withdraw amount from the account balance.
    Withdraw amount has to be non-negative and not greater than the balance with overdraft limit.
    Return the value of the withdrawn amount
    """
    if amount < 0:
        print("Withdraw fail: withdraw amount has to be non-negative.")
        return 0
    elif self._balance + self._overdraft_limit >= amount:
        self._balance -= amount
        return amount
    else:
        print("Withdraw fail: overdraft limit does not allow this withdrawal")
        return 0

def get_overdraft_limit(self):
    """
    Return the overdraft limit
    """
    return self._overdraft_limit

```

The first line of the code tells Python to define a new class `OverdraftAccount` based on the `BankAccount`. It inherits *all* the methods from `BankAccount`, and that is why we do not need to define `deposit()` or `get_balance()`.

Note we do define other methods that are already available in `BankAccount` like `withdraw()`. This is because we want the `withdraw()` method for `OverdraftAccount` to behave differently to the `withdraw()` method for `BankAccount` to reflect that the `OverdraftAccount` allows negative balance as long as it is under the overdraft limit. This is called *polymorphism*, for which a single interface to entities of different types.

Let us rerun the code above except we change the account type from `BankAccount` to `OverdraftAccount`:

```

account_b = OverdraftAccount(100, 300)
account_b.deposit(20)

```

```

20
print(account_b.get_balance())
120
money = account_b.withdraw(200)
print(account_b.get_balance())
-80
money = account_b.withdraw(50)
print(account_b.get_balance())
-130

```

Now we allow withdrawal that results in a negative balance.

There are two more methods defined for `OverdraftAccount`. One is `__init__()`, which we redefine so that we can initialise the overdraft limit, and check the `balance` argument differently (to reflect the balance can be negative). Another one is `get_overdraft_limit()`, which is not available in `BankAccount`. While `OverdraftAccount` has all the methods that `BankAccount` has, it can have *more* methods than the parent class.

## Functions

For the function approach, we *modify* the function `withdraw()` in order to accommodate overdraft limit.

```

def withdraw(balance, withdraw_amount, overdraft_limit = 0):
    """
    Return the balance with the withdraw amount deducted from the account balance.
    Deposit amount has to be non-negative and smaller than the balance
    """

    if withdraw_amount < 0:
        print("Withdraw fail: withdraw amount has to be non-negative.")
        return balance

    elif balance + overdraft_limit >= withdraw_amount:
        return balance - withdraw_amount

    else:
        print("Withdraw fail: withdraw amount is more than the balance.")
        return balance

balance = 100
overdraft_amount = 300
balance = deposit(balance, 20)
print(balance)
120
balance = withdraw(balance, 200, overdraft_amount)
print(balance)

```

```

-80
balance = withdraw(balance, 50, overdraft_amount)
print(balance)
-130

```

When using `class` for this question, we define a new child class `OverdraftAccount` to accommodate the other type of bank account that has overdraft limit. We *do not* need to change the original implementation of `BankAccount`. However, with the use of `function`, we do need to write a new function but we need to modify the original implementation of `BankAccount`.

## Programming Paradigms

*Programming paradigm* is a way or style to program. The programs we have written so far can be described as:

- *Procedural*: Programs are lists of instructions that tell the computer what to do with the program's input.
- *Functional*: Programming decomposes a problem into a set of functions, and avoids change of states as much as possible
- *Object oriented*: Programming decomposes a problem into a set of objects.

All programs we have written before this week were mostly procedural with functional *style*. We solve the problems by writing functions or a list of procedures.

The code we have written so far in Python is not actually *functional*. For functional programming, functions should only take inputs and generate outputs. It should not have any side effect (like `print()` or mutate a `list`). It also should not have any internal state that affects the output produced for a given input. While we have highlighted the danger of mutating the input and using global variables in a function, we do not prohibit the use of them. Also, we often have some internal states in a function (e.g. `x = (point_1[0] + point_2[0]) / 2` in the function `mid_point()`). Nevertheless, we have written our code in the functional *style*, in the sense that we solve the problem by a flow of functions. For the rest of the note, the word *functional programming* refers to the *style* of decomposing the problem into a series of functions, and avoid the side effect if possible.

This week we introduced how to write our own class, and we can solve problems by writing different objects. Above we have seen that in Python we can solve the same problem with a more procedural/functional programming approach or a more object oriented approach. Python is a *multiple programming paradigm* language, and it supports all three programming paradigms mentioned above.

Which paradigm to use depends on the situation:

- How the problem can be naturally represented: If the problem can be easily / naturally to be decomposed into objects, then object-oriented programming (OOP) is likely to be more suitable. If the problem can be easily/naturally be decomposed into functions, then functional programming style is likely to be more suitable.
- Readiness to use: OOP tends to require more code that takes longer to write compared to functional programming style. So if you want something fast, functional programming style may be more suitable
- Future update: If you foresee that you will have some more new types of objects with the same interface (e.g. to have more different types of bank accounts in the future with the same `withdraw()`, `deposit()` interface), then OOP is more suitable as what you need to do is to create a new child class. If you foresee that you will have the same type of objects but likely to have new functionalities (e.g. data exploration when you are unsure

what analysis to do on a piece of data), then functional programming style is more suitable as what you need is to create a new function.

- Data hiding: It is easier to do data hiding in OOP than functional programming style.

Remember different programming paradigms are different tools for you to solve a problem. You do not have to stick to one paradigm, and it is likely that you will use a mixture of paradigms for a program.

## Object-Oriented Programming in R

R provides multiple systems for OOP (e.g. `S3`, `S4`, `R6`). `S3` and `S4` are provided by base R, while `R6` is provided by the `R6` R package. Generally in R, functional programming is much more important than OOP because it is easier to solve complex problems by breaking them down into simpler functions than simple objects. See Chapter 10 of the [Advanced R textbook](#) for more details.

The example below illustrates how polymorphism is provided by the default `S3` OOP system in R:

```
x <- rnorm(100)
y <- x + rnorm(100)
mod <- lm(y~x) # fit a linear model

print(class(x))
[1] "numeric"
print(class(mod))
[1] "lm"
summary(x)
   Min. 1st Qu. Median Mean 3rd Qu. Max.
-2.290494 -0.734283 -0.008068 -0.050195 0.743667 2.429998
summary(mod)

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q      Max
-3.3788 -0.7515 -0.0380  0.8287  2.3064

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.007164   0.106501   0.067   0.947
x           1.006858   0.105160   9.574 1.02e-15 ***
---
Signif. codes:  0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1
```

```
Residual standard error: 1.064 on 98 degrees of freedom  
Multiple R-squared:  0.4833,    Adjusted R-squared:  0.478  
F-statistic: 91.67 on 1 and 98 DF,  p-value: 1.015e-15
```

Note here the function `summary()` behaves differently depending on the type of objects passed to it.

## Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Chapter 8.
- [Official Python tutorial on classes](#)
- [Functional programming how to](#)
- [Chapter 10 from “Advanced R”](#)

# ST2195 Programming for Data Science

## Block 6 Introduction to Data Wrangling

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:  
<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-6>



### Data Wrangling

Data wrangling (also known as data munging or data carpentry) refers to the design, implementation, and execution of processes that take us from raw, typically unstructured, data to data that are more appropriate for subsequent data-analytic processes. We have already seen an instance of data wrangling in “Structured, Semi-Structured and Unstructured Data,” where we used the **tm** R package to convert unstructured text to a document-term matrix, which is a structured data format that can be used for topic modelling.

As with everything in data science, it is extremely important to:

- Think carefully and design what data wrangling processes are appropriate for what you need to do;
- Implement the processes in a way that is reproducible, reusable and shareable.

Investing time to implement and share the data wrangling processes using open-source tools (e.g. git), technologies (e.g. Markdown and R Markdown), and programming languages (like Python and R) will get you a long way in the above directions.

The article [“Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions”](#) by Benureau, F. C. Y and Rougier, N. P (Benureau & Rougier, 2018) is a highly-recommended read about reproducibility and reusability of code.

The main data wrangling activities involve at least a combination of the following:

- Discovering patterns in data: For example, identifying correlations and patterns through basic data analysis and visualizations
- Structuring data: For example, subsetting, merging, re-ordering, transforming, reshaping, etc
- Cleaning and validating data: For example, identifying missing or misrecorded data that can impact the accuracy of subsequent data-analytic processes
- Enriching data: For example, thinking of what other data sources should be used to maximize the value of the data

You have already learned how to go about most of those operations in previous weeks, and you will learn more in the upcoming weeks. It is important to know that during data wrangling activities most often ad-hoc decisions need to be taken! It is good practice to record what decisions have been taken, or even “parameterize” those decisions in order to come back to them, for example to test how they impact the outcome of an analysis.

For example, the data in `heart_rates.csv` is a runner’s heart rates as recorded by a smart watch during jogging.

```
heart_rates <- read.csv("heart_rates.csv")
# convert times from character into POSIX (see ?as.POSIXct)
heart_rates$time <- as.POSIXct(heart_rates$time)
str(heart_rates)
'data.frame': 1191 obs. of 2 variables:
 $ time      : POSIXct, format: "2013-06-08 08:04:42" "2013-06-08 08:04:43"
 ...
 $ heart_rate: int  83 84 84 86 89 93 96 98 140 102 ...
head(heart_rates)
      time heart_rate
1 2013-06-08 08:04:42     83
2 2013-06-08 08:04:43     84
3 2013-06-08 08:04:44     84
4 2013-06-08 08:04:45     86
5 2013-06-08 08:04:46     89
6 2013-06-08 08:04:47     93
```

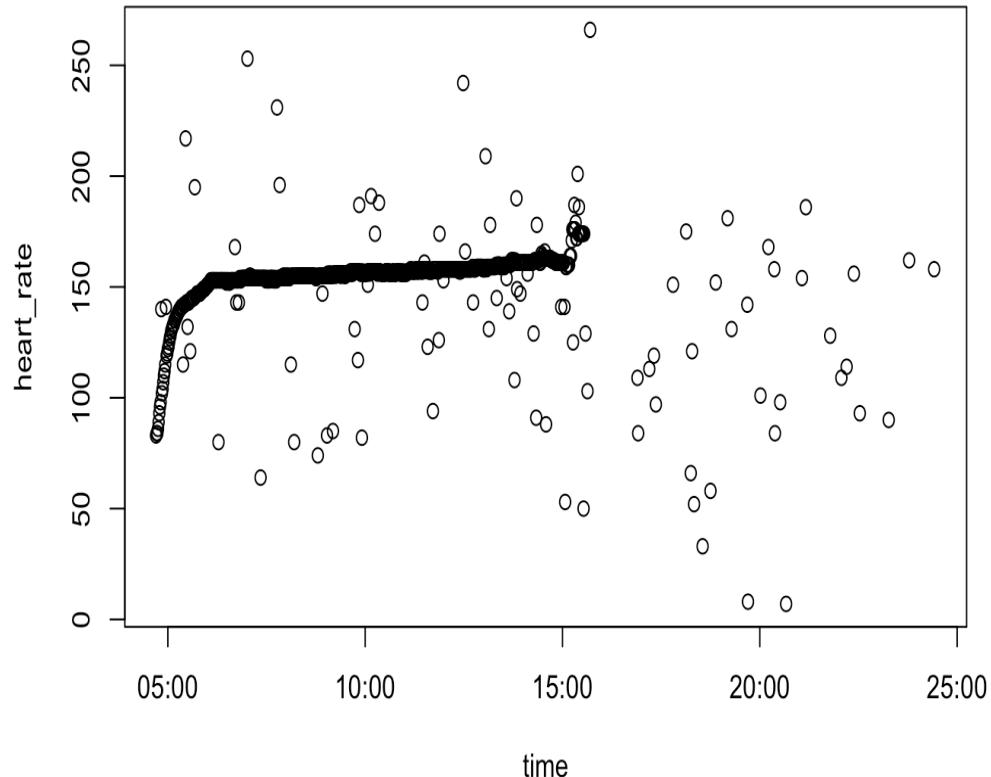
From a quick look using `str()` and `head()` we see that `heart_rate` is a data frame with times and the corresponding heart rate recordings. However, a closer look reveals some issues

```
summary(heart_rates)
      time                  heart_rate
Min.   :2013-06-08 08:04:42   Min.   : 7.0
1st Qu.:2013-06-08 08:09:38  1st Qu.:153.0
Median :2013-06-08 08:14:35  Median :156.0
Mean   :2013-06-08 08:14:34  Mean   :151.5
3rd Qu.:2013-06-08 08:19:31  3rd Qu.:158.0
Max.   :2013-06-08 08:24:27  Max.   :266.0
NA's    :501
```

There are 501 missing heart rate values, which may be because the heart rate monitor could not take a measurement at that particular timestamp. A more serious issue, though, is that the minimum recorded heart rate is `r min(heart_rates$heart_rate, na.rm = TRUE)`, which seems a bit low for someone jogging, and the maximum recorded heart rate is `r`

`max(heart_rates$heart_rate, na.rm = TRUE)`, which is a bit high for a human! A plot of the heart rates versus time reveals some issues.

```
plot(heart_rates)
```



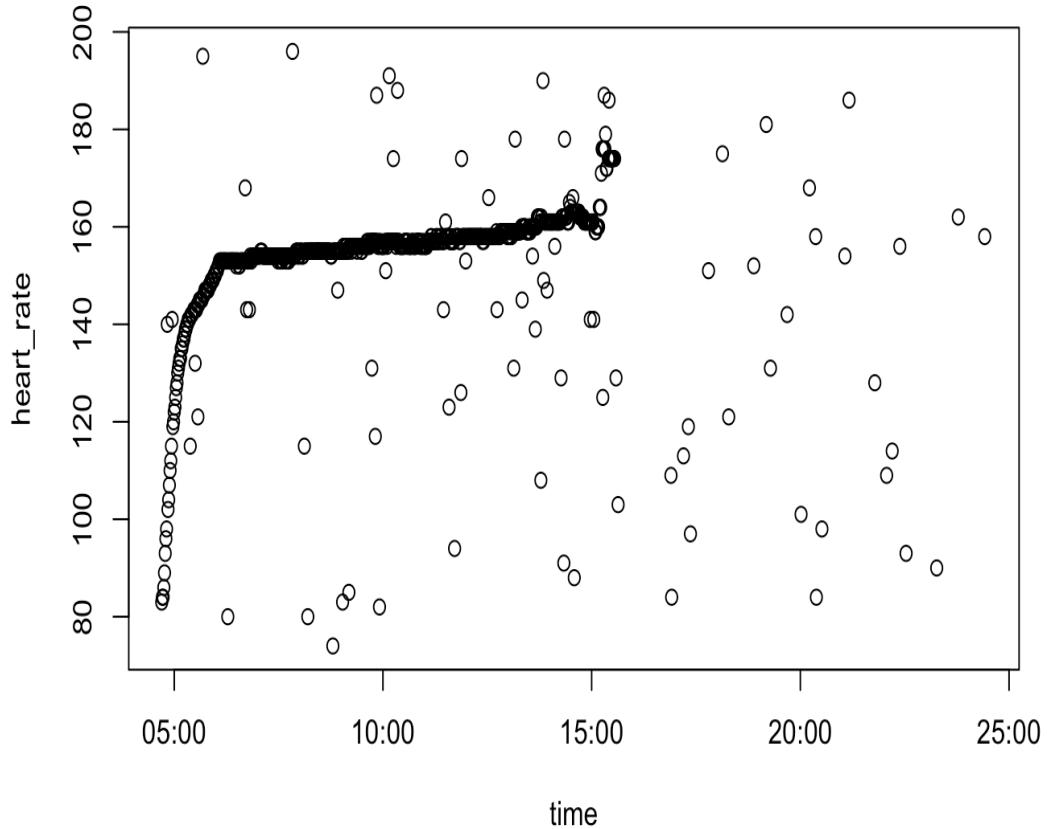
First, there seems to be a clear evolution of the heart rates for about 10 minutes from around 70 to 180, which seems normal. But the clear evolution stops abruptly after minute 15, possibly because the heart rate monitor fell off. Also there appears to be some noisy heart rate observations during the workout, which may point to an issue with the device.

Depending on what we want to do with this data, using it as is in statistical modelling may lead to misleading conclusions. We can alleviate some of those problems by writing a short function that removes “implausible” observations. For example, the below function replaces any heart rate outside the interval `(min_hr, max_hr)` with `NA`.

```
trim <- function(x, min_hr, max_hr) {
  within(heart_rates, {
    heart_rate[heart_rate < min_hr] <- NA
    heart_rate[heart_rate > max_hr] <- NA
  })
}
```

So, we can now `NA` heart rates that are less than 70 and above 200 as implausible, before passing the data to further analysis and modelling.

```
hr <- trim(heart_rates, min_hr = 70, max_hr = 200)
plot(hr)
```



In addition to that, after we complete the analysis and modelling, we can come back and adjust the values of `min_hr` and `max_hr` and repeat the analyses to see how our decisions regarding the data impact our conclusions!

In the following weeks we will discuss how missing values can be handled also as part of machine learning pipelines.

## Useful Links and Resources

- [Wikipedia's data wrangling page](#)
- [Jenny Bryan's course on "Data wrangling, exploration, and analysis with R"](#)

## References

Benureau, F. C. Y., & Rougier, N. P. (2018). Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. *Frontiers in Neuroinformatics*, 11, 69.

# Data Wrangling Operations in R



## Introduction

In this page, we will focus on some R packages and functionality that is extremely useful during data wrangling processes. In particular, we will introduce:

- The `*apply` family of R functions that allow us to apply functions to specific dimensions of matrices and arrays, and to data frames and lists
- Reshaping data sets
- Stacking data sets
- Subsetting data sets
- The verbs of the `dplyr` R package

## `*apply` Functions

The `*apply()` family is a collection of functions in R that is used to manipulate slices of data from matrices, arrays, lists and data frames in a repetitive way without explicitly writing loops. These functions apply a function to each element of the selected input data. Below, we focus on the following `*apply()` functions:

- `apply()`: Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.
- `lapply()`: Returns a list of the same length as the selected input data, each element of which is the result of applying the chosen function to the corresponding element of the selected input data.
- `sapply()`: A more user-friendly version of `lapply()`. Returns a vector or matrix by default.
- `mapply()`: This is the multivariate version of `sapply()`. It applies a function to multiple lists or vector arguments.

Please check the help pages of each of these functions for a complete description of their arguments.

### `apply()`

The `apply()` function takes a data frame or matrix as input and returns an output of vector, list or array type. The `apply()` function is mainly used to avoid explicit uses of loops.

```
# Create a simple matrix
my_matrix <- matrix(c(1:20), nrow = 5)
my_matrix
 [,1] [,2] [,3] [,4]
```

```

[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
# Using apply() to calculate the sum of each row
apply(X = my_matrix, MARGIN = 1, FUN = sum)
[1] 34 38 42 46 50
# Using apply() to calculate the sum of each column
apply(X = my_matrix, MARGIN = 2, FUN = sum)
[1] 15 40 65 90

```

In the above code chunk, we use the `apply()` function to calculate the sum of each row of the matrix by setting `MARGIN = 1`, calculate the sum of each column of the matrix by setting `MARGIN = 2`.

In both examples, we applied the `sum()` function to the margins. However, we could have used any other function, even user-defined ones, as long as they can work with the inputs specified by the margin we use. For example, if we want to get basic summaries of each of the columns we can do

```

apply(my_matrix, 2, summary)
      [,1] [,2] [,3] [,4]
Min.     1    6   11   16
1st Qu.  2    7   12   17
Median   3    8   13   18
Mean     3    8   13   18
3rd Qu.  4    9   14   19
Max.     5   10   15   20

```

As we would expect from the `summary()` function applied to a vector, the third column of the output above has

```

min(my_matrix[, 3])
[1] 11
quantile(my_matrix[, 3], 0.25)
25%
 12
median(my_matrix[, 3])
[1] 13
mean(my_matrix[, 3])
[1] 13
quantile(my_matrix[, 3], 0.75)
75%
 14
max(my_matrix[, 3])
[1] 15

```

## **lapply()**

The function `lapply()` takes a vector or list `x` and applies the function `FUN` to each of its elements. Then, `lapply()` will output a list which is of the same length as `x`, where each element is the outcome of applying the function `FUN` on the corresponding element of `x`.

```

First_name <- c("John", "Jane", "Tim", "Michael", "Emma")
Last_name <- c("Doe", "Smith", "Williams", "Taylor", "Wilson")

```

```

# Create a list from the names
Name_list <- list(First_name, Last_name)
Name_list
[[1]]
[1] "John"     "Jane"      "Tim"       "Michael" "Emma"

[[2]]
[1] "Doe"       "Smith"     "Williams" "Taylor"    "Wilson"
# Convert to lower case
Names_lower <- lapply(X = Name_list, FUN = tolower)
Names_lower
[[1]]
[1] "john"     "jane"      "tim"       "michael" "emma"

[[2]]
[1] "doe"       "smith"     "williams" "taylor"   "wilson"
str(Names_lower)
List of 2
$ : chr [1:5] "john" "jane" "tim" "michael" ...
$ : chr [1:5] "doe" "smith" "williams" "taylor" ...

```

In the above code chunk, we worked with a list of strings containing the first and last name of five people. We used `lapply()` together with the `tolower()` function to convert the names to lower case.

### **sapply()**

The function `sapply()` works in the same way as `lapply()` but simplifies (hence the `s`) the output to return a vector or a matrix.

```

# Create a simple function to raise the input to a chosen power
raise_power <- function(x, power) {
  x^power
}
# Examples
raise_power(x = 2, power = 3)
[1] 8
raise_power(x = 4, power = 2)
[1] 16
raise_power(x = 5, power = 4)
[1] 625
# Create a simple list containing numbers
numbers <- list(1:5, 6:10)
numbers
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] 6 7 8 9 10
# Using the sapply() function to raise each element of the numbers vector
# to the power of 3
sapply(X = numbers, FUN = raise_power, power = 3)
[,1] [,2]
[1,]    1   216
[2,]    8   343
[3,]   27   512
[4,]   64   729
[5,]  125  1000

```

In the above example, we created a function called `raise_power()` that takes the arguments `x` and `power` and returns `x` raised to `power`.

We then created a list called `numbers`, which contains two vectors, and used `sapply()` to apply the `raise_power()` function to each element of `numbers`. We set `power = 3` just after we have specified `x = numbers` and `FUN = raise_power`. The output is a  $(5 \times 2)$  matrix, where each column represents the elements of the first and second vectors, respectively, raised to the power of three.

### **mapply()**

The function `mapply()` is the multivariate version of `sapply()`. It applies a function in parallel over a set of arguments.

```
# Short demo of rep()
rep(1, 3)
[1] 1 1 1
rep(2, 3)
[1] 2 2 2
rep(3, 3)
[1] 3 3 3
# Create a 3x3 matrix
matrix(c(rep(1, 3), rep(2, 3), rep(3, 3)), 3, 3)
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
# Do the same thing via mapply()
mapply(rep, 1:3, 3)
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

In this example, we created a matrix `mat`, using the `rep()` function thrice. We then create the same matrix `mat1` by using `mapply()`. Basically, `mapply()` applies the `rep()` function to a vector containing the numbers 1 to 3 with additional argument 3 specifying how many times each number should be repeated. Another example is

```
mapply(rep, 1:3, (1:3)^2)
[[1]]
[1] 1

[[2]]
[1] 2 2 2 2

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

### **sweep() and aggregate()**

The `sweep()` and `aggregate()` functions are closely related to the `apply()` family. See their help files and the useful links and resources for more information.

# Reshaping Data Sets

R provides a variety of methods for reshaping your data before analysis. Base R provides the `reshape()` function which, as the help file states reshapes a data frame between “wide” format with repeated measurements in separate columns of the same record, and “long” format with the repeated measurements in separate records. The **reshape2** R package provides similar functionality but with a simplified interface. To install the **reshape2** package, type in `install.packages("reshape2")`.

The **reshape2** package makes it easy to transform data between wide and long formats:

- Wide-format data has a column for each variable
- Long-format data has a column for possible variable types and a column for the values for those variables. However, long-format data isn’t necessarily only two columns

As you may have already figured out, you may need wide-format data for some type of analyses and long-format data for others. For example, it is easy to work with **ggplot2** (we will discuss **ggplot2** graphics) with long-format data. Also, most statistical modelling functions like `lm()`, `glm()` and `gam()` work well with long-format data. However, many people often find wide-format data easier and more intuitive for recording data. Therefore, it is important to be able to work with both and be able to transform data from and into each of these two formats.

**reshape2** has two key functions:

- `melt()`: Convert wide- to long-format data
- `*cast()`: Convert long- to wide-format data

## Wide to Long

To illustrate how to use the `melt()` function in the **reshape2** package, we will consider the built-in R dataset `airquality`. This data set consists of daily air quality measurements in New York, from May to September 1973. This data is in wide format because for each data point, we have a column for each variable. In **reshape2**, to convert wide-format data to long-format data, we use the `melt()` function:

```
library("reshape2")
# print out the first few records of the airquality data frame
head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41      190  7.4   67     5    1
2    36      118  8.0   72     5    2
3    12      149 12.6   74     5    3
4    18      313 11.5   62     5    4
5    NA       NA 14.3   56     5    5
6    28       NA 14.9   66     5    6
# use the melt function to convert the data into long format
airquality_long <- melt(airquality)
No id variables; using all as measure variables
# print out the first few records of the new airquality_long data frame
```

```

head(airquality_long)
  variable value
1    Ozone    41
2    Ozone    36
3    Ozone    12
4    Ozone    18
5    Ozone    NA
6    Ozone    28
# print out the last few records of the new airquality_long data frame
tail(airquality_long)
  variable value
913    Day    25
914    Day    26
915    Day    27
916    Day    28
917    Day    29
918    Day    30

```

We see now our data is in long-format where we have one column named `variable` which records the name of the variable and a column named `value` which records the value of that variable. By default, the `melt()` function assumes that all columns are numeric with numeric values are variables with values. Often, we may want to know specific values and keep them as columns. We can identify these by using the `id.vars` argument. Here we can specify the ID variables, which are variables that identify individual rows of the data. Suppose we want to have the month and the day as ID variables:

```

# use the melt function to convert the data into long format with
# specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'))
head(airquality_long)
  Month Day variable value
1      5    1    Ozone    41
2      5    2    Ozone    36
3      5    3    Ozone    12
4      5    4    Ozone    18
5      5    5    Ozone    NA
6      5    6    Ozone    28

```

We can see that by passing in the `Month` and `Day` variables into the `id.vars` argument, we kept `Month` and `Day` as columns and the rest of the data was converted into long format. We may also wish to control the column names in our long format by passing in the column names for the variable and value into the `variable.name` and `value.name` arguments respectively:

```

# use the melt function to convert the data into long format with
# specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
                        variable.name = 'climate_var',
                        value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1      5    1    Ozone        41
2      5    2    Ozone        36
3      5    3    Ozone        12
4      5    4    Ozone        18
5      5    5    Ozone        NA
6      5    6    Ozone        28

```

```

tail(airquality_long)
  Month Day climate_var climate_value
607     9   25      Temp        63
608     9   26      Temp        70
609     9   27      Temp        77
610     9   28      Temp        75
611     9   29      Temp        76
612     9   30      Temp        68

```

## Long to Wide

Going from long to wide format can take a bit more thought than going from wide- to long-format data. In `reshape2`, there are multiple cast functions. To work with data frames, we use the `dcast()` function—there are also the `acast()` function to return a vector, matrix or array. To illustrate how to use the `dcast()` function, we work with the `airquality_long` data frame we created earlier.

The `dcast()` function uses a formula to describe the shape of the data. In this function we need to tell `dcast()` what are ID variables and what is the variable column that describes the measured variables. `dcast()` will use the last remaining column as the column that contains the values by default, but we can also pass this into the `value.var` argument. We can print this out and see that we can recover the original `airquality` data frame (but with the columns in different order):

```

# use the melt function to convert the data into long format with
# specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
                       variable.name = 'climate_var',
                       value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1     5   1      Ozone        41
2     5   2      Ozone        36
3     5   3      Ozone        12
4     5   4      Ozone        18
5     5   5      Ozone       NA
6     5   6      Ozone        28
# use the dcast function to convert the data back into wide format
airquality_wide <- dcast(airquality_long,
                         formula = Month + Day ~ climate_var,
                         value.var = 'climate_value')
head(airquality_wide)
  Month Day Ozone Solar.R Wind Temp
1     5   1    41     190  7.4   67
2     5   2    36     118  8.0   72
3     5   3    12     149 12.6   74
4     5   4    18     313 11.5   62
5     5   5    NA      NA 14.3   56
6     5   6    28      NA 14.9   66

```

A common problem users may face is when you cast a data set where there is more than one value per data cell. For instance, consider only using `Month` as an ID variable:

```

# use the melt function to convert the data into long format with
# specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),

```

```

variable.name = 'climate_var',
value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1      5    1       Ozone          41
2      5    2       Ozone          36
3      5    3       Ozone          12
4      5    4       Ozone          18
5      5    5       Ozone         NA
6      5    6       Ozone          28
# use the dcast function to convert the data back into wide format
airquality_wide <- dcast(airquality_long,
                         formula = Month ~ climate_var,
                         value.var = 'climate_value')
Aggregation function missing: defaulting to length
head(airquality_wide)
  Month Ozone Solar.R Wind Temp
1      5     31     31    31   31
2      6     30     30    30   30
3      7     31     31    31   31
4      8     31     31    31   31
5      9     30     30    30   30

```

Here we got the warning: “Aggregation function missing: defaulting to length.” By only using `Month` as the ID variable, we can see here that the cells are now filled with the number of data rows for each month. This is because we did not pass an aggregation function into the `fun.aggregate` argument which tells R how to aggregate the data.

If you cast your data and there are multiple values per cell, you will also need to tell `dcast()` how to aggregate the data. Depending on the aim of the analysis, we can use a range of scalar summaries, like the `mean()`, `median()`, `sum()`, etc. Here, we take the mean using the `mean()` function and we also use `na.rm = TRUE` to remove the NA values:

```

# use the melt function to convert the data into long format with
# specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
                        variable.name = 'climate_var',
                        value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1      5    1       Ozone          41
2      5    2       Ozone          36
3      5    3       Ozone          12
4      5    4       Ozone          18
5      5    5       Ozone         NA
6      5    6       Ozone          28
# use the dcast function to convert the data back into wide format
airquality_wide <- dcast(airquality_long,
                         formula = Month ~ climate_var,
                         value.var = 'climate_value',
                         fun.aggregate = mean,
                         na.rm = TRUE)
head(airquality_wide)
  Month   Ozone  Solar.R     Wind     Temp
1      5 23.61538 181.2963 11.622581 65.54839
2      6 29.44444 190.1667 10.266667 79.10000
3      7 59.11538 216.4839  8.941935 83.90323
4      8 59.96154 171.8571  8.793548 83.96774

```

```
5      9 31.44828 167.4333 10.180000 76.90000
```

We see here that we now obtained the mean value of the variables for each month.

## tidyr Package

There is also another popular package for data reshaping and more general data tidying, called **tidyr**. **tidyr** provides functionality for:

- Pivoting data (i.e. reshaping);
- Rectangling data (e.g. turning deeply nested lists into rectangular data);
- Nesting and unnesting (e.g. converting grouped data to a few where each group becomes a single row with a nested data frame, and the reverse);
- Splitting and combining character columns (e.g. pulling a single character column into multiple columns);
- Making implicit missing values explicit and vice versa;

Useful Links and Resources, below, provides some links of the basic functionality from **tidyr**.

## Stacking

The `stack()` and `unstack()` functions in R are handy when working with data frames. Applying `stack()` to a data frame simply stacks the columns vectors on top of each other to form a single vector along with a factor indicating where each observation came from. As expected, the `unstack()` function does the complete opposite.

Below is a demonstration using the built-in `PlantGrowth` dataset. The dataset contains the weights of 30 plants from 3 different groups, namely `ctrl` (control), `trt1` (treatment 1) and `trt2` (treatment 2). Because of the way `stack()` works, we first convert the group variable from factor to character to avoid getting a warning message.

```
# Use the built-in PlantGrowth dataset
head(PlantGrowth)
  weight group
1   4.17  ctrl
2   5.58  ctrl
3   5.18  ctrl
4   6.11  ctrl
5   4.50  ctrl
6   4.61  ctrl
# Converting group variable from factor to character
PlantGrowth$group <- as.character(PlantGrowth$group)
# Stacking
stack(PlantGrowth)
  values    ind
1   4.17 weight
2   5.58 weight
3   5.18 weight
4   6.11 weight
5     4.5 weight
6   4.61 weight
```

```
7  5.17 weight
8  4.53 weight
9  5.33 weight
10 5.14 weight
11 4.81 weight
12 4.17 weight
13 4.41 weight
14 3.59 weight
15 5.87 weight
16 3.83 weight
17 6.03 weight
18 4.89 weight
19 4.32 weight
20 4.69 weight
21 6.31 weight
22 5.12 weight
23 5.54 weight
24 5.5 weight
25 5.37 weight
26 5.29 weight
27 4.92 weight
28 6.15 weight
29 5.8 weight
30 5.26 weight
31 ctrl group
32 ctrl group
33 ctrl group
34 ctrl group
35 ctrl group
36 ctrl group
37 ctrl group
38 ctrl group
39 ctrl group
40 ctrl group
41 trt1 group
42 trt1 group
43 trt1 group
44 trt1 group
45 trt1 group
46 trt1 group
47 trt1 group
48 trt1 group
49 trt1 group
50 trt1 group
51 trt2 group
52 trt2 group
53 trt2 group
54 trt2 group
55 trt2 group
56 trt2 group
57 trt2 group
58 trt2 group
59 trt2 group
60 trt2 group
# Unstacking
unstack(PlantGrowth)
  ctrl trt1 trt2
1  4.17 4.81 6.31
2  5.58 4.17 5.12
3  5.18 4.41 5.54
4  6.11 3.59 5.50
```

```
5  4.50 5.87 5.37
6  4.61 3.83 5.29
7  5.17 6.03 4.92
8  4.53 4.89 6.15
9  5.33 4.32 5.80
10 5.14 4.69 5.26
```

Check the help file of `stack()` for more options and examples.

## Subsetting

R has some powerful indexing features that allows you to quickly access object elements. In this subsection, we will learn how to efficiently select or exclude particular elements in vectors or data frames.

### Subsetting Vectors

Consider a simple numeric vector:

```
x <- c(1.4, 5.6, 7.8, 2.6)
```

The three ways in which we can subset elements of a vector in R are:

- Using positive integers: We use a vector of positive integers to specify the index of the elements we want to return / keep.
- Using negative integers: We use a vector of negative integers to specify the index of the elements we want to exclude.
- Using logical vectors: We use a vector of logical values where elements are selected if the corresponding logical value is `TRUE`.

```
# define x
x <- c(1.4, 5.6, 7.8, 2.6)
# select the 2nd and 4th element
x[c(2,4)]
[1] 5.6 2.6
# exclude the 2nd and 4th element
x[-c(2,4)]
[1] 1.4 7.8
# select the 2nd and 4th element
x[c(FALSE, TRUE, FALSE, TRUE)]
[1] 5.6 2.6
# select the elements that are strictly greater than 5
x[x > 5]
[1] 5.6 7.8
```

By running the above code, we can see that a very useful way to subset vectors is by using logical vectors. In the last line, we first evaluated the conditional statement `x > 5` and then we used the result to subset `x`.

### Subsetting Data Frames

The most basic way of subsetting a data frame in R is using square brackets `[ ]`:

```
dataframe[x, y]
```

where `dataframe` is a data frame in R, `x` is the rows that we want to be returned, and `y` is a vector of the columns that we want returned.

```
# create dataframe
dataframe <- data.frame(v1 = 1:5, v2 = 6:10, v3 = 11:15)
# select the first 3 rows
dataframe[1:3, ]
  v1 v2 v3
1  1  6 11
2  2  7 12
3  3  8 13
# exclude the first 3 rows
dataframe[-(1:3), ]
  v1 v2 v3
4  4  9 14
5  5 10 15
# select the first 3 rows and first 2 columns
dataframe[1:3, 1:2]
  v1 v2
1  1  6
2  2  7
3  3  8
# select the rows using a logical vector (condition: if the variable v3 is
# divisible by 3
dataframe[dataframe$v3 %% 3 == 0, ]
  v1 v2 v3
2  2  7 12
5  5 10 15
# select columns v1 and v3
dataframe[, c("v1", "v3")]
  v1 v3
1  1 11
2  2 12
3  3 13
4  4 14
5  5 15
# select columns v1 and v3 (alternative)
dataframe[c("v1", "v3")]
  v1 v3
1  1 11
2  2 12
3  3 13
4  4 14
5  5 15
```

As we have seen in “Data Structures in R,” we can also subset the rows of a data frame using the `subset()` function.

## dplyr Verbs

One of the most useful packages to manipulate data is **dplyr**. This package contains the so-called **dplyr** verbs:

- `mutate()`: Adds new variables that are functions of existing variables
- `select()`: Picks variables based on their names.

- `filter()`: Picks cases based on their values.
- `summarize()`: Reduces multiple values down to a single summary.
- `arrange()`: Changes the ordering of the rows.

When working with **dplyr** verbs, we typically want to use the result of a verb applied to a data frame as input for another verb. In such cases, it is more intuitive to use the pipe `%>%` operator from the **magrittr** R package to chain operations. The **magrittr** package is loaded automatically when **dplyr** is loaded.

The variable to the left of `%>%` operator is passed as the first argument to the function on the right of the `%>%` operator.

We have already encountered some of those verbs and functionality when working with databases in R. Here, we take a more in-depth view of **dplyr**'s functionality.

Throughout this section, we use the `mtcars` data set to illustrate the capabilities of the **dplyr** verbs. Here is a preview of the dataset and its structure:

```
head(mtcars)
      mpg cyl disp hp drat wt qsec vs am gear carb
Mazda RX4     21.0   6 160 110 3.90 2.620 16.46 0 1 4 4
Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02 0 1 4 4
Datsun 710    22.8   4 108 93 3.85 2.320 18.61 1 1 4 1
Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44 1 0 3 1
Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02 0 0 3 2
Valiant       18.1   6 225 105 2.76 3.460 20.22 1 0 3 1
str(mtcars)
'data.frame': 32 obs. of 11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

### **mutate()**

The variable `mpg` stands for miles per gallon. Suppose we want to add another variable called `kpg` which stands for kilometres per gallon. We know that 1 mile is approximately equal to 1.61 kilometres. We can use `mutate()` to add the variable `kpg`:

```
# Loading the dplyr package
library(dplyr)

Attaching package: 'dplyr'
The following objects are masked from 'package:stats':
  filter, lag
The following objects are masked from 'package:base':
```

```

    intersect, setdiff, setequal, union
# Adding kpg
mtcars1 <- mutate(mtcars, kpg = 1.61 * mpg)
head(mtcars1)
      mpg cyl disp hp drat    wt  qsec vs am gear carb     kpg
Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1    4      4 33.810
Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4      4 33.810
Datsun 710    22.8   4 108  93 3.85 2.320 18.61  1  1    4      1 36.708
Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3      1 34.454
Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0    3      2 30.107
Valiant       18.1   6 225 105 2.76 3.460 20.22  1  0    3      1 29.141
# Alternative approach using the pipe %>% operator
mtcars2 <- mtcars %>% mutate(kpg = 1.61 * mpg)
head(mtcars2)
      mpg cyl disp hp drat    wt  qsec vs am gear carb     kpg
Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1    4      4 33.810
Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4      4 33.810
Datsun 710    22.8   4 108  93 3.85 2.320 18.61  1  1    4      1 36.708
Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3      1 34.454
Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0    3      2 30.107
Valiant       18.1   6 225 105 2.76 3.460 20.22  1  0    3      1 29.141
# mtcars1 is identical to mtcars2
identical(mtcars1, mtcars2)
[1] TRUE

```

### **select()**

Suppose we only want to select the mpg, cyl, hp, vs and gear columns. We can use the `select()` verb to do so.

```

mtcars %>%
  select(mpg, cyl, hp, vs, gear)
      mpg cyl hp vs gear
Mazda RX4     21.0   6 110  0    4
Mazda RX4 Wag 21.0   6 110  0    4
Datsun 710    22.8   4  93  1    4
Hornet 4 Drive 21.4   6 110  1    3
Hornet Sportabout 18.7   8 175  0    3
Valiant       18.1   6 105  1    3
Duster 360    14.3   8 245  0    3
Merc 240D     24.4   4  62  1    4
Merc 230      22.8   4  95  1    4
Merc 280      19.2   6 123  1    4
Merc 280C     17.8   6 123  1    4
Merc 450SE    16.4   8 180  0    3
Merc 450SL    17.3   8 180  0    3
Merc 450SLC   15.2   8 180  0    3
Cadillac Fleetwood 10.4   8 205  0    3
Lincoln Continental 10.4   8 215  0    3
Chrysler Imperial 14.7   8 230  0    3
Fiat 128      32.4   4  66  1    4
Honda Civic    30.4   4  52  1    4
Toyota Corolla 33.9   4  65  1    4
Toyota Corona   21.5   4  97  1    3
Dodge Challenger 15.5   8 150  0    3
AMC Javelin    15.2   8 150  0    3
Camaro Z28     13.3   8 245  0    3
Pontiac Firebird 19.2   8 175  0    3
Fiat X1-9      27.3   4  66  1    4
Porsche 914-2   26.0   4  91  0    5

```

Lotus Europa	30.4	4	113	1	5
Ford Pantera L	15.8	8	264	0	5
Ferrari Dino	19.7	6	175	0	5
Maserati Bora	15.0	8	335	0	5
Volvo 142E	21.4	4	109	1	4

### filter()

We can use the `filter()` function to get only observations that match a condition (or multiple conditions). `filter()` is the **dplyr** equivalent to `subset()`, and allows us to write more intuitive code.

```
# Get observations where gear is equal to 4 and disp is less than 80
mtcars %>% filter(gear == 4, disp < 80)
      mpg cyl disp hp drat    wt  qsec vs am gear carb
Fiat 128     32.4   4 78.7 66 4.08 2.200 19.47  1  1     4     1
Honda Civic  30.4   4 75.7 52 4.93 1.615 18.52  1  1     4     2
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1     4     1
Fiat X1-9     27.3   4 79.0 66 4.08 1.935 18.90  1  1     4     1
## this is equivalent to
subset(mtcars, gear == 4 & disp < 80)
      mpg cyl disp hp drat    wt  qsec vs am gear carb
Fiat 128     32.4   4 78.7 66 4.08 2.200 19.47  1  1     4     1
Honda Civic  30.4   4 75.7 52 4.93 1.615 18.52  1  1     4     2
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1     4     1
Fiat X1-9     27.3   4 79.0 66 4.08 1.935 18.90  1  1     4     1
```

### arrange()

We can arrange our observations in any convenient way using the `arrange()` function:

```
# Arrange mtcars in descending order of mpg
mtcars %>%
  arrange(mpg) %>%
  head()
      mpg cyl disp hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood 10.4   8 472 205 2.93 5.250 17.98  0  0     3     4
Lincoln Continental 10.4   8 460 215 3.00 5.424 17.82  0  0     3     4
Camaro Z28         13.3   8 350 245 3.73 3.840 15.41  0  0     3     4
Duster 360          14.3   8 360 245 3.21 3.570 15.84  0  0     3     4
Chrysler Imperial  14.7   8 440 230 3.23 5.345 17.42  0  0     3     4
Maserati Bora       15.0   8 301 335 3.54 3.570 14.60  0  1     5     8
# Arrange mtcars in descending order of disp
mtcars %>%
  arrange(desc(disp)) %>%
  head()
      mpg cyl disp hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood 10.4   8 472 205 2.93 5.250 17.98  0  0     3     4
Lincoln Continental 10.4   8 460 215 3.00 5.424 17.82  0  0     3     4
Chrysler Imperial  14.7   8 440 230 3.23 5.345 17.42  0  0     3     4
Pontiac Firebird    19.2   8 400 175 3.08 3.845 17.05  0  0     3     2
Hornet Sportabout   18.7   8 360 175 3.15 3.440 17.02  0  0     3     2
Duster 360          14.3   8 360 245 3.21 3.570 15.84  0  0     3     4
```

### summarize()

We can compute several summaries using the `summarize()` function. For example, if we want to calculate the maximum `mpg` value, the minimum value `disp`, and the mean of `qsec` in `mtcars` we do:

```
mtcars %>%
  summarize(maximum_mpg = max(mpg),
            minimum_disp = min(disp),
            average_qsec = mean(qsec))
maximum_mpg minimum_disp average_qsec
1          33.9        71.1      17.84875
```

`summarize()` combines well with the `group_by()` function. For example,

```
iris %>%
  group_by(Species) %>%
  summarize(average_Sepal_Length = mean(Sepal.Length),
            Average_Sepal_Width = mean(Sepal.Width),
            average_Petal_Length = mean(Petal.Length),
            average_Petal_Width = mean(Petal.Width))
# A tibble: 3 x 5
#>   Species  average_Sepal_L... Average_Sepal_Wi... average_Petal_L...
#>   <fct>       <dbl>           <dbl>           <dbl>
#> 1 setosa       5.01            3.43            1.46
#> 2 versico...    5.94            2.77            4.26
#> 3 virginici... 6.59            2.97            5.55
```

Try to understand exactly what the above code chunk does.

### \*`_join()`

**dplyr** also provides verbs for joining data sets. Below, we use the `band_members` and `band_instruments` tibbles (which are `data.frame` structures) to illustrate.

```
band_members
# A tibble: 3 x 2
  name   band
  <chr> <chr>
1 Mick   Stones
2 John   Beatles
3 Paul   Beatles
band_instruments
# A tibble: 3 x 2
  name   plays
  <chr> <chr>
1 John   guitar
2 Paul   bass
3 Keith  guitar
```

Suppose that we want to find all rows of `band_members` for which there are matching values in `band_instruments`, and return all columns from the two tibbles. We can achieve that using `inner_join`:

```

band_members %>% inner_join(band_instruments)
Joining, by = "name"
# A tibble: 2 x 3
  name   band   plays
  <chr> <chr>  <chr>
1 John   Beatles guitar
2 Paul   Beatles bass

```

We can also use `left_join()` to return all rows of `band_members` and all columns from the two tibbles, with NA in the new columns if there are no matches in the rows of `band_members` and `band_instruments`:

```

band_members %>% left_join(band_instruments)
Joining, by = "name"
# A tibble: 3 x 3
  name   band   plays
  <chr> <chr>  <chr>
1 Mick   Stones <NA>
2 John   Beatles guitar
3 Paul   Beatles bass

```

We can use `right_join()` to return all rows of `band_instruments` and all columns from the two tibbles, with NA in the new columns if there are no matches in the rows of `band_instruments` and `band_members`

```

band_members %>% right_join(band_instruments)
Joining, by = "name"
# A tibble: 3 x 3
  name   band   plays
  <chr> <chr>  <chr>
1 John   Beatles guitar
2 Paul   Beatles bass
3 Keith  <NA>    guitar

```

Similarly, we can return all rows and columns of both `band_members` and `band_instruments` with NA in the new columns if there are no matches in the rows of `band_instruments` and `band_members`

```

band_members %>% full_join(band_instruments)
Joining, by = "name"
# A tibble: 4 x 3
  name   band   plays
  <chr> <chr>  <chr>
1 Mick   Stones <NA>
2 John   Beatles guitar
3 Paul   Beatles bass
4 Keith  <NA>    guitar

```

Notice that the same join (with different ordering of rows) can be obtained by

```

band_instruments %>% full_join(band_members)
Joining, by = "name"
# A tibble: 4 x 3
  name   plays   band
  <chr> <chr>  <chr>
1 John   guitar  Beatles
2 Paul   bass    Beatles

```

```
3 Keith guitar <NA>
4 Mick <NA> Stones
```

All examples above also work with `data.frames`. More information and examples on **dplyr** verbs for joining data sets can be found at [dplyr's join pages](#).

## Useful Links and Resources

- [Datacamp's tutorial on the `\*apply\(\)` family and associated functions](#)
- [MarinStatsLectures in Youtube on the `apply` function](#)
- [Xianjun Dong's post on `reshape2` and `tidy2`](#)
- [Overview of the `tidyR` R package](#)
- [The chapter on subsetting in \*Advanced R\*](#)
- [dplyr's join pages](#) for verbs to join `data.frames` and `tibbles`

# Data Wrangling Operations in Python



\*\* Note: The code chunks below should be run in the following order \*\*

## Introduction

In this page we will use the same dataset `airquality`, from the unit on data wrangling in R. We first export the data from R to CSV files so that the dataset can be accessible for Python:

```
write.table(airquality, file = "airquality.csv", sep = ",", row.names = FALSE)
```

We then load the data to Python, and rearrange the columns so that `Month` and `Day` information are in the first two columns:

```
import pandas as pd
import numpy as np

airquality = pd.read_csv("airquality.csv")
airquality = airquality[['Month', 'Day', 'Ozone', 'Solar.R', 'Temp', 'Wind']]
airquality.head()

   Month  Day  Ozone  Solar.R  Temp  Wind
0       5     1    41.0    190.0    67    7.4
1       5     2    36.0    118.0    72    8.0
2       5     3    12.0    149.0    74   12.6
3       5     4    18.0    313.0    62   11.5
4       5     5      NaN      NaN    56   14.3
```

There is some missing data in the dataset. We can see the data as in the wide format as as it has a column for each variable (`Ozone`, `Solar.R`, `Temp`, `Wind`). In each row, we get the values for `Ozone`, `Solar.R`, `Temp`, `Wind` level for a particular day. For example, for example, the first row has `Ozone = 41.0`, `Solar.R = 190.0`, `Temp = 67`, `Wind = 7.4` for the day 1 May.

In this page we will first show how we can reshape the dataset. We then show how we can select a subset of the data. Finally we show how we can handle the missing data.

# Reshaping Dataset

Python `pandas` provides a variety of methods/functions for reshaping your data before analysis, and many of the methods are very similar to the R functions. We will reshape the data with the following methods:

- Wide to long: `pandas.melt()`, `stack()`
- Long to wide: `pivot_table()`

## Wide to Long

We can convert the data from wide to long by the `pandas.melt()` function:

```
pd.melt(airquality)
      variable   value
0       Month    5.0
1       Month    5.0
2       Month    5.0
3       Month    5.0
4       Month    5.0
..
...
913     Wind    6.9
914     Wind   13.2
915     Wind   14.3
916     Wind    8.0
917     Wind   11.5

[918 rows x 2 columns]
```

Similar to `melt()` in R, `pandas.melt()` in Python will set the first column named `variable` which records the name of the variable and a second column named `value` which records the value of that variable. For this data set it is not really useful to convert the data to the `DataFrame` above. Instead we should keep `Month` and `Day` as columns:

```
airquality_long = pd.melt(airquality, id_vars = ['Month', 'Day'], value_vars = ['Ozone', 'Solar.R', 'Wind', 'Temp'],
                           var_name='climate_var', value_name='climate_value')

airquality_long.head()
      Month  Day climate_var  climate_value
0       5     1      Ozone        41.0
1       5     2      Ozone        36.0
2       5     3      Ozone        12.0
3       5     4      Ozone        18.0
4       5     5      Ozone        NaN
```

In this representation, each row gives the reading of a particular measure for a particular day. For example, the first row tells us that the ozone level is 41.0 on 1 May. This is done by providing extra arguments to the function `pandas.melt()`. The `id_vars = ['Month', 'Day']` argument specifies that `Month` and `Day` columns should be kept. By providing the argument `value_vars = ['Ozone', 'Solar.R', 'Wind', 'Temp']`, the other columns (`Ozone`, `Solar.R`, `Temp`, `Wind`) are converted into long format. We set the column names in our long format by passing the additional arguments `var_name='climate_var'`, `value_name='climate_value'`. You may notice the syntax for `pandas.melt()` is very similar to the one used in `melt()` in R.

## Long to Wide

We can use the method `pivot_table()` from `pandas.DataFrame` to convert data from long to wide format:

```
airquality_wide = airquality_long.pivot_table(values='climate_value', index=['Month', 'Day'], columns=['climate_var'])

airquality_wide.head()

climate_var  Ozone  Solar.R  Temp  Wind

Month Day
5      1       41.0     190.0   67.0    7.4
                  2       36.0     118.0   72.0    8.0
                  3       12.0     149.0   74.0   12.6
                  4       18.0     313.0   62.0   11.5
                  5       NaN      NaN    56.0   14.3
```

Note that while the resulting table `airquality_wide` is very similar to the original table `airquality`, we now have `Multindex` which groups `Month` and `Day` together for `airquality_wide`:

```
airquality_wide.index
MultiIndex([(5, 1),
            (5, 2),
            (5, 3),
            (5, 4),
            (5, 5),
            (5, 6),
            (5, 7),
            (5, 8),
            (5, 9),
            (5, 10),
            ...
            (9, 21),
            (9, 22),
            (9, 23),
            (9, 24),
```

```

(9, 25),
(9, 26),
(9, 27),
(9, 28),
(9, 29),
(9, 30)],
names=['Month', 'Day'], length=153)

```

This is because we set `index=['Month', 'Day']` so that two columns are used as key. For the other arguments, `columns=['climate_var']` specifies that the data should be grouped by the values in the column `climates_var` (i.e. `Ozone`, `Solar.R`, `Temp`, `Wind`) for the columns, and `values='climate_value'` specifies that the values from the column `climate_value` should be used to fill in the table. In our example, there is only one instance for the combination of row and column (e.g. `Month`, `Day`, `Ozone`), so the value from `climate_value` is used directly. For `airquality_wide`, each cell in the table represents the reading of a air quality measure (read from the column name) on a particular day (read from the `MultiIndex` in the row). For example, the value `41.0` in the first cell is the reading of `Ozone` on 1 May.

What if there is more than one instance for the combination of row and column? For example, if we consider using only `Month` for the index, then for a given `Month` and `Ozone`, we will have multiple values `41.0`, `36.0`, etc from different values of `Day`. What does `pivot_table()` return in this case?

```

airquality_month = airquality_long.pivot_table(values='climate_value', index=['Month'],
                                               columns=['climate_var'])

airquality_month.head()

climate_var      Ozone      Solar.R       Temp       Wind
Month
5              23.615385  181.296296  65.548387  11.622581
6              29.444444  190.166667  79.100000  10.266667
7              59.115385  216.483871  83.903226  8.941935
8              59.961538  171.857143  83.967742  8.793548
9              31.448276  167.433333  76.900000  10.180000

```

It seems that the value inside the table is the average value. We can confirm it by considering the following code, which we specify using `mean` to aggregating the data by the argument `aggfunc=np.mean`:

```

airquality_month = airquality_long.pivot_table(values='climate_value', index=['Month'],
                                               columns=['climate_var'], aggfunc=np.mean)

airquality_month.head()

climate_var      Ozone      Solar.R       Temp       Wind
Month
5              23.615385  181.296296  65.548387  11.622581
6              29.444444  190.166667  79.100000  10.266667
7              59.115385  216.483871  83.903226  8.941935
8              59.961538  171.857143  83.967742  8.793548

```

```
9          31.448276  167.433333  76.900000  10.180000
```

As we can see, the return values are the same, showing us indeed averaging is used by default if we have multiple values for a given combination. For the table above, the value in the first cell (23.6) represents the monthly average ozone reading for `Month = 5` (i.e. May).

We can use other aggregating function, for example `np.max` to get the maximum value:

```
airquality_month = airquality_long.pivot_table(values='climate_value', index=['Month'], columns=['climate_var'], aggfunc=np.max)

airquality_month.head()

climate_var  Ozone  Solar.R  Temp  Wind

Month

5            115.0    334.0   81.0  20.1
6             71.0    332.0   93.0  20.7
7            135.0    314.0   92.0  14.9
8            168.0    273.0   97.0  15.5
9             96.0    259.0   93.0  16.6
```

For the table above, the value in the first cell (115.0) represents the maximum ozone reading for `Month = 5` (i.e. May).

## Stack and Unstack

Like R, we can stack the `DataFrame`. All the columns except the index (or MultiIndex) are stacked into one new column:

```
airquality_stack = airquality_wide.stack()

airquality_stack

Month  Day  climate_var

5      1    Ozone        41.0
                  Solar.R     190.0
                  Temp         67.0
                  Wind         7.4
      2    Ozone        36.0
                  ...
9      29   Wind         8.0
      30    Ozone       20.0
                  Solar.R    223.0
                  Temp        68.0
                  Wind        11.5

Length: 568, dtype: float64
```

We can turn it back to the original data structure by `unstack()`:

```
airquality_stack.unstack()
```

```

climate_var  Ozone  Solar.R  Temp  Wind
Month   Day
5       1      41.0    190.0   67.0   7.4
        2      36.0    118.0   72.0   8.0
        3      12.0    149.0   74.0  12.6
        4      18.0    313.0   62.0  11.5
        5      NaN     NaN    56.0  14.3
...
9       26     30.0    193.0   70.0   6.9
        27     NaN     145.0   77.0  13.2
        28     14.0    191.0   75.0  14.3
        29     18.0    131.0   76.0   8.0
        30     20.0    223.0   68.0  11.5

```

[153 rows x 4 columns]

## Subsetting

Like in R, `pandas` in Python allows users to access a subset of data very easily.

### Subsetting `DataFrame` Columns

We can get a column from a `DataFrame` by simply giving the column name in the square bracket:

```

airquality['Ozone']

0      41.0
1      36.0
2      12.0
3      18.0
4      NaN
...
148    30.0
149    NaN
150    14.0
151    18.0
152    20.0

Name: Ozone, Length: 153, dtype: float64

```

Note that the above code chunk returns a `Series` (i.e. not a `DataFrame`)

```
type(airquality['Ozone'])
```

```
<class 'pandas.core.series.Series'>
```

To return a `DataFrame` use:

```
airquality[['Ozone']]
```

```
    Ozone  
0      41.0  
1      36.0  
2      12.0  
3      18.0  
4      NaN  
..     ...  
148    30.0  
149    NaN  
150    14.0  
151    18.0  
152    20.0
```

```
[153 rows x 1 columns]
```

To select more than one column, use `[['col1', 'col2', ...]]`:

```
airquality[['Ozone', 'Temp']]
```

```
    Ozone  Temp  
0      41.0    67  
1      36.0    72  
2      12.0    74  
3      18.0    62  
4      NaN     56  
..     ...     ...  
148    30.0    70  
149    NaN     77  
150    14.0    75  
151    18.0    76  
152    20.0    68
```

```
[153 rows x 2 columns]
```

Alternative, you can use the following:

```
airquality.iloc[:, np.array([2, 4])]
```

```
    Ozone  Temp
```

```
0      41.0    67
1      36.0    72
2      12.0    74
3      18.0    62
4      NaN     56
...
148    30.0    70
149    NaN     77
150    14.0    75
151    18.0    76
152    20.0    68
```

```
[153 rows x 2 columns]
```

In the above, inside the square brackets, the rows are specified first (here `:` means all) and, after the coma, the columns are specified by `(np.array([2,4]))` that corresponds to the indices of the `Ozone` and `Temp` columns.

## Subsetting `DataFrame` Rows:

We can get rows from a `DataFrame` by simply giving the row indexes in the square bracket:

```
airquality[:5]
   Month Day Ozone Solar.R Temp Wind
0      5    1  41.0  190.0   67  7.4
1      5    2  36.0  118.0   72  8.0
2      5    3  12.0  149.0   74 12.6
3      5    4  18.0  313.0   62 11.5
4      5    5    NaN      NaN   56 14.3
```

We can also use `.loc[]` and `.iloc[]` to select rows. `.loc[]` is a label-based way to get the specific rows (or columns as well), whereas `.iloc[]` is an *index-based* way to get the specific rows (or columns as well). For example, for the `airquality` dataset, we can get the first five rows by

```
airquality.loc[:5]
   Month Day Ozone Solar.R Temp Wind
0      5    1  41.0  190.0   67  7.4
1      5    2  36.0  118.0   72  8.0
2      5    3  12.0  149.0   74 12.6
3      5    4  18.0  313.0   62 11.5
4      5    5    NaN      NaN   56 14.3
5      5    6  28.0      NaN   66 14.9
```

or

```
airquality.iloc[:5]
```

	Month	Day	Ozone	Solar.R	Temp	Wind
0	5	1	41.0	190.0	67	7.4
1	5	2	36.0	118.0	72	8.0
2	5	3	12.0	149.0	74	12.6
3	5	4	18.0	313.0	62	11.5
4	5	5	NaN	NaN	56	14.3

However, if we use the same method on `airquality_wide` instead, the result is different for `.loc[]`:

```
airquality_wide.loc[:5]
```

	climate_var	Ozone	Solar.R	Temp	Wind
Month	Day				
5	1	41.0	190.0	67.0	7.4
	2	36.0	118.0	72.0	8.0
	3	12.0	149.0	74.0	12.6
	4	18.0	313.0	62.0	11.5
	5	NaN	NaN	56.0	14.3
	6	28.0	NaN	66.0	14.9
	7	23.0	299.0	65.0	8.6
	8	19.0	99.0	59.0	13.8
	9	8.0	19.0	61.0	20.1
	10	NaN	194.0	69.0	8.6
	11	7.0	NaN	74.0	6.9
	12	16.0	256.0	69.0	9.7
	13	11.0	290.0	66.0	9.2
	14	14.0	274.0	68.0	10.9
	15	18.0	65.0	58.0	13.2
	16	14.0	334.0	64.0	11.5
	17	34.0	307.0	66.0	12.0
	18	6.0	78.0	57.0	18.4
	19	30.0	322.0	68.0	11.5
	20	11.0	44.0	62.0	9.7
	21	1.0	8.0	59.0	9.7
	22	11.0	320.0	73.0	16.6
	23	4.0	25.0	61.0	9.7
	24	32.0	92.0	61.0	12.0
	25	NaN	66.0	57.0	16.6

```

26      NaN    266.0   58.0  14.9
27      NaN      NaN  57.0   8.0
28     23.0    13.0   67.0  12.0
29     45.0   252.0   81.0  14.9
30    115.0   223.0   79.0   5.7
31     37.0   279.0   76.0   7.4

airquality_wide.iloc[:5]

climate_var  Ozone  Solar.R  Temp  Wind
Month Day
5      1      41.0    190.0   67.0   7.4
      2      36.0    118.0   72.0   8.0
      3      12.0    149.0   74.0  12.6
      4      18.0    313.0   62.0  11.5
      5      NaN      NaN   56.0  14.3

```

Why is this the case? Note that the *label* of the index for `airquality_wide` is:

```

airquality_wide.index
MultiIndex([(5,  1),
            (5,  2),
            (5,  3),
            (5,  4),
            (5,  5),
            (5,  6),
            (5,  7),
            (5,  8),
            (5,  9),
            (5, 10),
            ...
            (9, 21),
            (9, 22),
            (9, 23),
            (9, 24),
            (9, 25),
            (9, 26),
            (9, 27),
            (9, 28),
            (9, 29),
            (9, 30)],
           names=['Month', 'Day'], length=153)

```

so `airquality_wide.loc[:5]` gets all rows with `Month = 5`. See [here](#) for more details.

We can also get the rows by condition, for example:

```
airquality.loc[airquality.Day == 3, 'Ozone']  
2      12.0  
33     NaN  
63     32.0  
94     16.0  
125    73.0  
  
Name: Ozone, dtype: float64
```

The above code chunk returns the ozone reading on 3 May, 3 June, ..., 3 September.

## Subsetting `DataFrame` Rows and Columns

Above we only specify one value in `.loc[]` and `.iloc[]`. If we specify one more value, we can specify the columns as well. For example, if we want to get the first row with ozone readings, we can use the `.loc[]`

```
airquality.loc[:5, 'Ozone']  
0      41.0  
1      36.0  
2      12.0  
3      18.0  
4      NaN  
5      28.0  
  
Name: Ozone, dtype: float64
```

Or the `.iloc[]`:

```
airquality.iloc[:5, 2]  
0      41.0  
1      36.0  
2      12.0  
3      18.0  
4      NaN  
  
Name: Ozone, dtype: float64
```

## Handling Missing Values

You probably can see that there are some missing value in our dataset:

```
airquality.head(10)  
Month  Day  Ozone  Solar.R  Temp  Wind
```

0	5	1	41.0	190.0	67	7.4
1	5	2	36.0	118.0	72	8.0
2	5	3	12.0	149.0	74	12.6
3	5	4	18.0	313.0	62	11.5
4	5	5	NaN	NaN	56	14.3
5	5	6	28.0	NaN	66	14.9
6	5	7	23.0	299.0	65	8.6
7	5	8	19.0	99.0	59	13.8
8	5	9	8.0	19.0	61	20.1
9	5	10	NaN	194.0	69	8.6

We can remove them by `.dropna()`, which removes rows containing *any* missing data.

```
airquality_no_na = airquality.dropna()
airquality_no_na.head(10)

   Month Day Ozone Solar.R Temp Wind
0      5    1   41.0    190.0  67  7.4
1      5    2   36.0    118.0  72  8.0
2      5    3   12.0    149.0  74 12.6
3      5    4   18.0    313.0  62 11.5
6      5    7   23.0    299.0  65  8.6
7      5    8   19.0    99.0   59 13.8
8      5    9    8.0    19.0   61 20.1
11     5   12   16.0    256.0  69  9.7
12     5   13   11.0    290.0  66  9.2
13     5   14   14.0    274.0  68 10.9
```

Sometimes, removing the rows with missing data may not be appropriate. Instead we may want to fill in the missing values in a systematic way. For example, we can replace the missing values with previous value:

```
airquality_fill_na = airquality.fillna(method='ffill')
airquality_fill_na.head(10)

   Month Day Ozone Solar.R Temp Wind
0      5    1   41.0    190.0  67  7.4
1      5    2   36.0    118.0  72  8.0
2      5    3   12.0    149.0  74 12.6
3      5    4   18.0    313.0  62 11.5
4      5    5   18.0    313.0  56 14.3
5      5    6   28.0    313.0  66 14.9
6      5    7   23.0    299.0  65  8.6
7      5    8   19.0    99.0   59 13.8
```

8	5	9	8.0	19.0	61	20.1
9	5	10	8.0	194.0	69	8.6

There are different settings for the fill rule. See [the pandas `fillna\(\)` method for DataFrame](#) for more details. Alternatively we can use interpolation to fill the missing values. For example:

```
airquality_linear = airquality.interpolate(method='linear')
airquality_linear.head(10)

   Month Day Ozone Solar.R Temp Wind
0      5    1   41.0  190.000000  67  7.4
1      5    2   36.0  118.000000  72  8.0
2      5    3   12.0  149.000000  74 12.6
3      5    4   18.0  313.000000  62 11.5
4      5    5   23.0  308.333333  56 14.3
5      5    6   28.0  303.666667  66 14.9
6      5    7   23.0  299.000000  65  8.6
7      5    8   19.0  99.000000  59 13.8
8      5    9    8.0  19.000000  61 20.1
9      5   10    7.5  194.000000  69  8.6
```

Here we use linear method to interpolate, with the missing ozone data on 5 May calculated as  $(18+28)/2 = 23$ .

## Combining Different Datasets

We downloaded the coronavirus new cases and new deaths data from <https://coronavirus.data.gov.uk>. We load the data into `cases_df` and `death_df` via `pandas`

```
import pandas as pd
cases_df = pd.read_csv("covid_new_cases.csv", index_col = 0)
deaths_df = pd.read_csv("covid_new_deaths.csv", index_col = 0)
```

Both `DataFrame` has `date` as the index (i.e. the row labels), although the `case_df` has more data.

```
print(cases_df)
      new_cases
date
2020-01-30      2
2020-01-31      0
2020-02-01      0
2020-02-02      0
2020-02-03      0
...       ...
```

```
2020-10-25    15654
2020-10-26    26467
2020-10-27    23757
2020-10-28    22887
2020-10-29    19525
```

```
[274 rows x 1 columns]
```

```
print(deaths_df)

new_deaths

date

2020-02-29      0
2020-03-01      0
2020-03-02      1
2020-03-03      2
2020-03-04      0
...
...
2020-10-25    234
2020-10-26    253
2020-10-27    227
2020-10-28    216
2020-10-29    217
```

```
[244 rows x 1 columns]
```

Here we want to join the two datasets together. `pandas` has fast and full-featured functions for combining datasets via the method `join()`, and these joins can be one-to-one, many-to-one or many-to-many. These merges can be done in a number of ways shown in the table below:

Method	Behaviour
left	Use calling frame's index (or column if on is specified)
right	Use other's index
outer	Use union of keys from both DataFrame
inner	Use intersection of keys from both DataFrame

For example, we can join the `cases_df` and `death_df` by:

```
df_1 = cases_df.join(deaths_df)

print(df_1)

new_cases  new_deaths

date

2020-01-30      2        NaN
2020-01-31      0        NaN
2020-02-01      0        NaN
2020-02-02      0        NaN
```

```

2020-02-03      0      NaN
...
2020-10-25  15654    234.0
2020-10-26  26467    253.0
2020-10-27  23757    227.0
2020-10-28  22887    216.0
2020-10-29  19525    217.0

```

[274 rows x 2 columns]

In the above we use all the row labels from `cases_df`, as the default is `left` join. We can see that the number of rows of the resulting `DataFrame` is the same as the number of rows in `cases_df`, as we use all the row labels from `cases_df`. We also see that there are some NA data for the `new_deaths` column in `df_1`, as for the dates like `2020-01-30` there is no data available from the `deaths_df`.

If we want to use all the keys of `deaths_df` instead, we can add the argument `how = 'right'`:

```

df_2 = cases_df.join(deaths_df, how = 'right')
print(df_2)

      new_cases  new_deaths
date
2020-02-29      5          0
2020-03-01     22          0
2020-03-02     40          1
2020-03-03     56          2
2020-03-04     56          0
...
2020-10-25  15654        234
2020-10-26  26467        253
2020-10-27  23757        227
2020-10-28  22887        216
2020-10-29  19525        217

```

[244 rows x 2 columns]

Now we have the number of rows of `df_2` equals to the number of rows in `deaths_df`, as we use all the row labels from `deaths_df`.

If we want to the keys that are found in both `cases_df` and `deaths_df`, we can use `how = 'inner')`:

```

df_3 = cases_df.join(deaths_df, how = 'inner')
print(df_3)

      new_cases  new_deaths

```

```
date          5      0
2020-02-29    5      0
2020-03-01   22      0
2020-03-02   40      1
2020-03-03   56      2
2020-03-04   56      0
...
...          ...
2020-10-25 15654    234
2020-10-26 26467    253
2020-10-27 23757    227
2020-10-28 22887    216
2020-10-29 19525    217
```

[244 rows x 2 columns]

For this particular example, `df_2` and `df_3` are the same as the intersection of the dates (rows) from `cases_df` and `deaths_df`, is the same as the dates in `deaths_df`.

## Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapter 7.
- [pandas.DataFrame official documentation](#)

# ST2195 Programming for Data Science

## Block 7 Exploratory Analysis and Data Visualisation

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:  
<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-7>

### Workflow of a Data Science Project

A data science project typically involves the following steps:

1. Importing the data, cleaning them and preparing them for use (data wrangling)
2. Get some basic level of understanding on the data by exploration and visualisation.
3. Consider several models as well as combinations thereof to separate signal from noise.
4. Compare models and inform future decisions.

So far we have covered the first step. In this week, as well as the next three weeks that follow, we will focus on the second of the steps above. The aim is to identify patterns in the data that can potentially help us take more informed decisions.

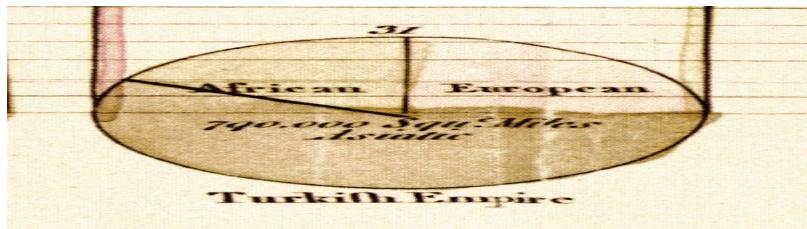
### First Examples of Graphs in History

The use of *graphs*, i.e. visual summaries to summarise information from data, certainly flourished and gained popularity as computers started to develop. But it all had started several centuries ago, even before 1800. Some examples are given below:

- Graphs can be found in the early work of the Scottish engineer and political economist, *William Playfair*, who essentially introduced the *bar plot*. See below for an example depicting the imports and exports in Scotland in 1781:



Playfair also produced *pie charts*, see below for one example providing information on the Turkish empire:



(Images taken from Wikipedia, visit [here](#) (bar plot) and [here](#) (pie chart) for more information)

- *Charles Joseph Minard*, a French civil engineer produced several graphs, with one example being the illustration of Napoleon's campaign in Russia:

# Carte Figurative des pertes successives en hommes de l'Armée Française dans la Campagne de Russie 1812-1813.

Dessiné par M. Minard, Inspecteur Général des Ponts et Chaussées en retraite.

Paris, le 20 Novembre 1869.

Les nombres d'hommes perdus sont représentés par les largesses des zones colorées à raison d'un millimètre pour dix mille hommes; ils sont de plus écrits en lettres dans les zones. Le rouge désigne les hommes qui entrent en Russie, le noir ceux qui en sortent. — Les renseignements qui ont servi à dresser la carte ont été pris dans les ouvrages de M. Chiers, de Léger, de Tocencac, de Chambray et le journal médical de Jacob, pharmacien de l'Armée depuis le 28 Octobre.

Pour mieux faire juger à l'œil la diminution de l'armée, j'ai supposé que les corps de l'Empereur et du Maréchal Davout qui avaient été détachés de Minsk au Mohilow et qui étaient rentrés vers Orsha et Wilna, avaient toujours marché avec l'armée.

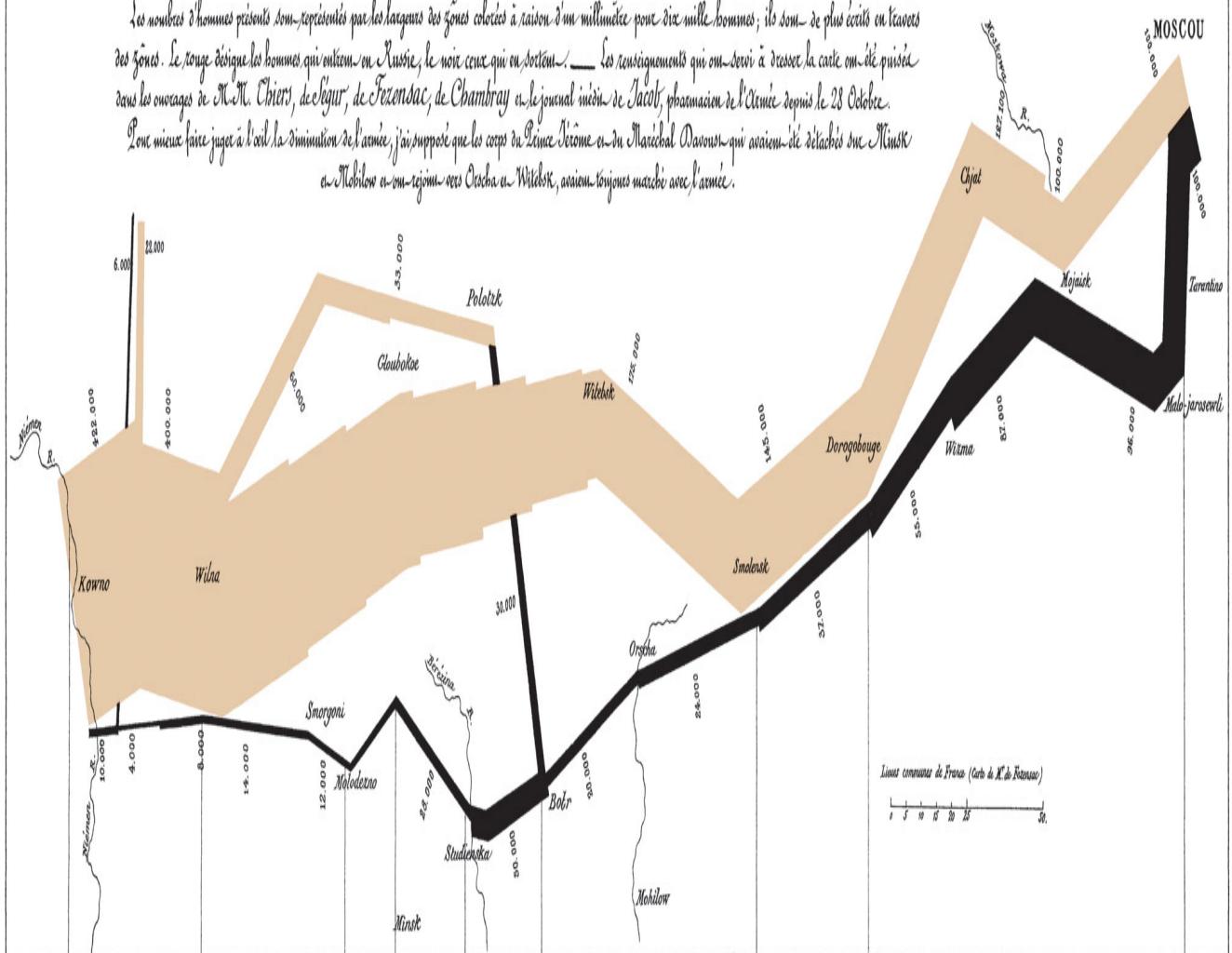
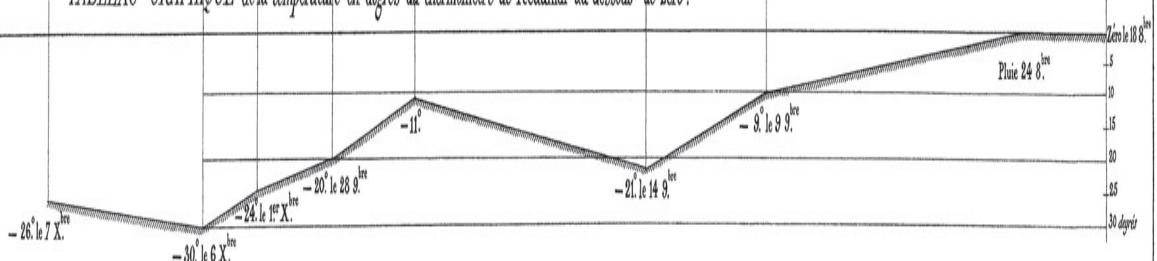


TABLEAU GRAPHIQUE de la température en degrés du thermomètre de Réaumur au dessous de zéro.

Les Cosaques passent au galop  
le Niémen gelé.

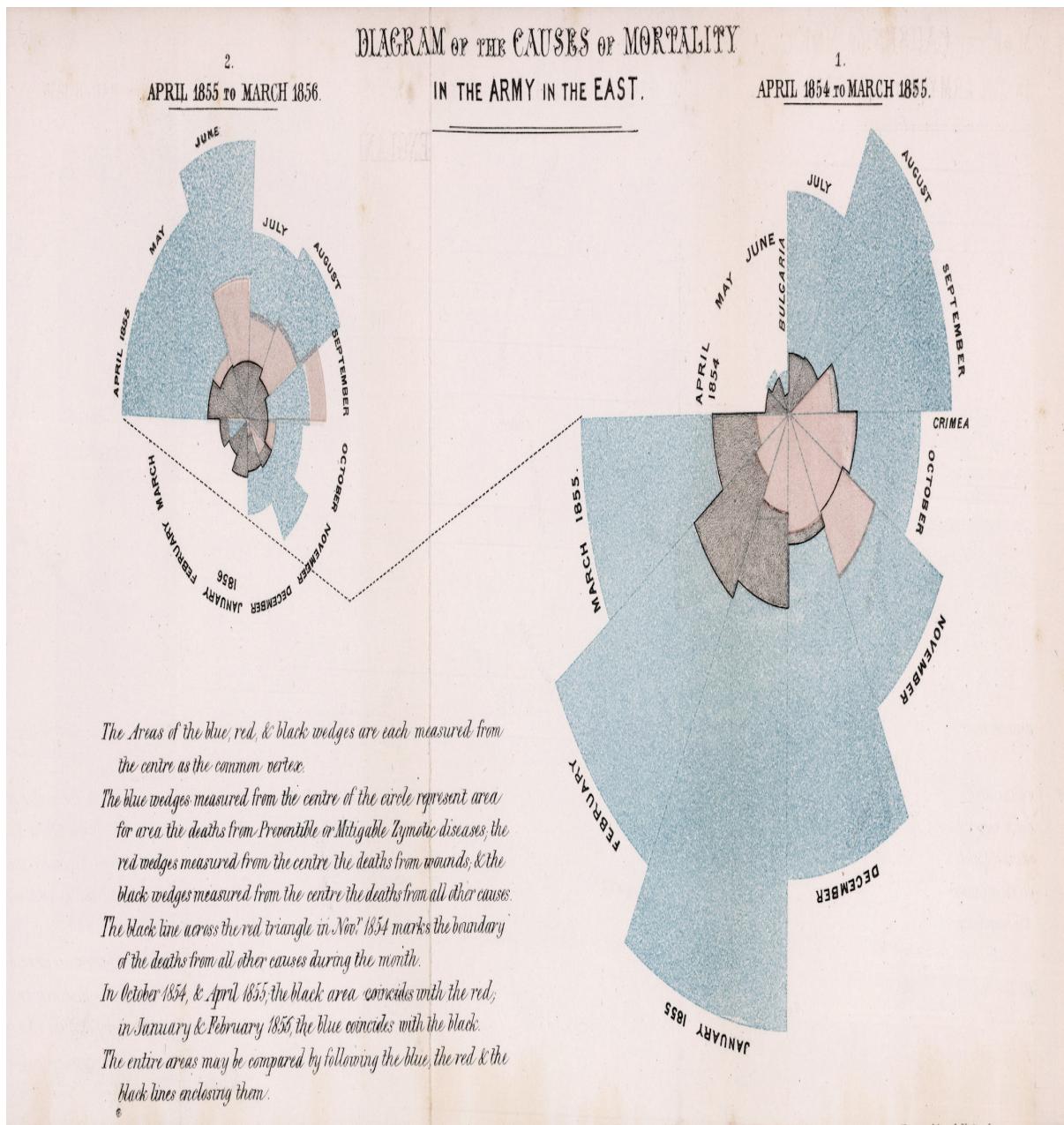


Analysé par Ruyer, à Paris, S<sup>e</sup> Marie S<sup>e</sup> G<sup>e</sup> à Paris.

Imp. J. Ruyer et Desordet.

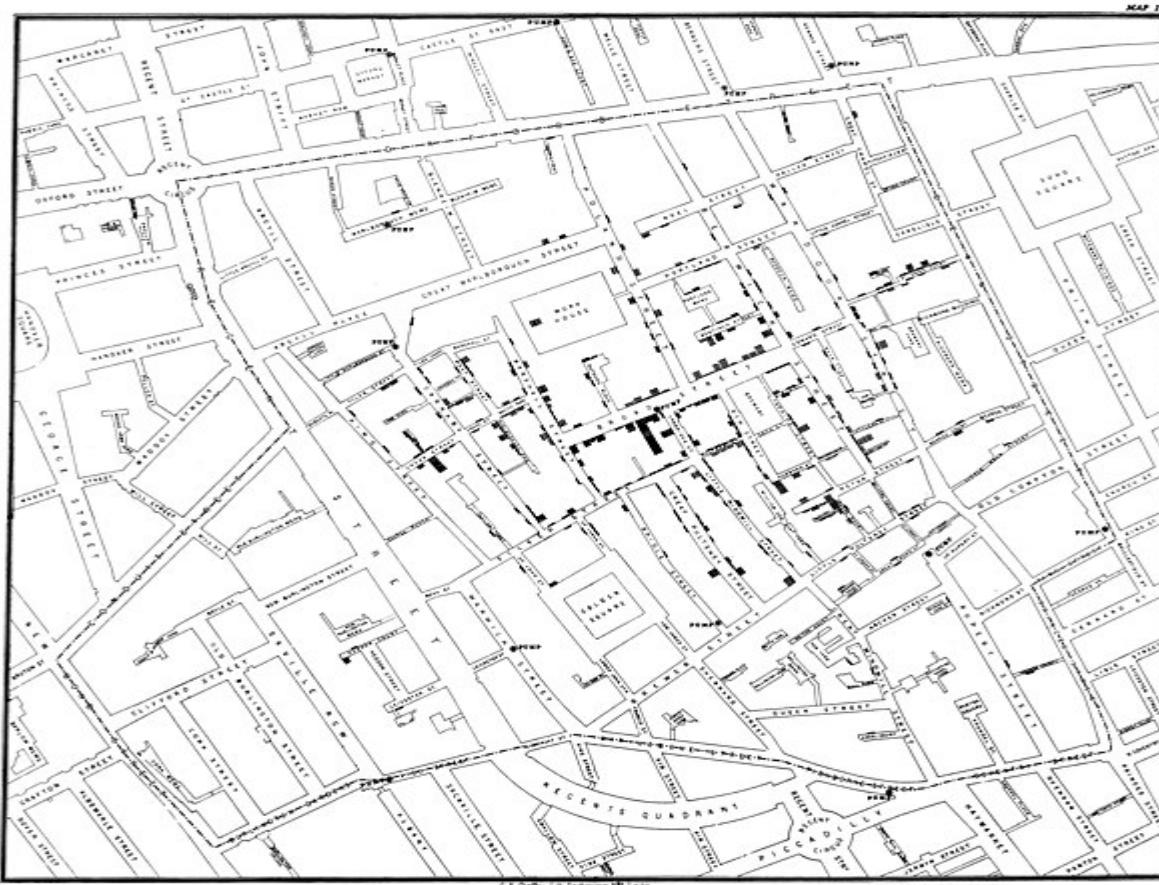
(Image taken from Wikipedia, go [here](#) for more information)

- Florence Nightingale, while investigating the causes of deaths among British troops in the Crimean war, produced the following area diagram:



(Image taken from Wikipedia, go [here](#) for more information)

- Finally, John Snow, while researching the 1854 London cholera epidemic, used the following map:



(Image taken from Wikipedia, see [here](#) for more information)

## Basic Graphs and Principles for Visualising Data

### Variables

A fundamental concept for any type of data analysis is the *random variable* or just *variable*. We typically view the data as the outcome of an experiment or an event with *uncertainty*. For example if we measure the highest temperature of a city on different days, we would expect a different outcome each day, which we will not be able to know for sure unless we measure it. In fact, even if we measure it, we may still not know it precisely as there is measurement error, but this is often assumed to be negligible. For another example, consider a ballot cast in an election between two candidates (A or B). Again, we will not be able to know if the vote was for candidate A or B (or none of the two), until we examine it.

A key task in data science, and other related disciplines, is to identify patterns in the data that can potentially help us make more informed decisions. The term *patterns* in the case of variables could be interpreted as identifying that some of values of the variable appear more often than others or, in higher level of complexity, certain values of some variables appear more often together with certain values of some other variables. As mentioned earlier *exploratory analysis* is the first step towards identifying such patterns and *data visualisation* is one of its key components (with others being based on numerical summaries and tables).

Before we proceed it is important to establish a clear understanding of what a variable is and what are its possible types. For a rigorous mathematical definition one is referred to probability theory but the following are sufficient for the purposes of this course:

- **Continuous variables:** They refer to quantities that can be measured, at least in theory, to several decimal places. Standard examples refer to a person's height, weight, income etc. An interesting feature here is that different levels of precision are possible. For example one may weigh 72 kilograms but with a more accurate scale it could be that the weight is actually 72.1 kilograms, with an even more accurate 72.12 and so on. So, it may make more sense to talk about intervals of their values, e.g. by saying weight of 72 to actually mean between a weight between 71.5 and 72.5 rather than exactly 72.
- **Categorical ordinal variables:** They refer to quantities that take values in some pre-specified categories than can be ordered in some way. A typical example is a person's income level, which can perhaps be classified as low, medium or high. The values of a categorical variable can still be numeric, e.g. when looking at the opinion of an individual on a specific argument, the potential values of the variable may be 1: completely agree, 2: somewhat agree, 3: neither agree nor disagree, 4: somewhat disagree, 5: completely disagree. Quite often, people treat categorical ordinal variables as continuous. While this is not recommended, due to the choice of the numerical values being arbitrary (we could have used  $1, 4, \lfloor \log 100 \rfloor, \lfloor \sqrt{35} \rfloor, 1000$  in the previous example), it may provide a useful approximate approach in some cases.
- **Categorical nominal variables:** They are similar to the categorical ordinal variables but their categories cannot be ordered (or it is not desirable to order them). Examples include gender or country of origin. Treating such a variable as continuous is definitely not recommended.

Exploratory analysis initially requires the following:

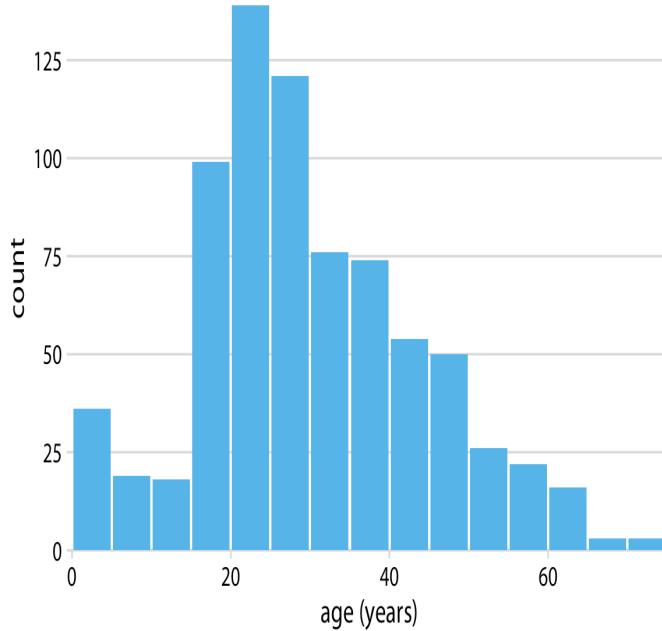
- Identify the **key variables** of interest in the data. It is essential that these are chosen after careful thought about the nature of the problem and aims of the project.
- Determine the **scale** of each of these variables. They could be continuous (measurable), categorical with ordered categories (ordinal scale) or they could be categorical with unordered categories (nominal scale). Specific graphs are suitable for each of these scales.

Once the above have been decided, data visualisation proceeds by looking at each of these variables **individually** or checking some of them together, mostly in pairs, to explore potential **associations** between them.

Next, we outline the main types of graphs for different types of variables. Note that the list of graphs mentioned in by no means exhaustive; we will see more in the weeks to follow. Note also that exploratory analysis can also be done without visualisation, more specifically with numerical summaries as well as tables. Nevertheless these are beyond the scope of this course.

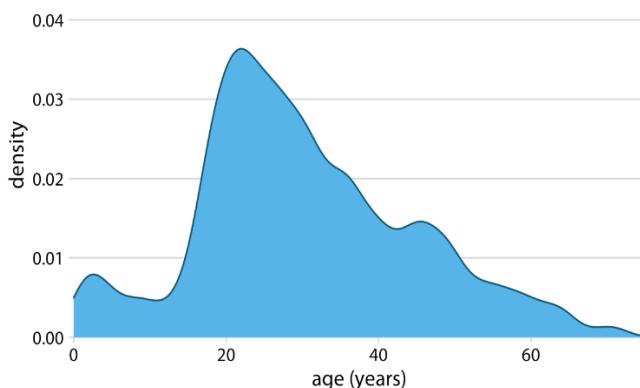
## Graphical Displays for a Single Variable

If a variable is continuous, it is of interest to determine which particular intervals of their values are more or less likely. The most common graphical display for this task is the *histogram*, an example of which is shown below:



Histogram of the ages of Titanic passengers; Figure 7.1 taken from Wilke (2017)

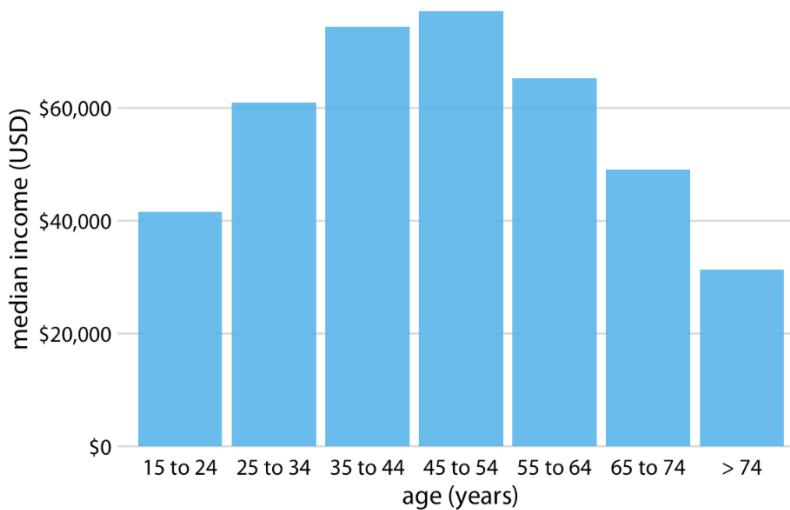
We see that people aged between 20 and 30 appear in higher proportions although other ages are also present. A smoother version of a histogram is offered by a kernel density plot, shown below for the previous example:



Kernel density plot of the ages of Titanic passengers; Figure 7.3 taken from Wilke (2017)

Other examples of graphs for continuous variables are *boxplots* and *violin plots*

For categorical ordinal variables the standard option is a bar plot. Below is an example depicting information on the United States annual household income versus age group in 2016, provided by United States Census Bureau.



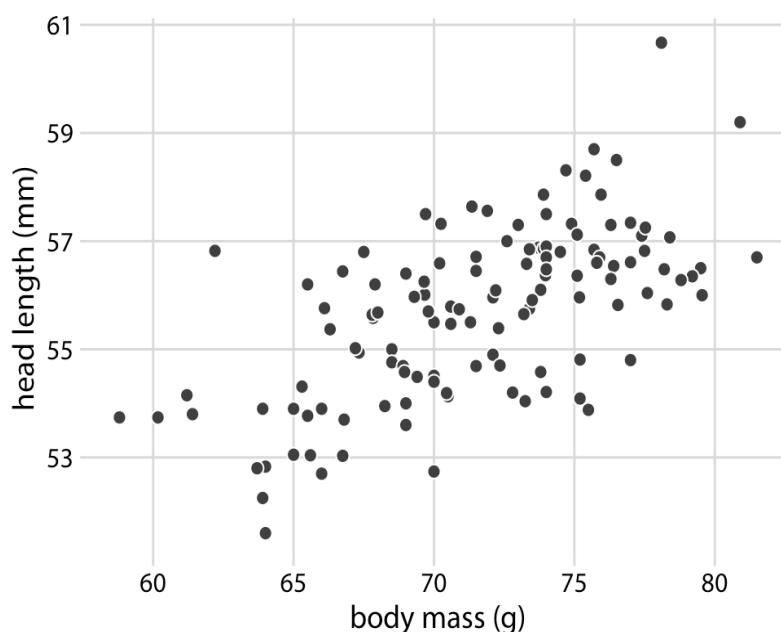
Bar plot on US annual household income versus age group in 2016. Figure 6.5 taken from Wilke (2017)

Note that the above appears to be very similar to a histogram. However a key difference is that the bars are not joined, thus breaking the continuum between the different categories. Actually in the histogram above the bars are not joined either but this is done for stylistic purposes only; they are typically meant to be joined. In categorical ordinal variables it is also essential to respect the order of the categories in the x-axis.

Bar plots can also be used for categorical nominal variables. In this case the order of values in the x-axis can be determined by their corresponding y-values, so that these are in increasing or decreasing order. An alternative graphical summary is the pie chart, an example of which was shown earlier via the work of Playfair.

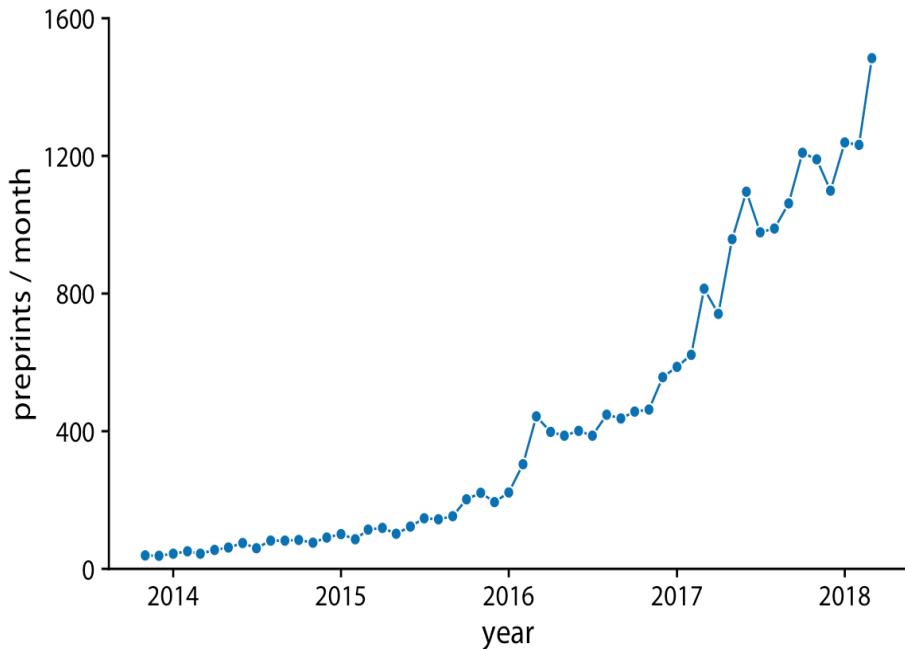
## Graphical Displays for Two Variables

As before, different types of plots are used depending on the types of the two variables involved. For the case of two continuous variables, the default choice is a scatter plot. Below we see an example from a dataset, consisting of measurement on 123 blue jays. There seems to be some tendency for heavier birds to have longer heads.



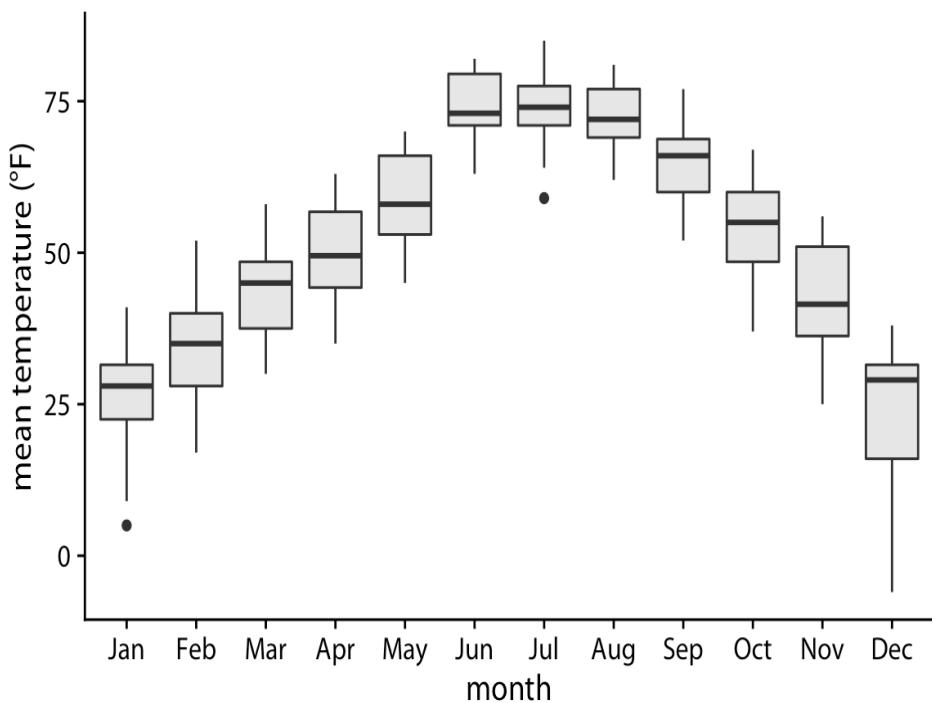
Head length (in millimetres) versus body mass (in grams); Figure 12.1 taken from Wilke (2017)

An interesting special case is when one of the two variables may be viewed as time. In this case there is a specific temporal order in the data and quite often we want to visualise it. This can be done using *line graphs*. Below is an example of a line graph consisting of the number of monthly submissions to bioRxiv, between November 2014 and April 2018.



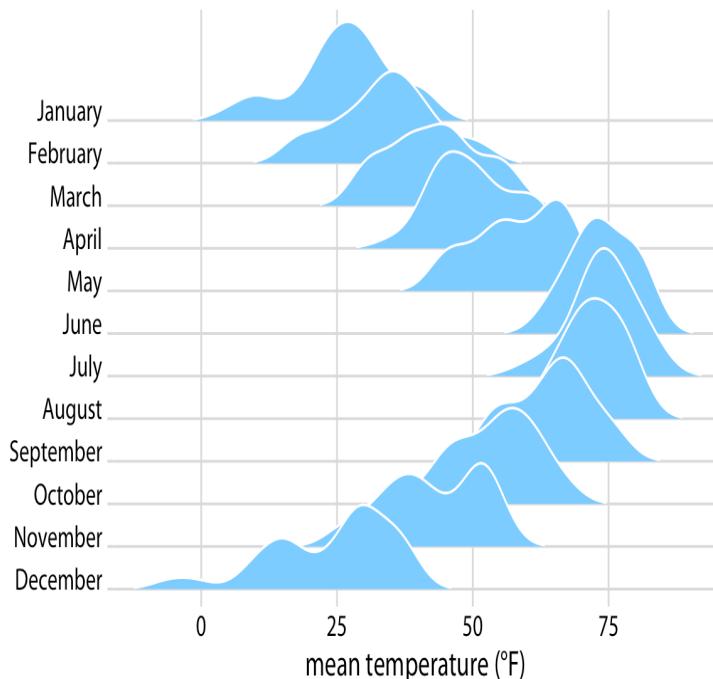
Number of submissions per month to bioRxiv between November 2014 and April 2018; Figure 13.2 taken from Wilke (2017)

In cases of a pair between a continuous variable  $\langle y \rangle$  and a categorical variable  $\langle x \rangle$ , it is useful to align separate plots summarising the  $\langle y \rangle$  variable for each of the categories of the variable  $\langle x \rangle$ . A simple option, often adopted in practice is to provide a single summary of the  $\langle y \rangle$  variable, such as its mean, for each of the  $\langle x \rangle$  categories. This can be done via a boxplot as in the previous example of the United States annual household income versus age group in 2016. Another option, providing more information on the entire distribution of  $\langle y \rangle$  across the categories of  $\langle x \rangle$ , is by separate boxplots. Below, we provide an example on data consisting of mean daily temperatures recorded in Lincoln, Nebraska (variable  $\langle y \rangle$ ), for several days of different months (variable  $\langle x \rangle$ ):



Mean daily temperatures in Lincoln, Nebraska, visualized as boxplots; Figure 9.3 taken from Wilke (2017)

In the recent years boxplots tend to be losing ground over violin plots; examples of those will be shown over the next few weeks. It is also possible to produce *heat maps* or to align kernel density plots across different categories of the variable  $\langle x \rangle$ , the so called *ridgeline plots*. For the previous example, a ridgeline plot is shown below:



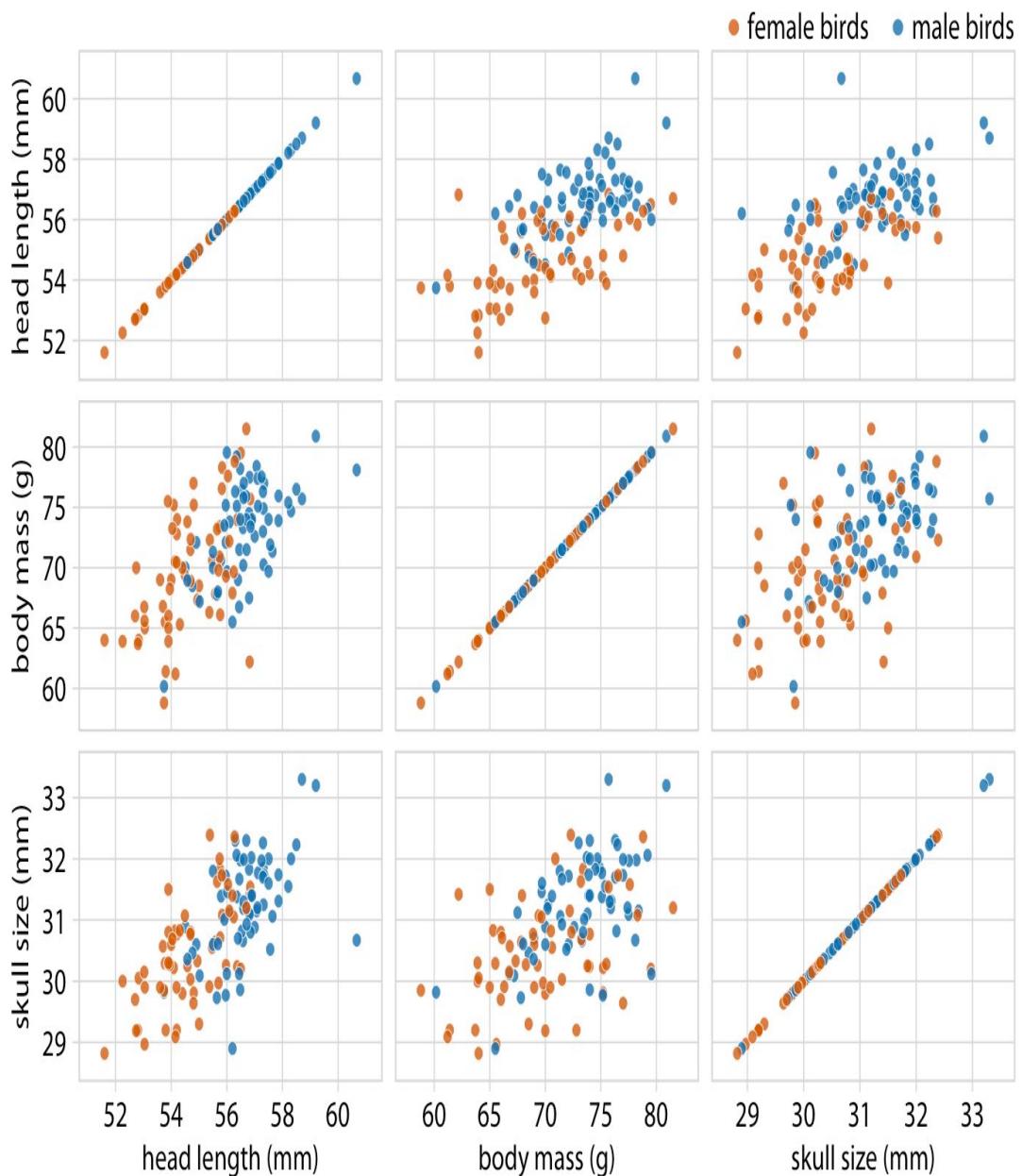
Mean daily temperatures in Lincoln, Nebraska, visualized as a ridgeline plot; Figure 9.9 taken from Wilke (2017)

Clearly, the above plot reveals more information compared to using boxplots. For example, with the kernel density plots we can see the two peaks around 35 and 50 degrees Fahrenheit in November, which were not visible with boxplots.

It is important to note at this point that association does not imply causation. Two variables may appear to be associated but this does not mean that either of them has a direct effect on the other. Quite often, we may observe patterns of association that are caused by other, known or unknown, variables. This effect is also known as *confounding* and it can cause what is often termed as *spurious correlation*.

## Graphical Displays For Three or More Variables

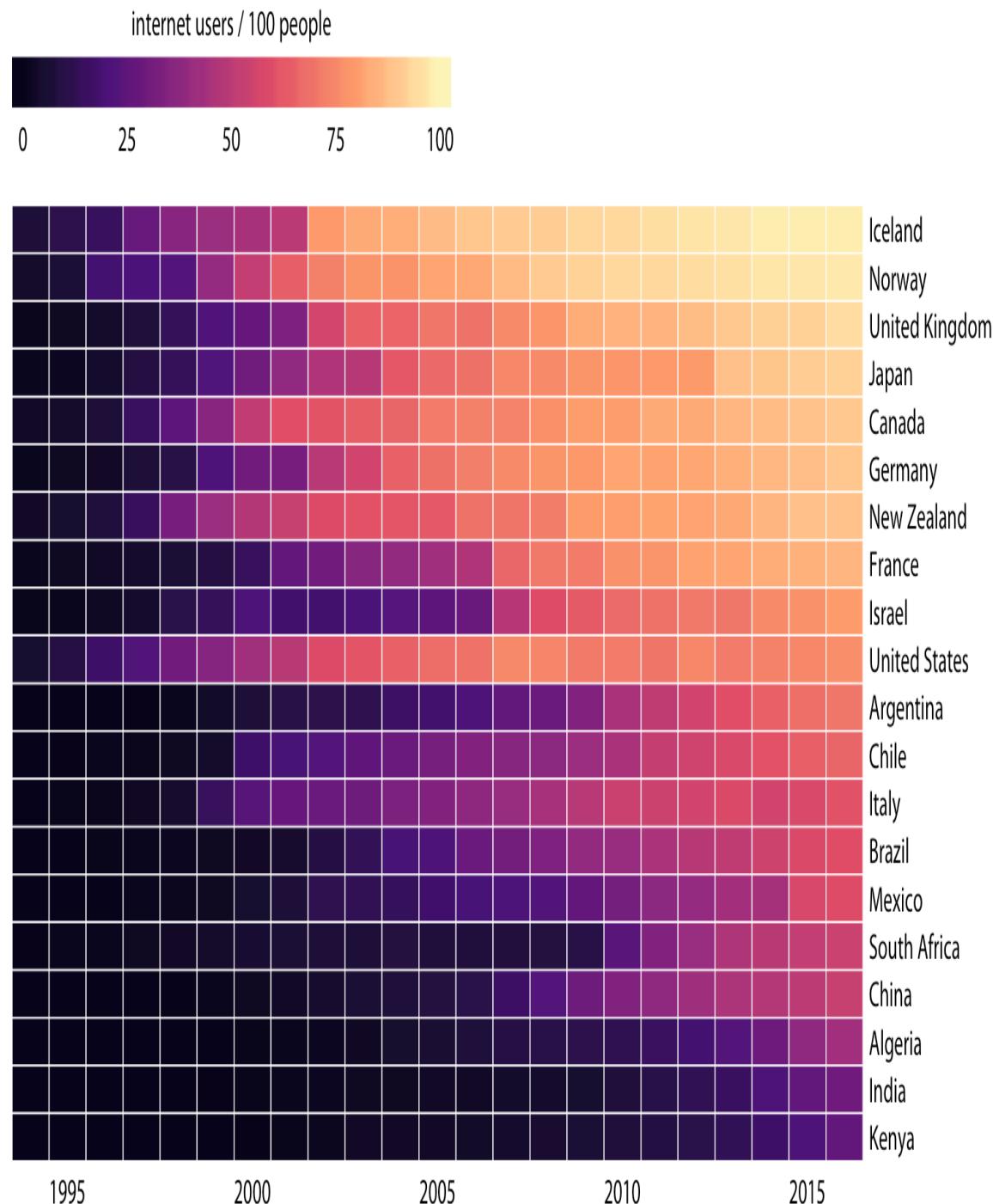
As before we can classify different types of plots based on the types of variables involved. For cases where we have three or more continuous variables we can use *matrix scatter plots*. As an example, we expand on the scatter plot on blue jays by also looking at skull size (in millimetres), in addition to head length (in millimetres) versus body mass (in grams), for each bird.



Matrix scatter plot matrix of head length, body mass, and skull size, for 123 blue jays; Figure 12.4 taken from Wilke (2017)

The figure above actually goes one step further by assigning different colours to each point depending on the sex of the bird; this is actually a case of three continuous and one categorical variable. Looking at the plots we can identify that male birds tend to be bigger and heavier.

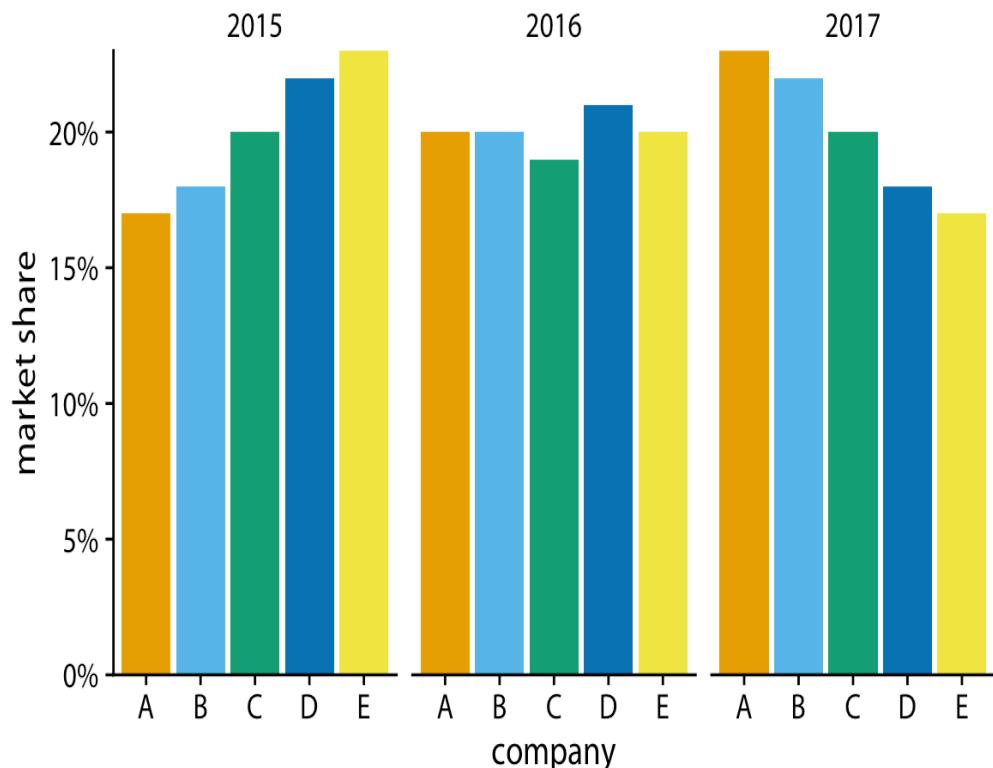
When two categorical variables are involved, an alternative to barplots that map data values onto bar heights is to map data values to colours; in other words to use a *heat map*. Below we see an example on the percentage of internet users across time, from 1994 to 2016, and 20 countries.



Internet use across countries and time; the colour reflects the percentage of internet users for each country and year combination; Figure 6.14 taken from Wilke (2017)

The above graph can provide information for things like identifying the countries where internet use began early, the countries with high internet use in 2016, etc.

Finally, for situations involving three categorical variables, *grouped or stacked barplots* may be used. An example on is shown below in order to examining the market share of five companies (A,B,C,D,E) from 2015 to 2017.



Market share corresponding to five companies for the years 2015-17; Figure 10.6 taken from Wilke (2017)

From the plot above we can see that companies A and B seem to be increasing their market share unlike companies D and E that seem to be losing ground. Other options for graphs of several categorical variables include *mosaic plots*, *treemaps* and *nested pies*.

## Useful Links and Resources

- [Fundamental of data visualisation](#): An easy to read book with many examples of good practice for producing graphs.

## References

Wilke, C. O. (2017). *Fundamentals of data visualization*. O'Reilly Media.

# Graphics and Data Visualization in R



## Data

We will illustrate in R most of the visualizations introduced this week on the `airquality` dataset. The data consist of daily air quality measurements in New York air from May to September 1973 (153 days). The data were obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

In each of the 153 days, the following four variables were measured

Variable	Description
Ozone	Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
Solar.R	Solar radiation in Langleys in the frequency band 4000–7700 angstroms from 0800 to 1200 hours at Central Park
Wind	Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
Temp	Maximum daily temperature in degrees Fahrenheit at La Guardia Airport

The month and the day of the month were also recorded. The data are available each time you open RStudio. Below we copy them to the data frame `df` and inspect the first six rows.

```
df=airquality
head(df)
   Ozone Solar.R Wind Temp Month Day
1     41     190  7.4   67      5    1
2     36     118  8.0   72      5    2
3     12     149 12.6   74      5    3
4     18     313 11.5   62      5    4
5     NA      NA 14.3   56      5    5
6     28     NA 14.9   66      5    6
```

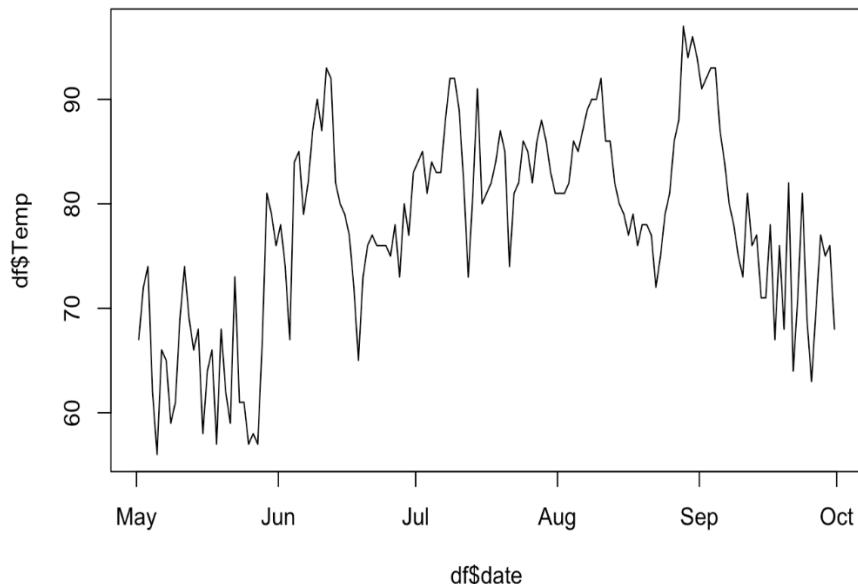
For plots over time and other related operations it is useful to create a date variable from the variables Month and Day. This can be done with the R function `ISOdate` as shown below:

```
df$date <- ISOdate(1973, df$Month, df$Day)
```

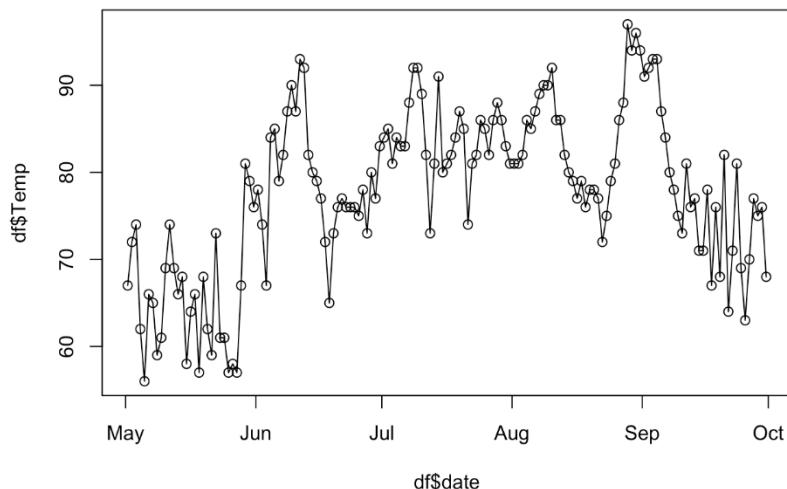
# Line and Scatter Plots

For line and scatter plots the function `plot()` can be used. This is a key function for the material of this week; next week we will work with `ggplot()`. For line plots the argument `type` can be set to `l` (lines only) or `o` (lines and points).

```
plot(df$date, df$Temp, type='l')
```



```
plot(df$date, df$Temp, type='o')
```

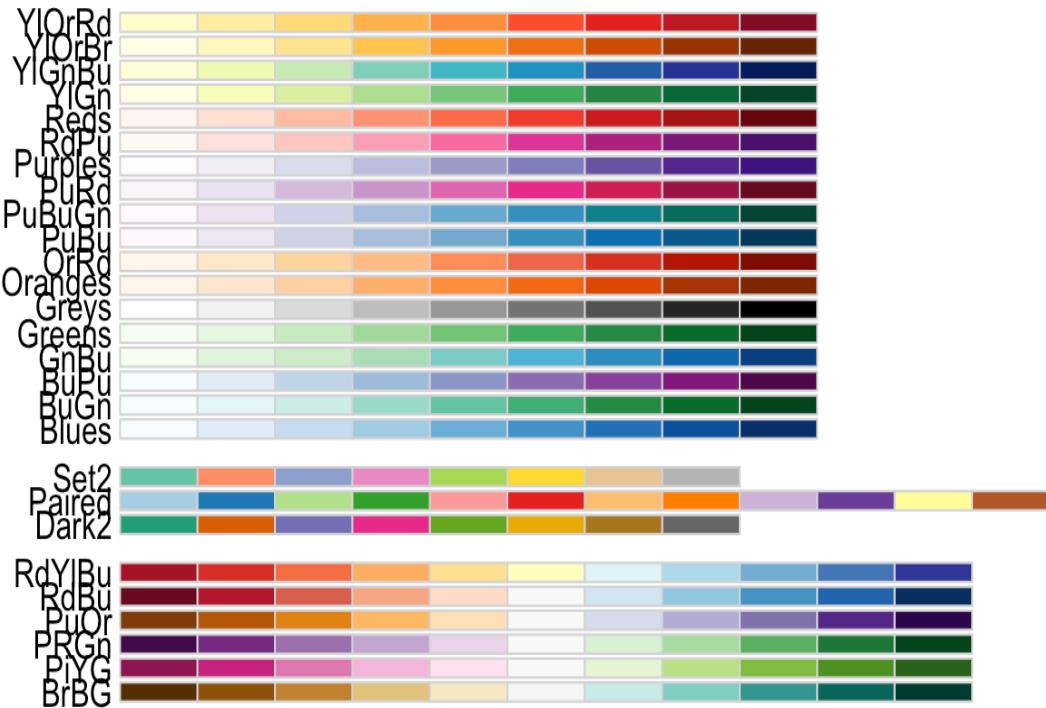


Next we present some graphical parameters that are also relevant for other visualizations such as colours, legends, titles, axis labels, types of lines and points.

## Graphical Parameters

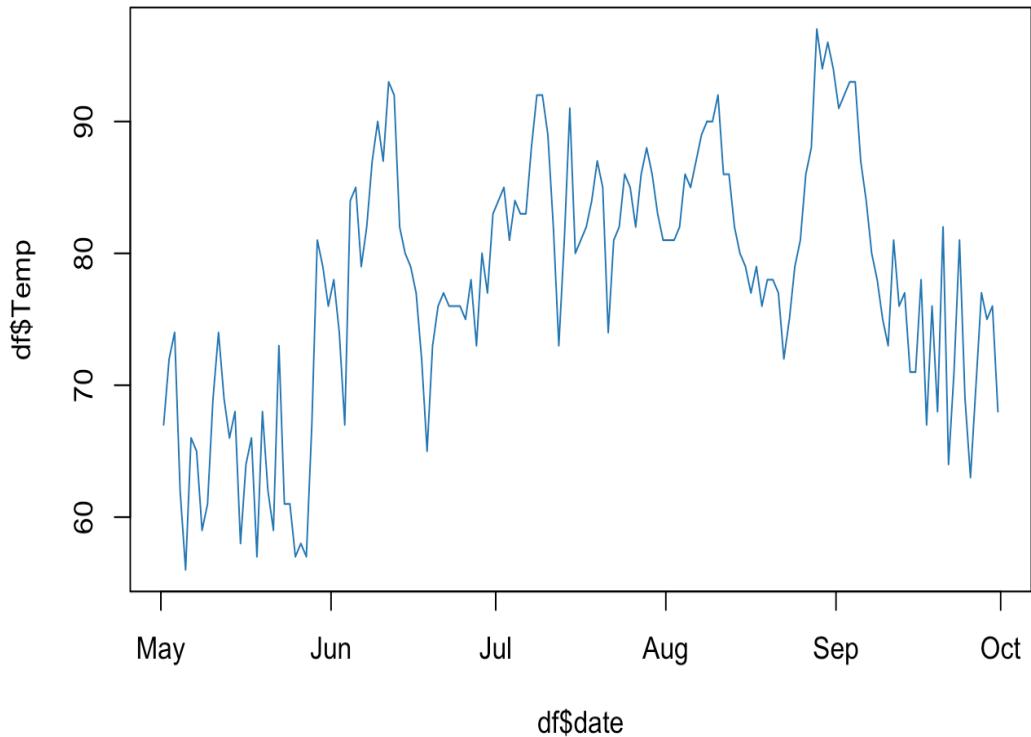
To add colours we recommend using colour-blind friendly palettes; see [here](#) for more on accessibility. One way to do that is via the **RColorBrewer** package. The code below loads this package and displays its colour-blind friendly palettes:

```
library("RColorBrewer")
display.brewer.all(colorblindFriendly = TRUE)
```



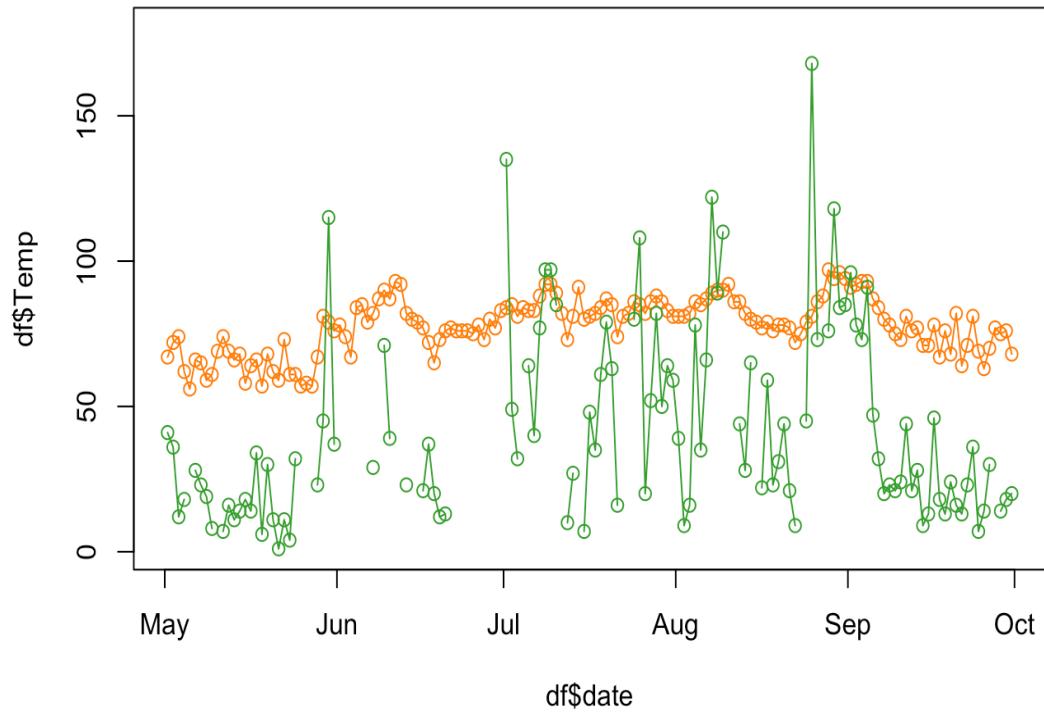
Below is an example of using the `Paired` palette that has 12 colours. Every element of the object `colours` corresponds to each of these 12 colours.

```
colours<-brewer.pal(n = 12, name = "Paired")
plot(df$date,df$Temp,type="l",col=colours[2])
```



Next we illustrate how we can add more than one lines to the plot with function `lines()` (the function `points()` does the same with just points). The argument `ylim` specifies the range of the y-axis (there is also `xlim` for the x axis).

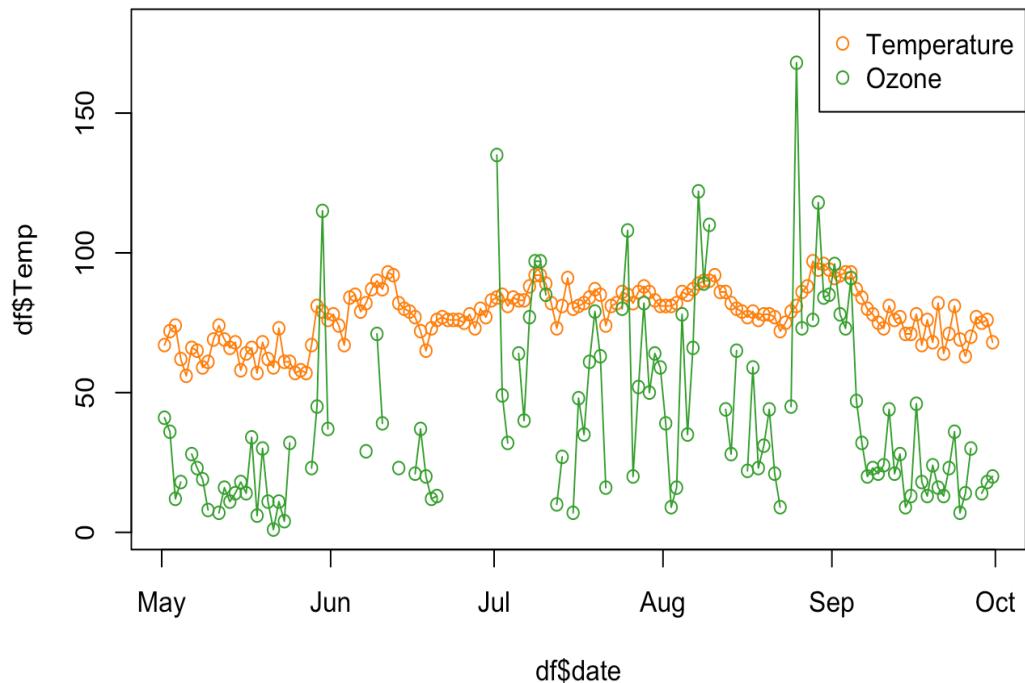
```
plot(df$date, df$Temp, type='o', ylim=c(1,180), col=colours[8])
lines(df$date, df$Ozone, type='o', col=colours[4])
```



There are some gaps in the green line above, due to the missing values in the `Ozone` variable. Also it is essential to use the `ylim` argument because the limits in the y-axis set by the `plot()` in the first line of code are based on `Temp` and are therefore too narrow for `Ozone`. So, without `ylim` the `Ozone` line will go out of bounds.

A legend can also be provided to display more information, i.e. inform which colour corresponds to each variable. The function `legend()` typically require the location, labels (of the different lines), colours and characters corresponding to points. For the points, different characters are possible with the argument `pch`.

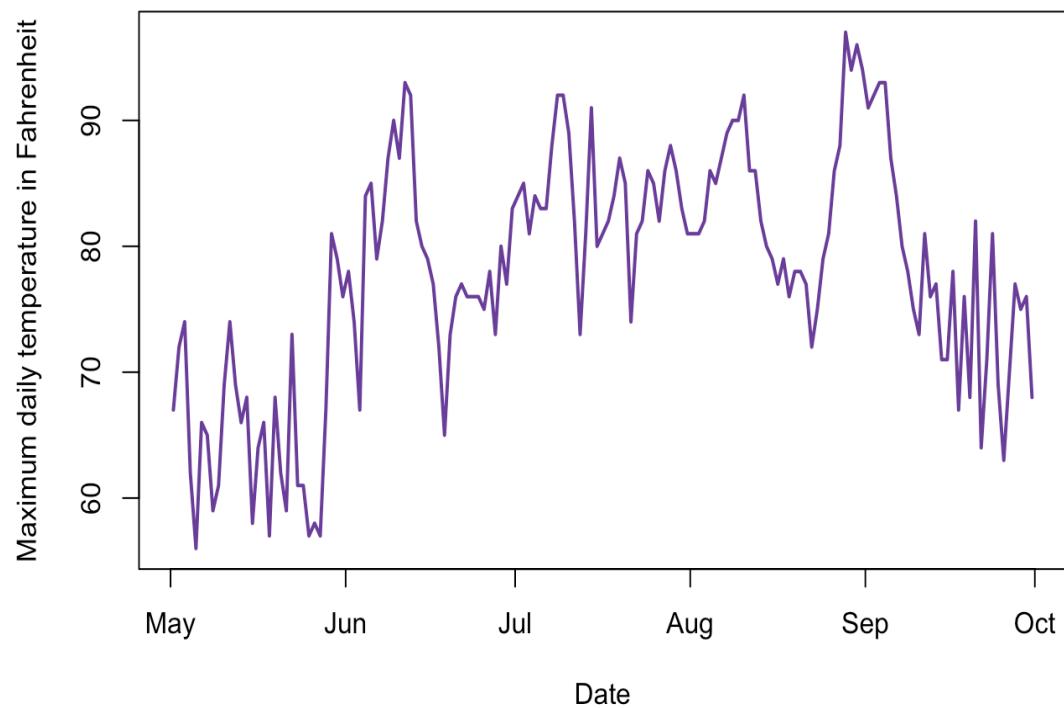
```
plot(df$date, df$Temp, type='o', ylim=c(1,180), col=colours[8])
lines(df$date, df$Ozone, type='o', col=colours[4])
legend('topright', legend=c('Temperature', 'Ozone'), col=colours[c(8,4)],
pch=1)
```



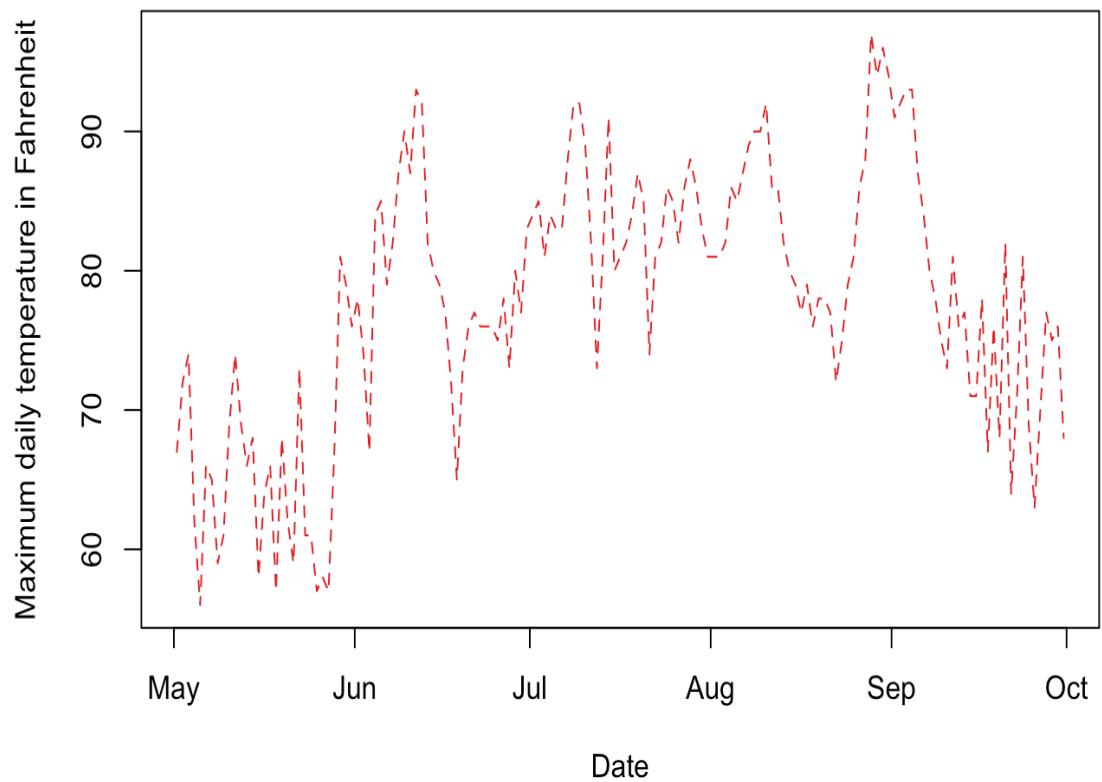
A title in the plot can be added by the argument `main`. Axes labels can be added with the arguments `xlab` and `ylab` for x and y axes respectively. The style of the line can be controlled by the argument `lty`, its thickness by `lwd`. Some examples are given below.

```
plot(df$date, df$Temp, type='l', lwd=2, xlab='Date', ylab='Maximum daily
temperature in Fahrenheit', main='Temperatures in New York
City', col=colours[10])
```

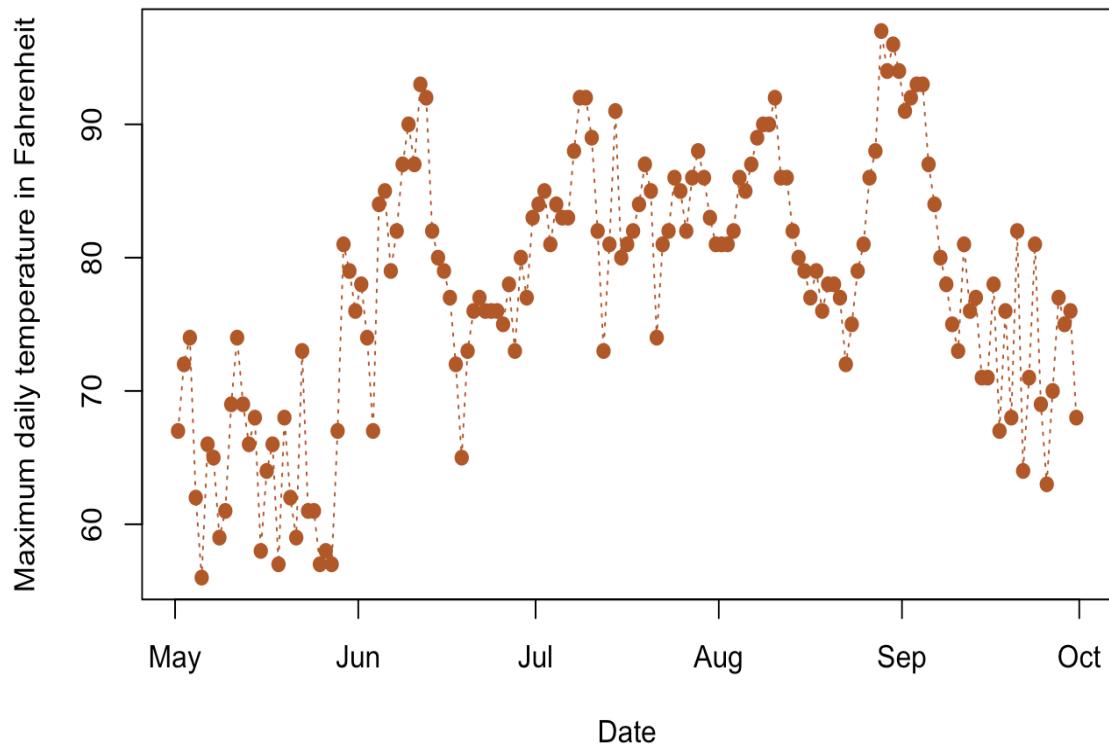
## Temperatures in New York City



```
plot(df$date,df$Temp, type='l',lty=2,xlab='Date',ylab='Maximum daily temperature in Fahrenheit',col=colours[6])
```



```
plot(df$date, df$Temp, type='o', lty=3, pch=19, xlab='Date', ylab='Maximum daily temperature in Fahrenheit', col=colours[12])
```

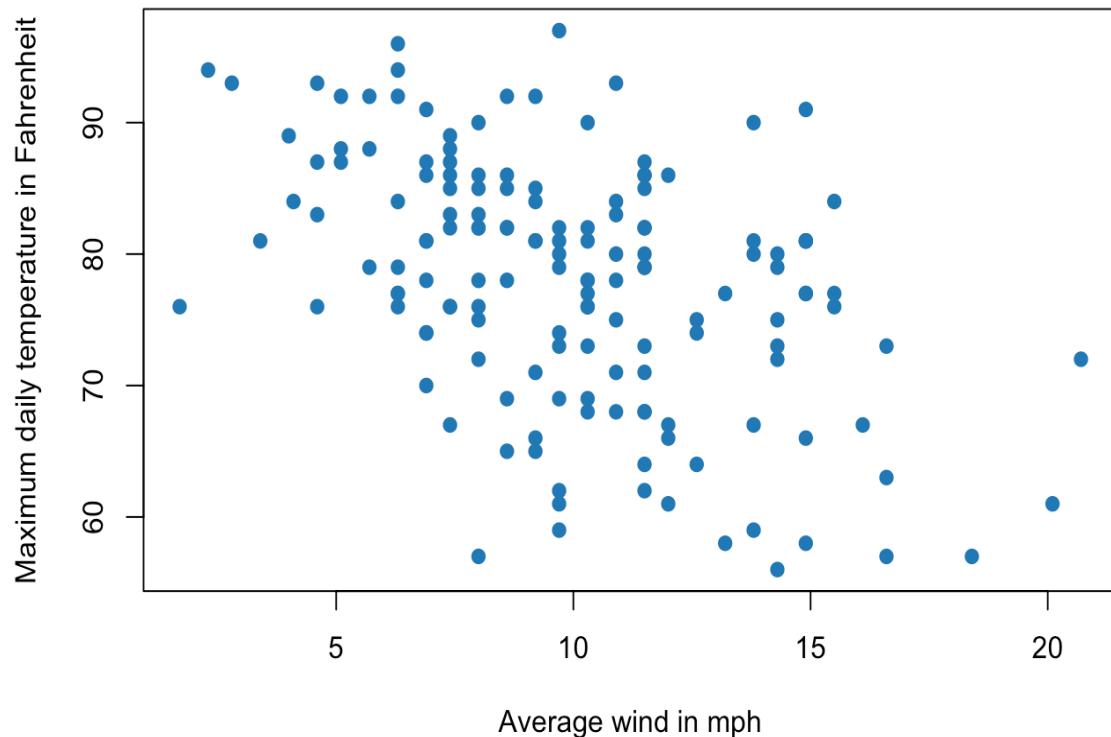


## Labelling points

In cases of checking associations between two continuous variables, scatter plots provide a standard option. They can still be produced using the `plot()` function:

```
plot(df$Wind, df$Temp, pch=19, xlab='Average wind in mph', ylab='Maximum daily temperature in Fahrenheit', main='Temperature vs Wind in New York', col=colours[2])
```

## Temperature vs Wind in New York



The plot above reveals a negative association between wind and temperature as expected.

In some cases it may be informative to label points belonging in different categories. For example, consider the binary variable depending on whether the measurement is between May and June or afterwards. This can be done with the code below. The `table()` command provides the frequencies corresponding to the categories of the binary variable `season`.

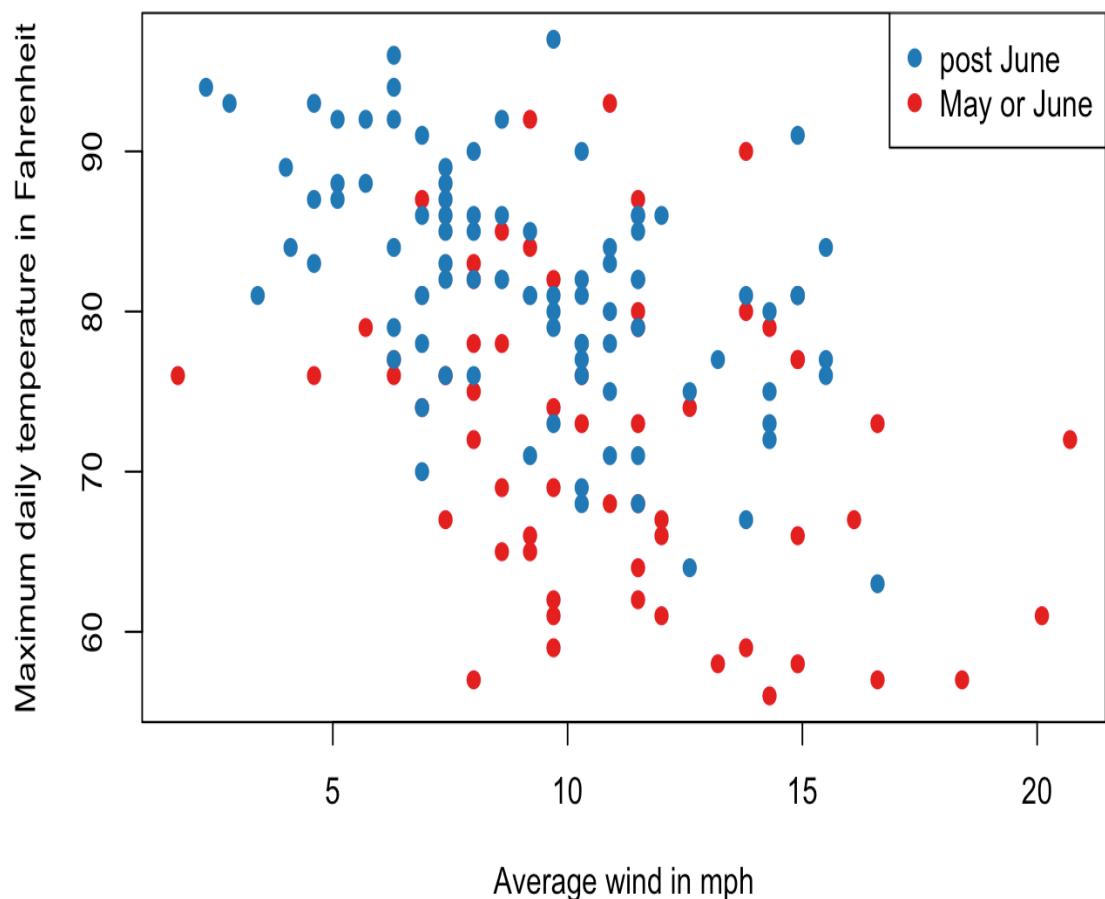
```
df$season[df$Month>6]<-1  
df$season[df$Month<=6]<-2  
table(df$season)
```

```
1 2  
92 61
```

The following code provides a scatter plot with different colours for the May and June measurements versus the other months. Note that we now select the colours from the palette explicitly for the two categories to be the second colour (blue) and the sixth (red).

```
colours<-brewer.pal(n = 12, name = "Paired")  
colours<-colours[c(2,6)]  
plot(df$Wind, df$Temp, pch=19, col=colours[df$season], xlab='Average wind in mph', ylab='Maximum daily temperature in Fahrenheit', main='Temperature vs Wind in New York')  
legend('topright', legend=c('post June', 'May or June'), col= colours, pch = 19)
```

## Temperature vs Wind in New York

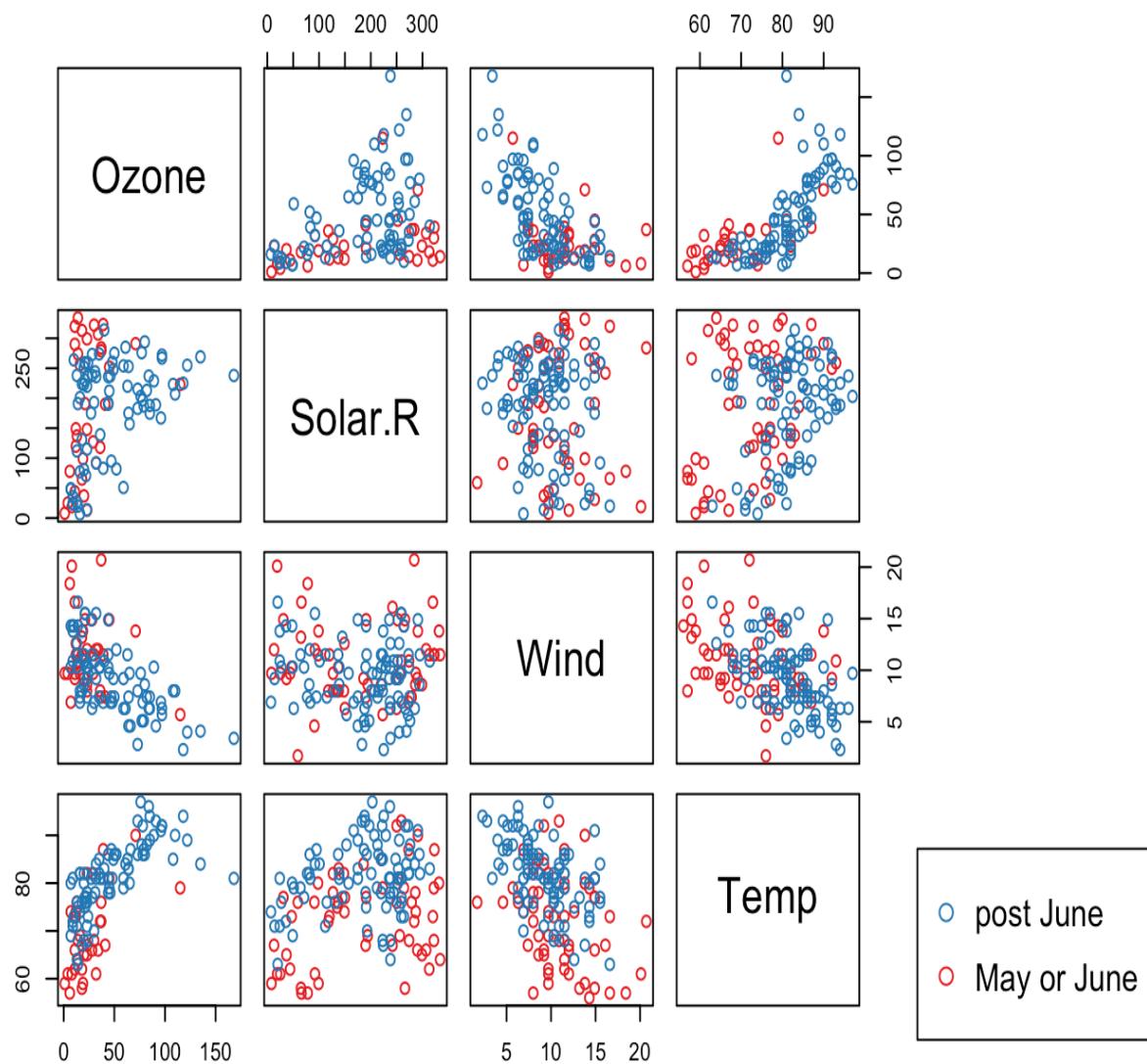


We can see from the plot above that after June the temperatures tend to be higher and the winds lower.

## Matrix Scatter Plots

In the presence of more than two continuous variables, a matrix scatter plot provides a good way to visualize all of them in pairs. The `pairs()` function can provide that. In order to place the legend outside of the plot we used the argument `oma` and used the line `par(xpd=TRUE)`.

```
pairs(df[1:4], col=colours[df$season], main='Matrix scatter plot of the air quality dataset', oma=c(3,3,3,15))
par(xpd = TRUE)
legend('bottomright', legend=c('post June', 'May or June'), col= colours,
pch = 1)
```

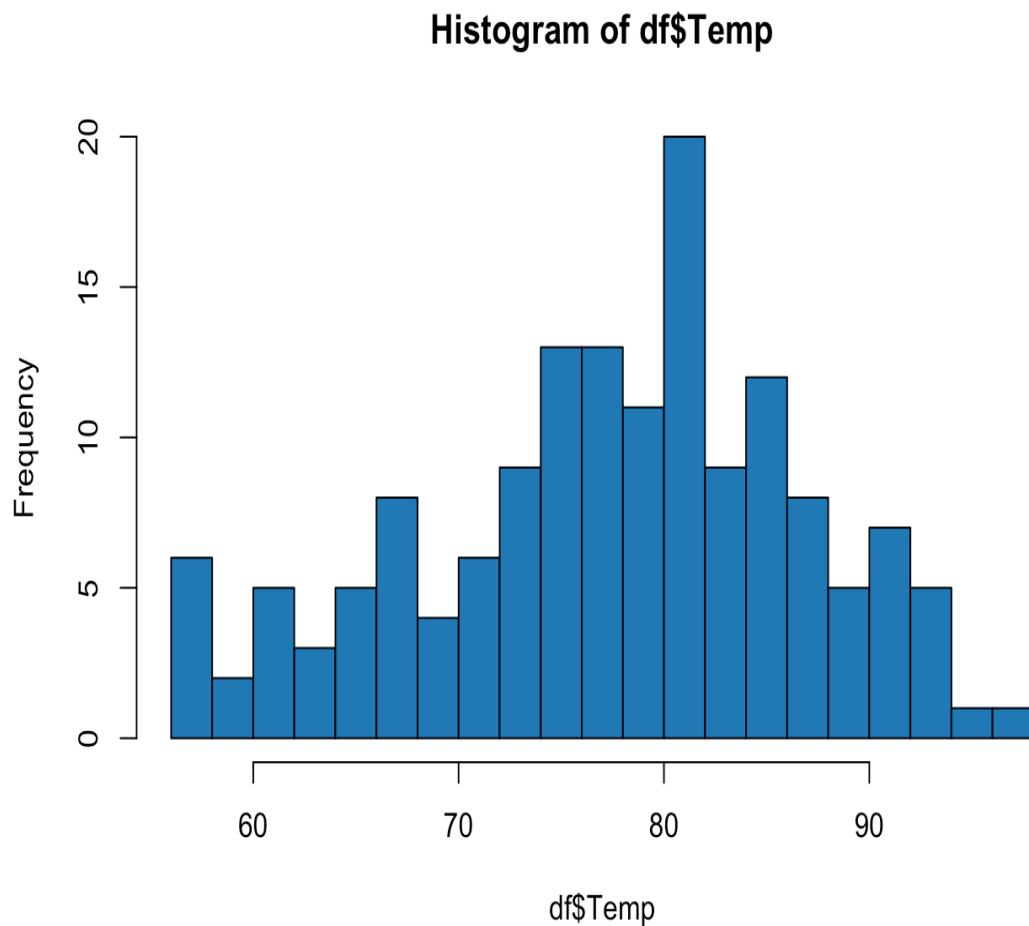


More details about the plot function and graphical parameters in general can be found [here](#).

# Histograms and Density Plots

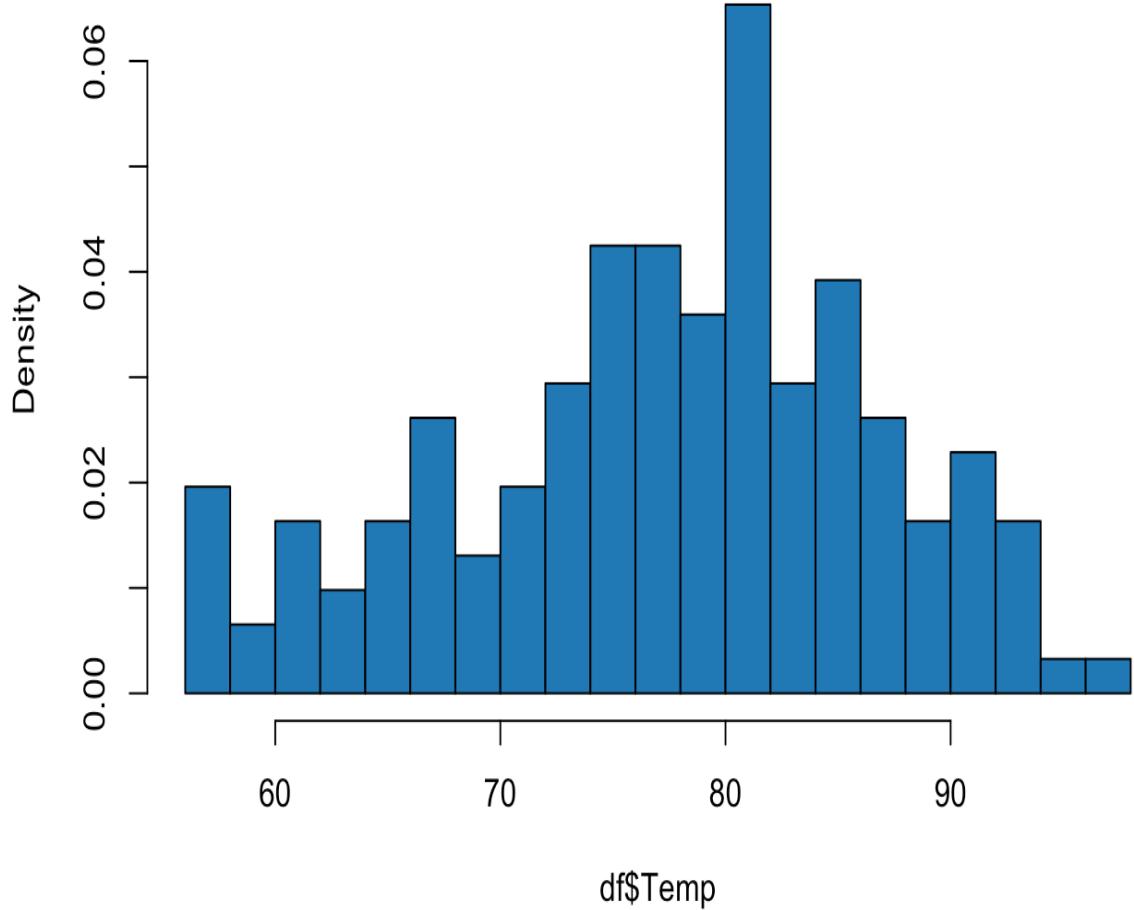
The distributions of a continuous variable can be portrayed with a histogram. Below we provide the code for constructing one. Using `freq=FALSE` results in a density histogram, otherwise a frequency histogram is obtained.

```
colours<-brewer.pal(n = 12, name = 'Paired')
hist(df$Temp, breaks=20, col=colours[2])
```



```
hist(df$Temp, breaks=20, col=colours[2], freq=FALSE)
```

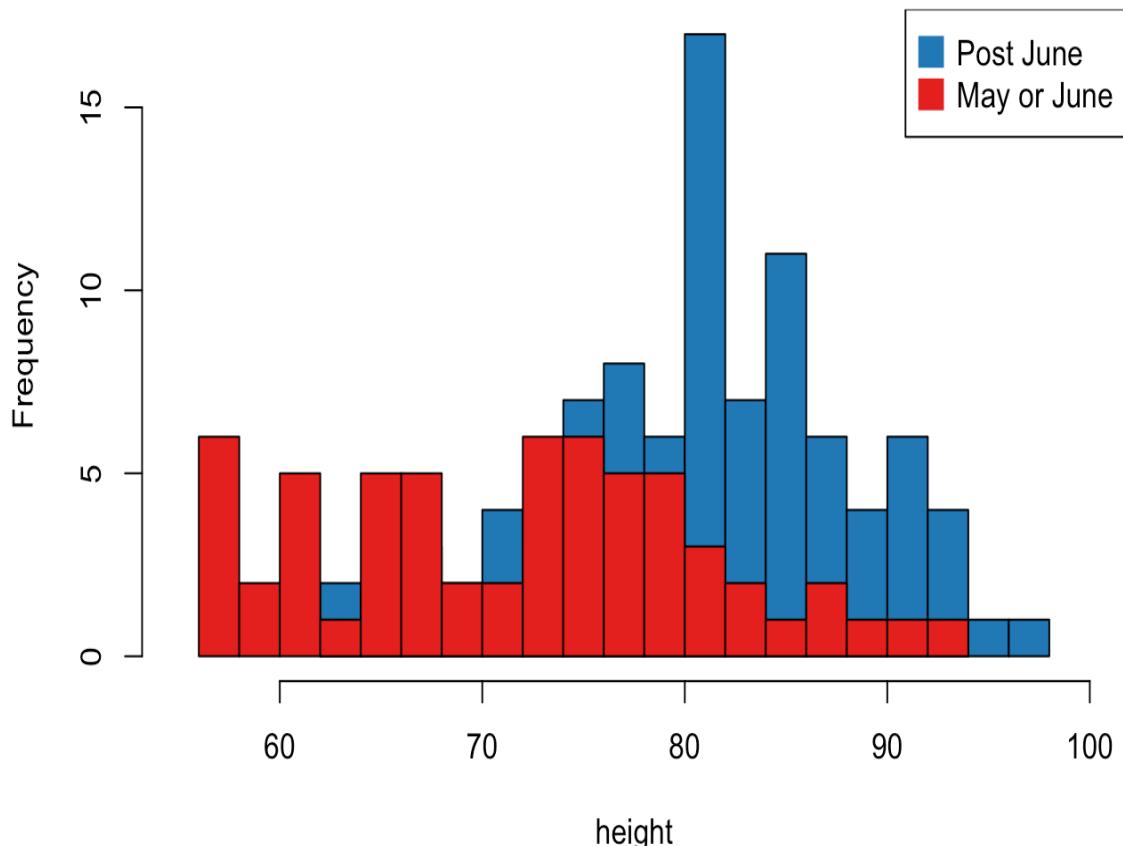
## Histogram of df\$Temp



We can also compare the distributions of two (or more) continuous variables by overlaying their histograms. In such cases it may be useful to set the limits of the x-axis with `xlim`.

```
hist(df$Temp[df$season==1], breaks=20, xlim=c(55,100), col=colours[2],  
xlab='height', main='Distribution of Temperature before and after June' )  
# Second with add=T to plot on top  
hist(df$Temp[df$season==2], breaks=20, xlim=c(55,100), col=colours[6],  
add=T)  
# Add legend  
legend("topright", legend=c('Post June', 'May or June'),  
col=colours[c(2,6)], pt.cex=2, pch=15)
```

## Distribution of Temperature before and after June

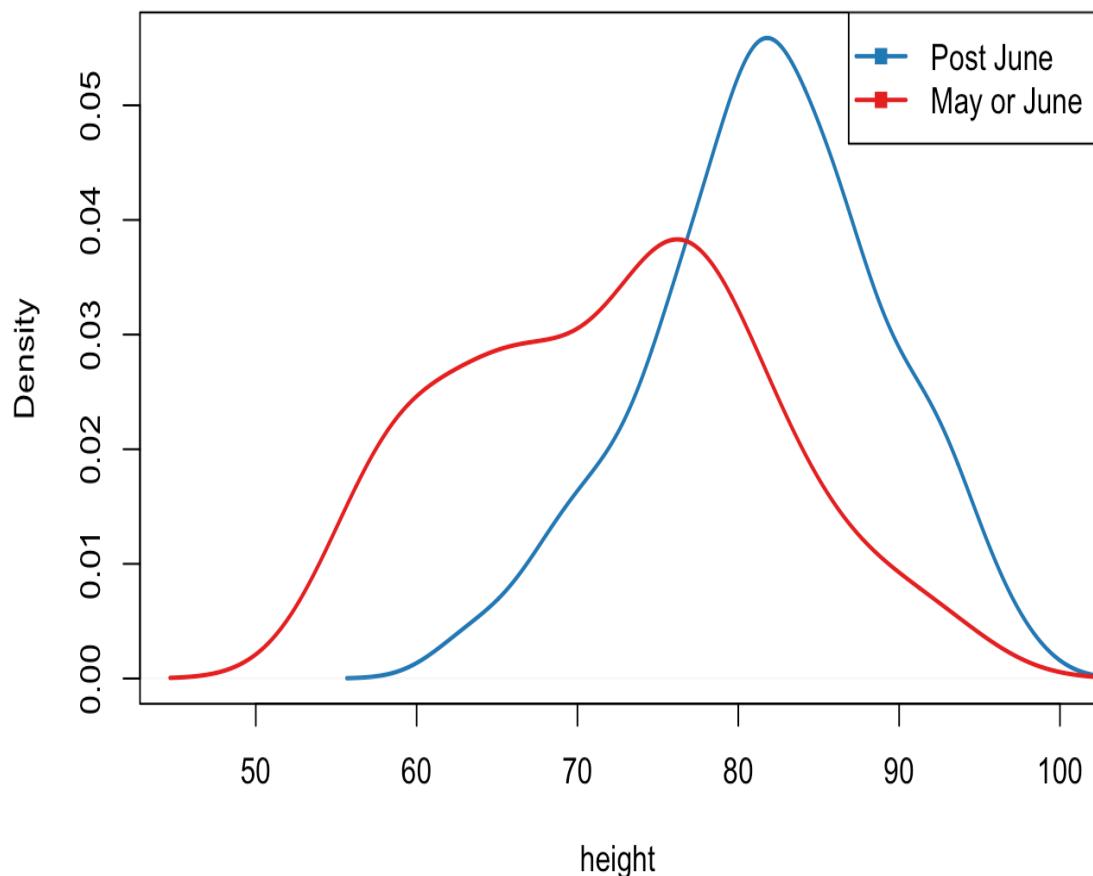


More details can be found [here](#).

A smoother alternative to the histogram is provided by a kernel density plot. This can be done by using the `density()` function inside `plot()` or `lines()`.

```
d1<-density(df$Temp[df$season==1])
d2<-density(df$Temp[df$season==2])
plot(d1,xlim=c(45,100), col=colours[2], lwd=2, xlab='height',
main='Distribution of Temperature before and after June')
lines(d2, xlim=c(45,100), col=colours[6], lwd=2)
legend('topright', legend=c('Post June','May or June'),
col=colours[c(2,6)], lwd=2, pch=15)
```

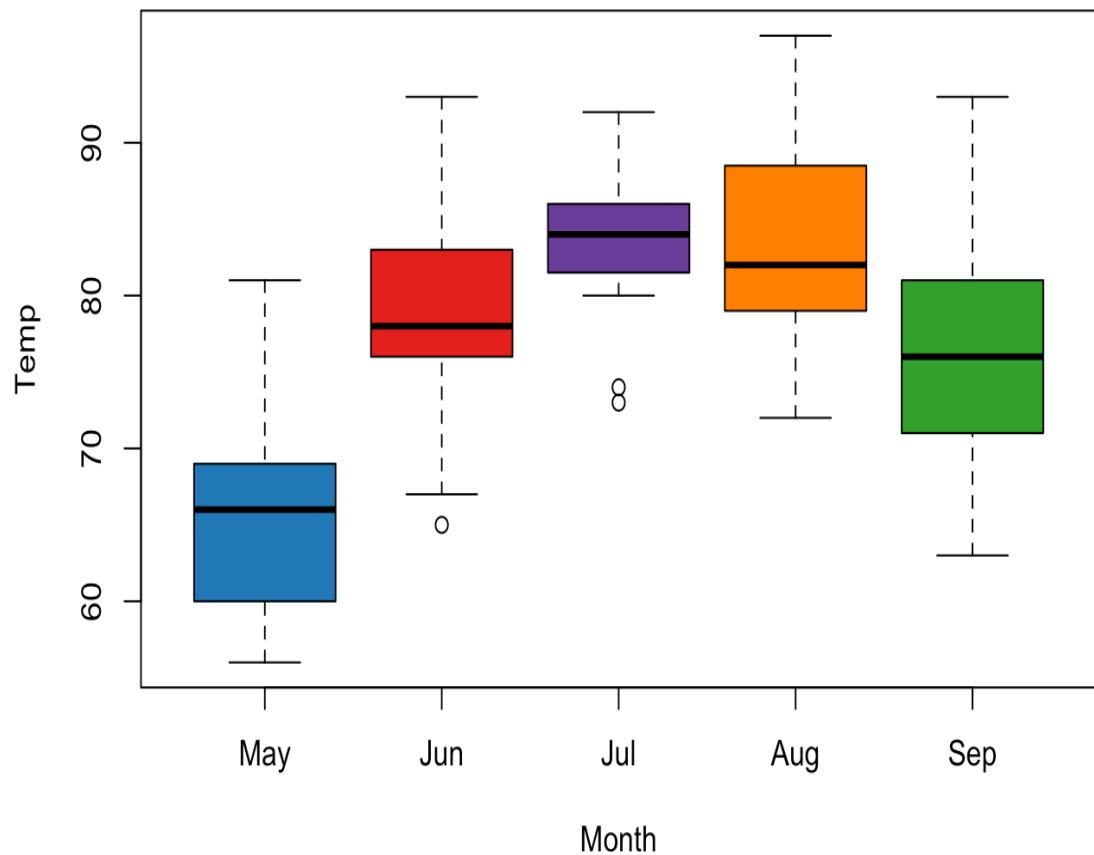
## Distribution of Temperature before and after June



More details can be found [here](#).

Another way to compare distributions across categories is via box plots, see below for an example. The use of a formula `Temp ~ Month` can be used to produce a separate box plot for each month.

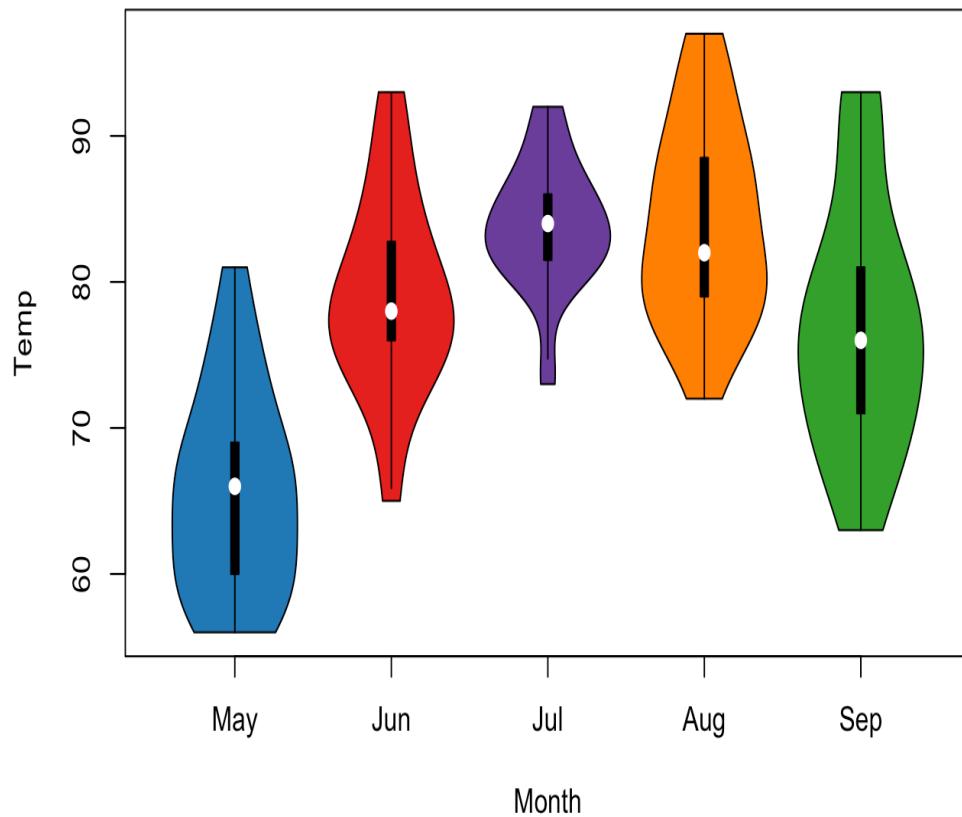
```
boxplot(Temp ~ Month, data=df, col=colours[c(2,6,10,8,4)], names=c('May','Jun','Jul','Aug','Sep'))
```



More information on box plots can be found [here](#).

An alternative to a box plot is a violin plot:

```
library(vioplot)
vioplot(Temp ~ Month, data=df, col=colours[c(2,6,10,8,4)], names =
c('May','Jun','Jul','Aug','Sep'))
```



For more on violin plots you can read [here](#).

# Bar Plots

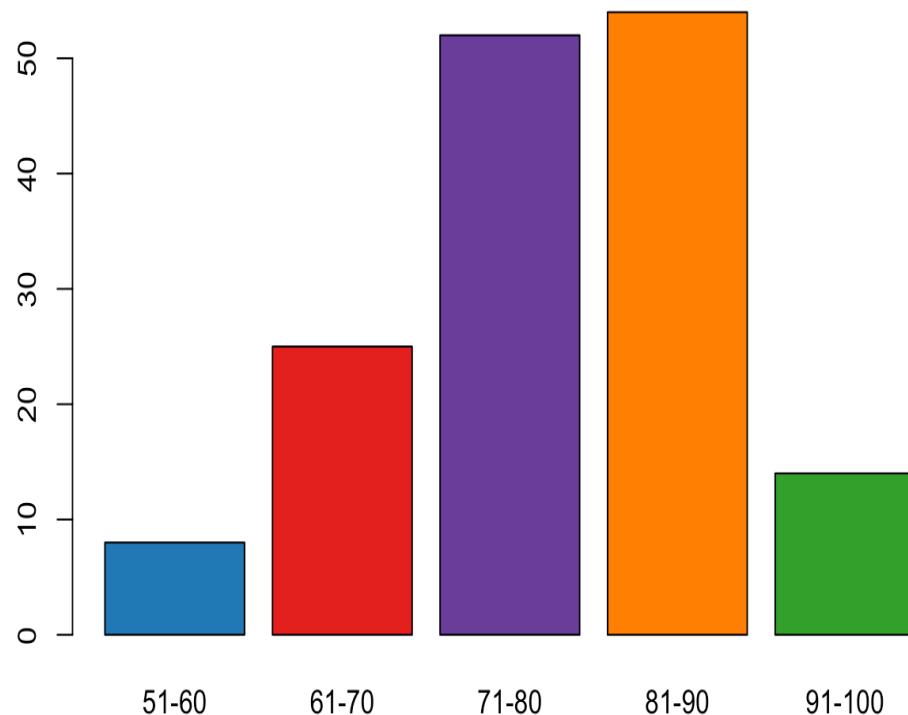
To illustrate bar plots we need a meaningful categorical variable. For this purpose, we will convert the temperature variable into categorical, reflecting a temperature level. More specifically we will create the categories ('51-60', '61-70', '71-80', '81-90' and '91-100') using the `cut()` function:

```
# Uniform colour
categories<-c('51-60','61-70','71-80','81-90','91-100')
df$Temp_cat<-cut(df$Temp, seq(50,100,10), labels=categories)
Temp_counts<-table(df$Temp_cat)
Temp_counts
```

51-60	61-70	71-80	81-90	91-100
8	25	52	54	14

A bar plot for the variable `Temp_cat` can be produced with the code below:

```
barplot(height=Temp_counts, names=categories, col=colours[c(2,6,10,8,4)])
```



Note that it is generally not recommended to convert continuous variables into categorical as information is lost in that case. Here we did it to illustrate the R code.

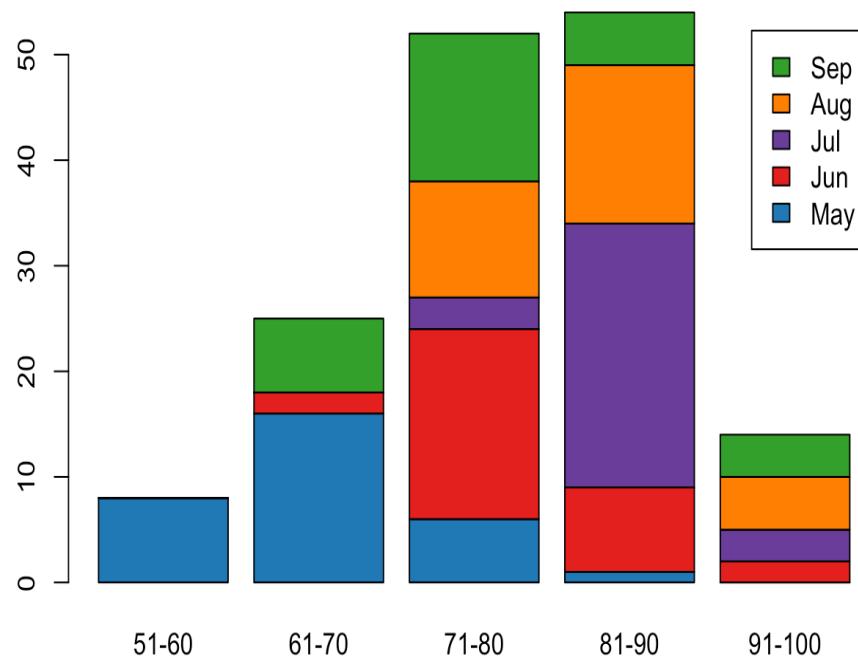
In case of more than two categorical variables, for example Temperature level across months, grouped or stacked bar plots can be used. This essentially uses a cross tabulation between temperature level and month shown below:

```
Temp_counts<-table(df$Month, df$Temp_cat)
Temp_counts
```

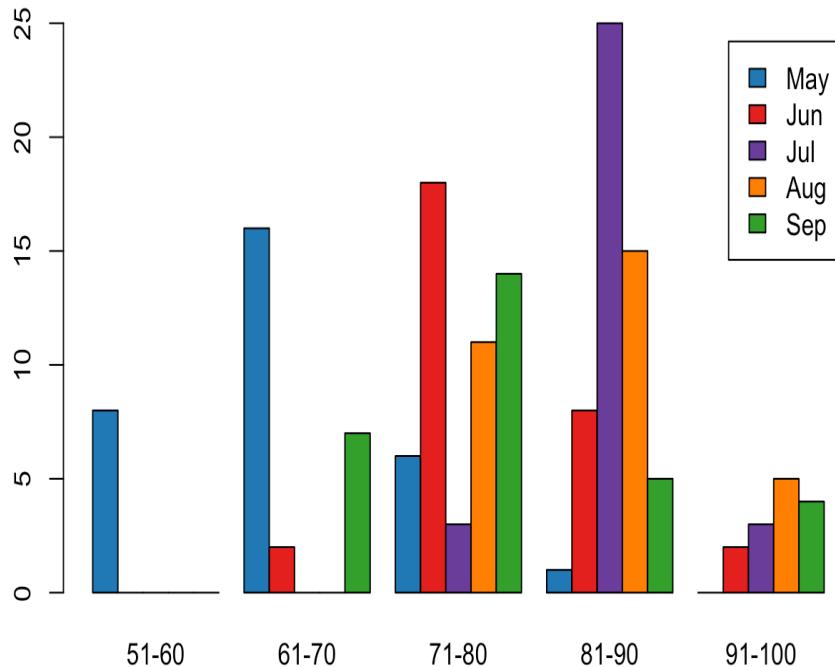
	51-60	61-70	71-80	81-90	91-100
5	8	16	6	1	0
6	0	2	18	8	2
7	0	0	3	25	3
8	0	0	11	15	5
9	0	7	14	5	4

The stacked and grouped (or else side by side) bar plots can be created with the following code:

```
barplot(height=Temp_counts, names=categories,
col=colours[c(2,6,10,8,4)], legend = c('May', 'Jun', 'Jul', 'Aug', 'Sep'))
```



```
barplot(height=Temp_counts, names=categories,
col=colours[c(2,6,10,8,4)], legend =
c('May', 'Jun', 'Jul', 'Aug', 'Sep'), beside=TRUE)
```



More information on bar plots can be found [here](#)

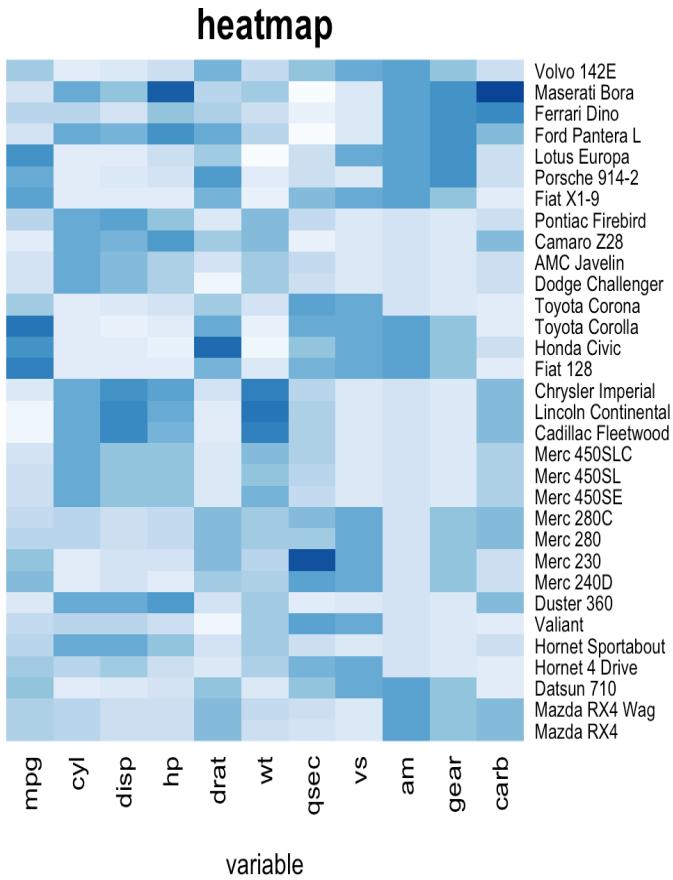
## Heat Maps

Finally, we illustrate the use of a heat map. For this we will use another dataset, called `mtcars`; see [here](#) for some information on the data.

```
head(mtcars)
      mpg cyl disp hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4 108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0    3    2
Valiant       18.1   6 225 105 2.76 3.460 20.22  1  0    3    1
```

For the heat map we will use the `Blues` palette via the code below. We will expand it to contain 25 colours (different versions of blues), rather than the default which is 8, with the function `colorRampPalette`. We will also need to convert the data in `mtcars` to a matrix format using `as.matrix`.

```
data <- as.matrix(mtcars)
colblues <- colorRampPalette(brewer.pal(8, 'Blues'))(25)
heatmap(data, Colv = NA, Rowv = NA, scale='column', xlab='variable',
main='heatmap', col=colblues)
```



## Exporting Graphics from R

To export graphics from R to a specific format, e.g. .jpg, .png, ,pdf, you can follow the next three steps.

First, you need to decide on the type of file and use one of the `jpg()`, `png()` or `pdf()` devices. You may also want to provide the path to the file you want to create. The default is to save the graphics in the working directory under a default name. You can also provide some additional, device-specific parameters such as `width` and `height` if you want to control the output more. Below is an example:

```
png(file = '/Users/Desktop/Plot.pdf',    # directory to save the file
    width = 4, # width of the plot (in inches) optional
    height = 4) # height of the plot (in inches) optional
```

In the second step, one needs to run the code to produce the plot to be exported, for example:

```
plot(x = 1:10,
      y = 1:10)
abline(v = 0) # Additional low-level plotting commands
text(x = 0, y = 1, labels = 'Random text')
```

Finally, an important step is to run the following line of code to create (of “close”) the file; this is an essential step as without it a partial plot may be exported or no plot at all.

```
dev.off()
```

# Useful Links and Resources

- [\*Fundamentals of data visualization\*](#): An easy to read book with many examples of good practice for producing graphs.
- [\*The R graph gallery\*](#): A wide collection of charts made with the R programming.
- [\*HTML accessibility\*](#) A blog on accessibility of HTML pages from R Markdown.

# The Grammar of Graphics and ggplot2

## ggplot2

From [ggplot2's CRAN page](#)

**ggplot2** is a system for “declaratively” creating graphics, based on “The Grammar of Graphics.” You provide the data, tell **ggplot2** how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

The Grammar of Graphics was introduced in Wilkinson (1999) and Wilkinson & Wills (2005) and is a framework for creating graphics following a layered, structured approach.

**ggplot2** is a package that evolved from the ideas in The Grammar of Graphics. The full functionalities of **ggplot2** are too extensive to cover in a week, however there are many tutorials and examples in the links provided to learn from, including the online book [ggplot2: Elegant Graphics for Data Analysis](#).

In this section, we will cover some basics to get started on creating some plots with **ggplot2**.

As we have seen, base R graphics work with functions and methods that are dedicated to particular graphics (e.g. `hist()`, `boxplot`, `plot`, etc). These functions have differences in their syntax and sometimes differences on what they expect as inputs. **ggplot2**, instead, introduces a consistent grammar (a set of rules) to specify the basic components of a graphic (e.g. data, coordinate systems, scales, aesthetics, etc.), which once put together can result in a wealth of graphics.

The basic components in **ggplot2** are

- Data: The data that we want to produce graphics from (typically a `data.frame` or `tibble` object).
- Geometries: The kind of graphic we want to produce, e.g. histogram, scatter plot, bar plot, box plot, etc (constructed by `geom_*` family of functions).
- Aesthetics: Aesthetics of the geometric and statistical objects, such as position, colour, size, shape, and transparency (specified using the `aes()` function).
- Facets: The arrangement of graphics into grids (specified using the `facet_*` family of functions).
- Visual themes: The overall visual defaults of a plot, such as background, grids, axes, default typeface, sizes, and colours (specified using the `theme_*` family of functions).

There are other layers that you can add that can greatly improve your plots.

For more extensive tutorials and descriptions on **ggplot2**, see the recommended resources in Useful Links and Resources. In particular, [RStudio’s “Data visualization with ggplot2” cheat sheet](#), which is a good resource to keep handy while working with **ggplot2**, and the online book [ggplot2: Elegant Graphics for Data Analysis](#) is the definitive resource on the philosophy and operation of the package. Chapter 22 of Wickham & Grolemund (2017) also provides an excellent walkthrough **ggplot2**’s capabilities.

### **ggplot()**

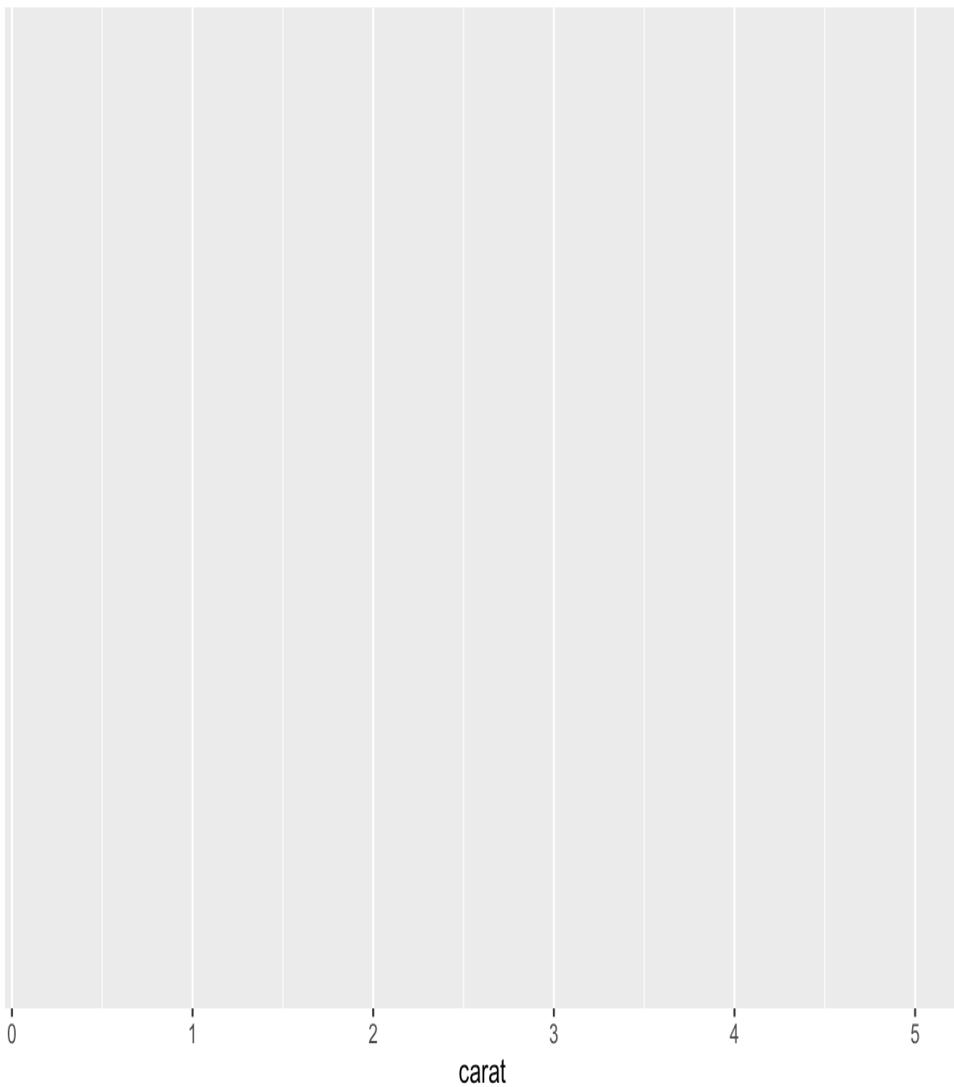
All **ggplot2** graphics begin with a call to `ggplot()`, where we can supply the data. After that call, we can keep enhancing the resulting object by adding more layers. Optionally with the `ggplot()` function, we can add whatever aesthetics we want to apply to the graphics, for example, by specifying what variables from the data will form the axes.

As an example, we work with the data in the `diamonds` dataset in R, which provides prices and other attributes of over 50000 diamonds. To initialize graphics, we start with the data layer through the `ggplot()` function:

```
library(ggplot2)
# initialize ggplot
ggplot(diamonds)
```

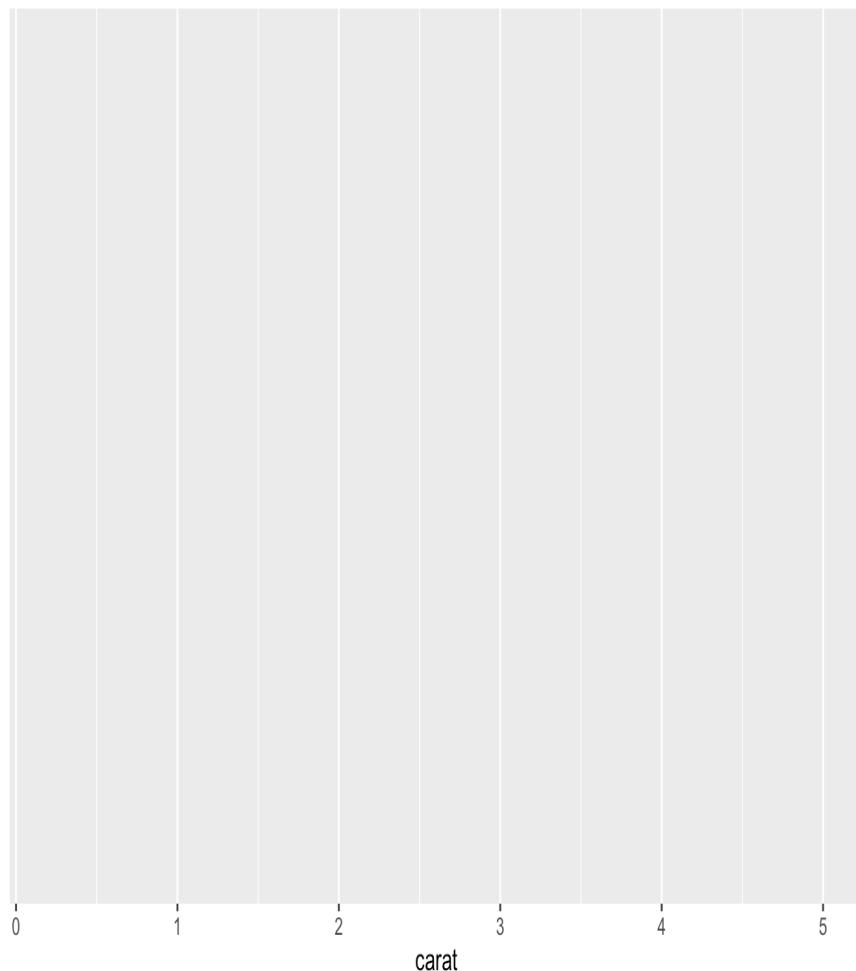
This does not produce a plot yet. To specify what features of the data we want to look at, we need to add some aesthetics. We specify aesthetics using the `aes()` function, and add that specification a layer by simply using the `+` operator. For example,

```
# initialise ggplot data and aesthetics layer
ggplot(diamonds) + aes(x = carat)
```



or, we can do the same by passing the aesthetics directly in the `ggplot()` function:

```
# initialise ggplot data and aesthetics layer  
ggplot(diamonds, aes(x = carat))
```



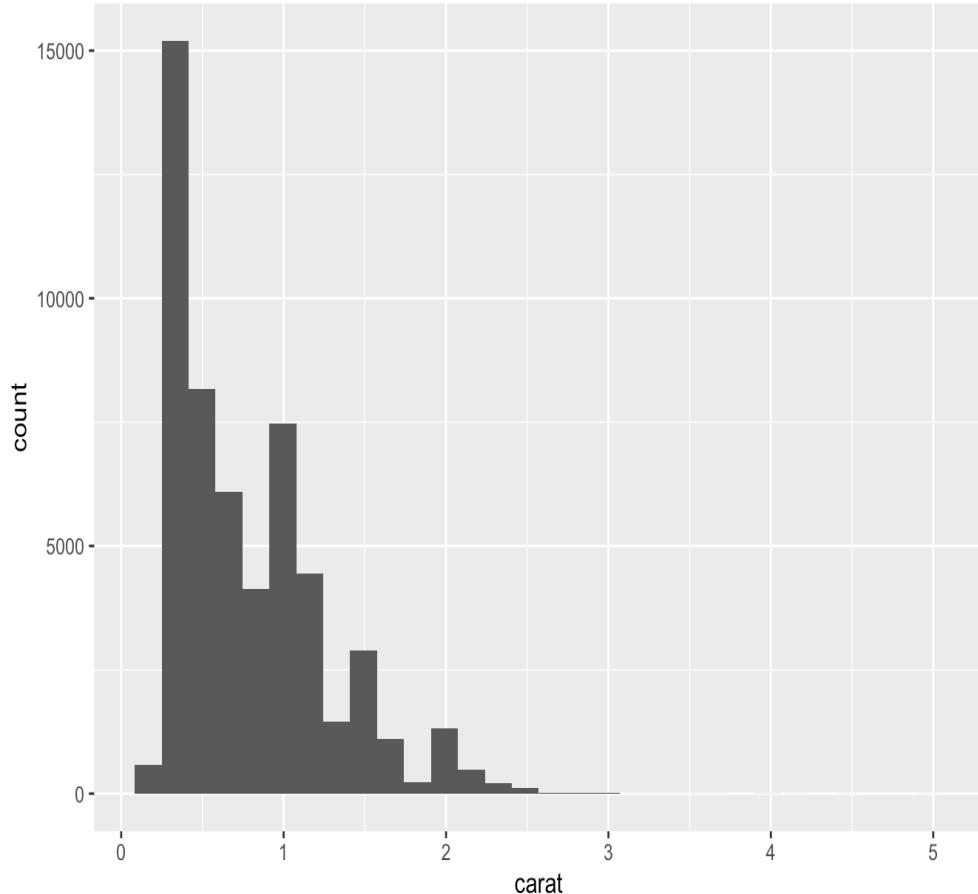
Again, this is not producing anything interesting yet. We can define the graphics we want to produce by adding a `geom_*` layer (geometric object). Some examples of geometric objects in ggplot2 are:

- `geom_point()`: scatter plot
- `geom_histogram()`: histogram
- `geom_bar()`: bar chart
- `geom_boxplot()`: box plot
- `geom_density()`: smooth density estimates

# Histograms

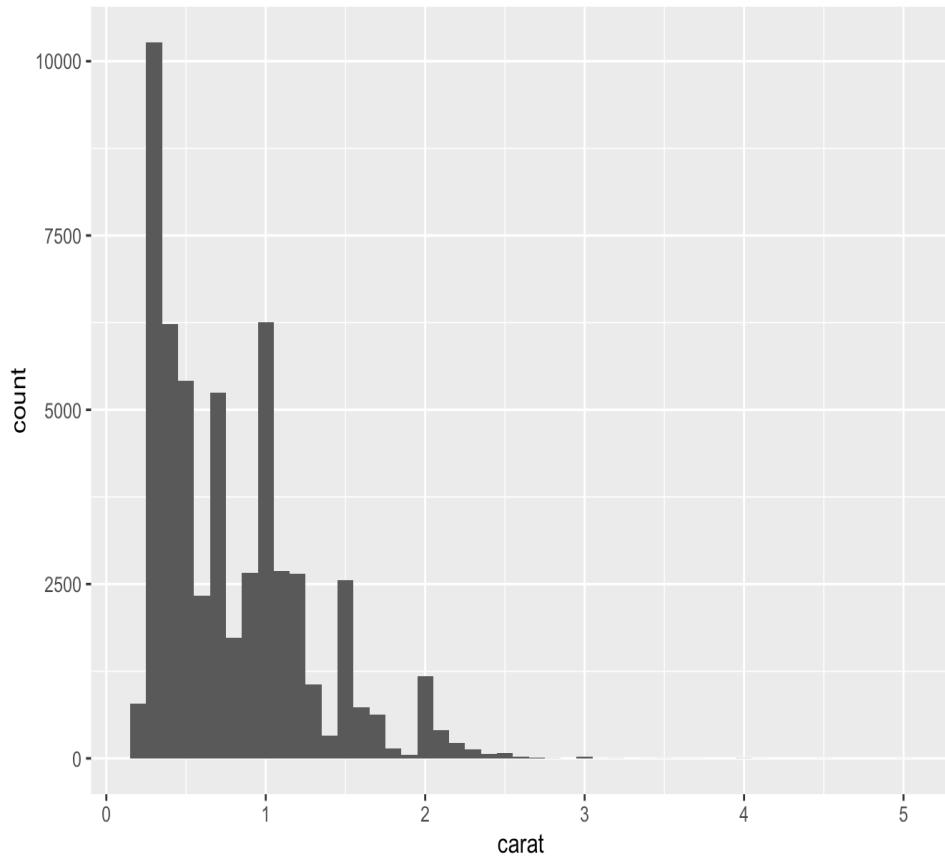
To plot a histogram of the diamond carats, we add the `geom_histogram()` function to the latest call.

```
# initialise ggplot data and aesthetics layer and
# create a histogram plot with geom_histogram()
ggplot(diamonds, aes(x = carat)) + geom_histogram()
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The help pages of `geom_histogram()` describe a range of optional arguments that can be used to customize the histogram further. For example,

```
# create histogram and add title and axis labels
my_hist <- ggplot(diamonds, aes(x = carat)) +
  geom_histogram(binwidth = 0.1)
my_hist
```

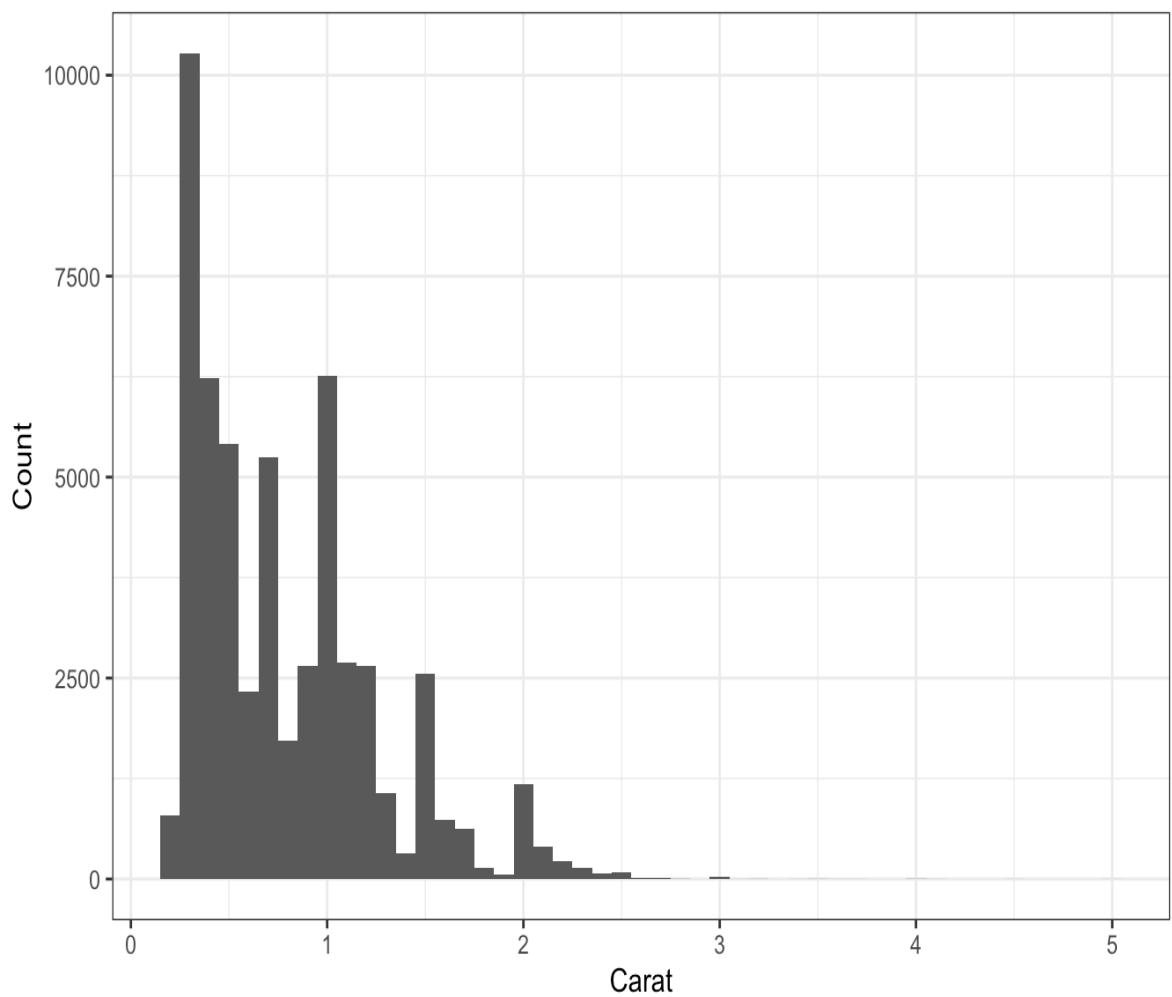


Note here that we assigned the name `my_hist` to the object returned after combining the **ggplot2** layers, and printed that. This is one the powers of **ggplot2**; it allows us to specify the graphics, and retrieve them for further enrichment or edits later.

For example, we can enrich `my_hist` by using the `labs()` layer to specify titles for the plot and labels for the axes, and the `theme_bw()` layer to use a theme with black and white elements (see `?labs` and `?theme_bw` for details).

```
my_hist +  
  labs(title = "Weight of Diamonds", x = "Carat", y = "Count") +  
  theme_bw()
```

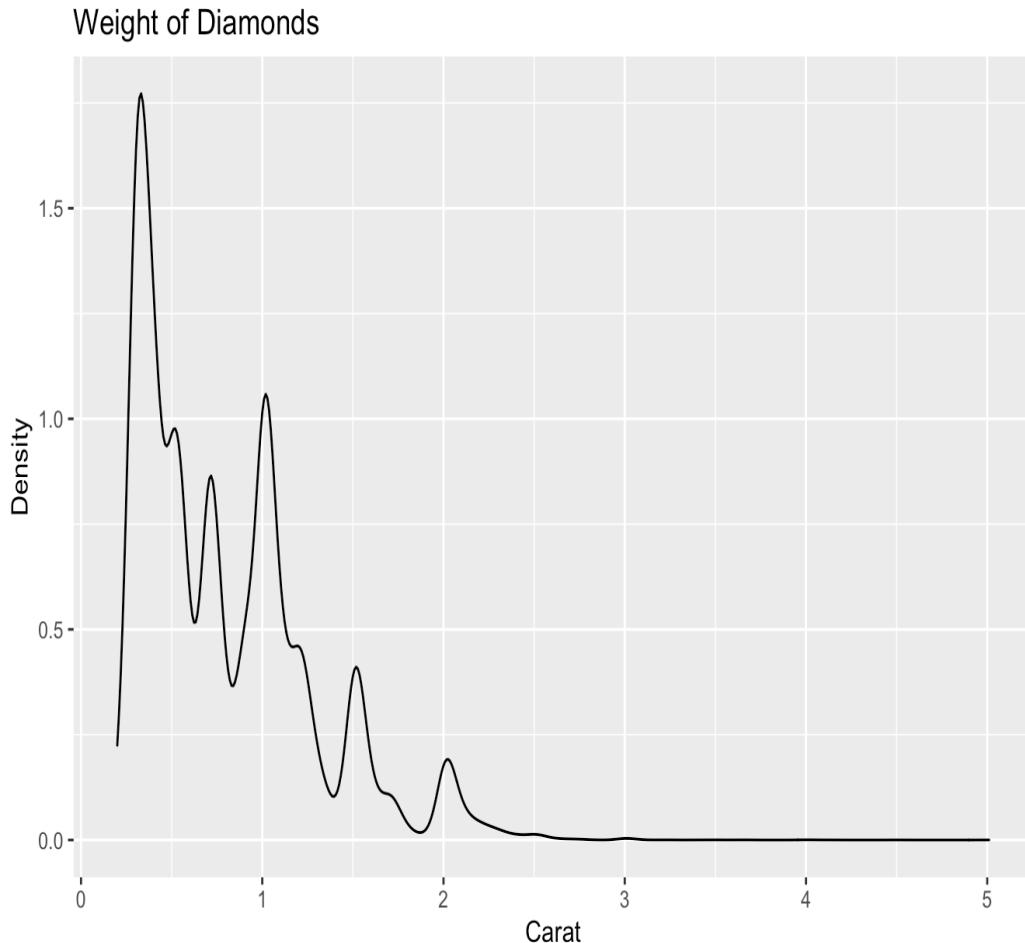
## Weight of Diamonds



# Density Plots

Here is another example where we create a graphic with a kernel density estimate with the `geom_density()` function:

```
# create kernel density plot and add title and axis labels
my_kd <- ggplot(diamonds, aes(x = carat)) +
  geom_density() +
  labs(title = "Weight of Diamonds", x = "Carat", y = "Density")
my_kd
```



# Plots

A scatter plot is a two-dimensional plot, so **ggplot2** expects to pass two arguments into the aesthetics layer.

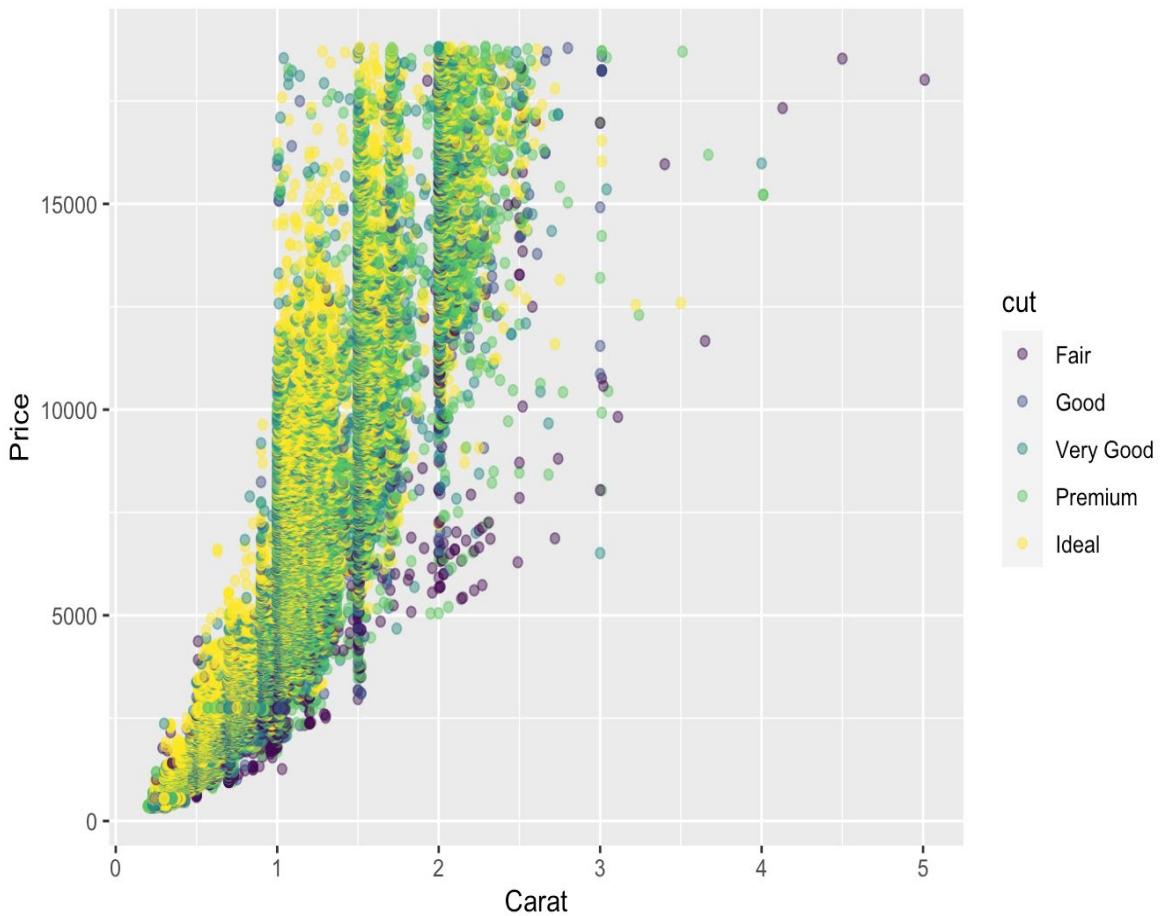
A quick view on the diamonds data set

```
str(diamonds)
## # tibble [53,940 x 10] (S3: tbl_df/tbl/data.frame)
## $ carat   : num [1:53940] 0.23 0.21 0.23 0.29 0.31 ...
## $ cut      : Ord.factor w/ 5 levels "Fair"<"Good"<...
## $ color    : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<...
## $ clarity  : Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<...
## $ depth    : num [1:53940] 61.5 59.8 56.9 62.4 63.3 ...
## $ table   : num [1:53940] 55 61 65 58 58 ...
## $ price   : int [1:53940] 326 326 327 334 335 336 336 ...
## $ x        : num [1:53940] 3.95 3.89 4.05 4.2 4.34 ...
## $ y        : num [1:53940] 3.98 3.84 4.07 4.23 4.35 ...
## $ z        : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

or a look at the help file (`?diamonds`) reveals that we have information on the price (in the `price` variable) of each diamond and a rating on the quality of the cut (in the `cut` variable). So, let's make a scatter plot of `price` versus `carat` and colour by `cut`. This can be done easily using `geom_point()` and enriching the aesthetics a bit

```
my_sc <- ggplot(diamonds) +
  geom_point(aes(x = carat, y = price, colour = cut), alpha = 0.5) +
  labs(title = 'Diamonds are forever...', x = 'Carat', y = 'Price')
my_sc
```

Diamonds are forever...

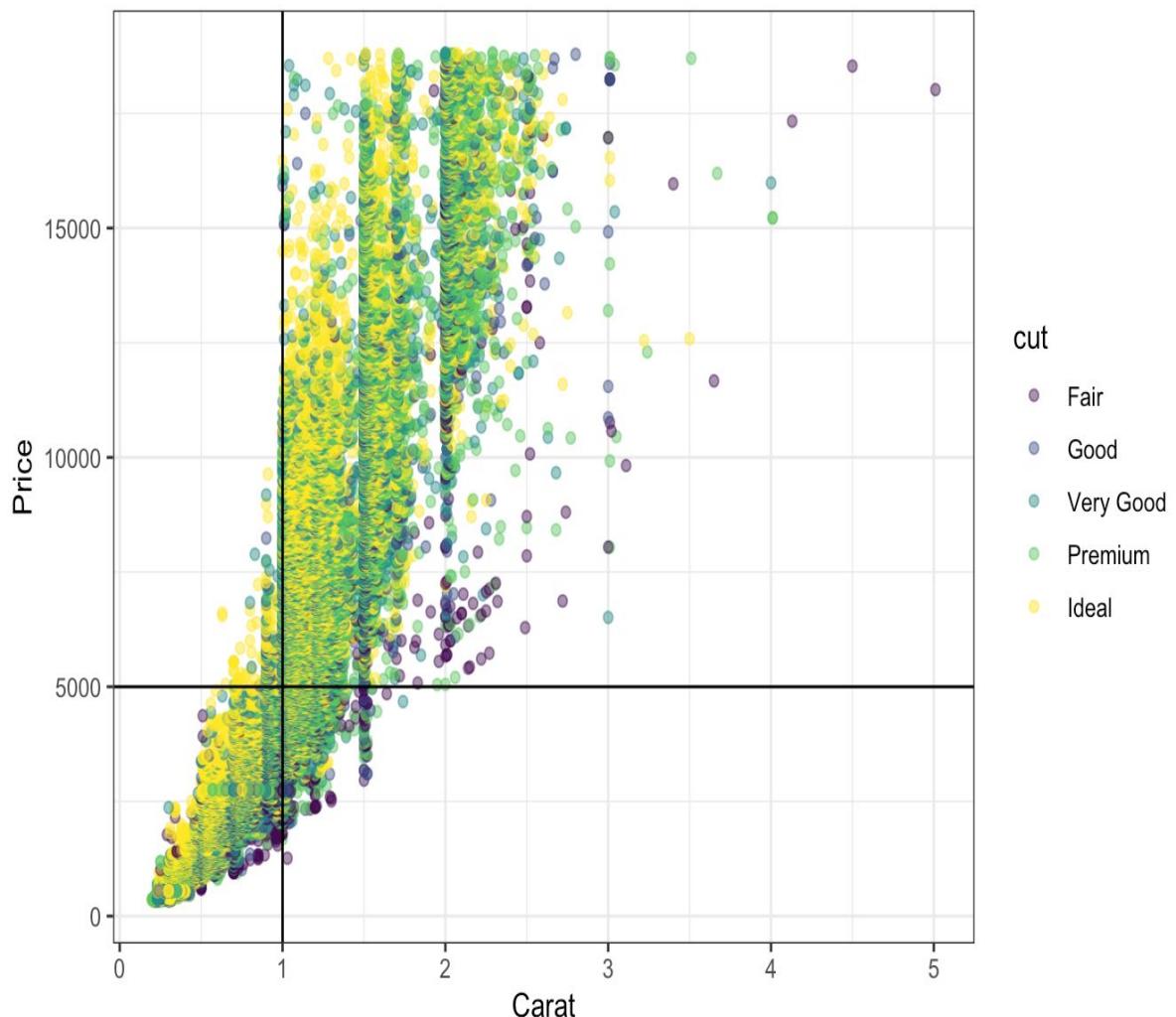


The above code chunk also shows how the aesthetics can be passed directly to the `geom_*`() functions. We passed `geom_point()` the aesthetic `aes(x = carat, y = price, colour = cut)` to ask **ggplot2** to colour the points according to the quality of their cut, and `alpha = 0.5` in order to ask **ggplot2** to add transparency to the points in order to make overplotting a bit less unpleasant to the eye.

In fact, the most convenient place for aesthetics in multi-layered plots with multiple geometries is the `geom_*`() functions. For example, for adding a horizontal line at 5000 and a vertical line at 2 in the above graphic we use `geom_hline()` and `geom_vline()`, each with its own aesthetic:

```
my_sc <- my_sc +
  geom_hline(aes(yintercept = 5000)) +
  geom_vline(aes(xintercept = 1)) +
  theme_bw()
my_sc
```

Diamonds are forever...

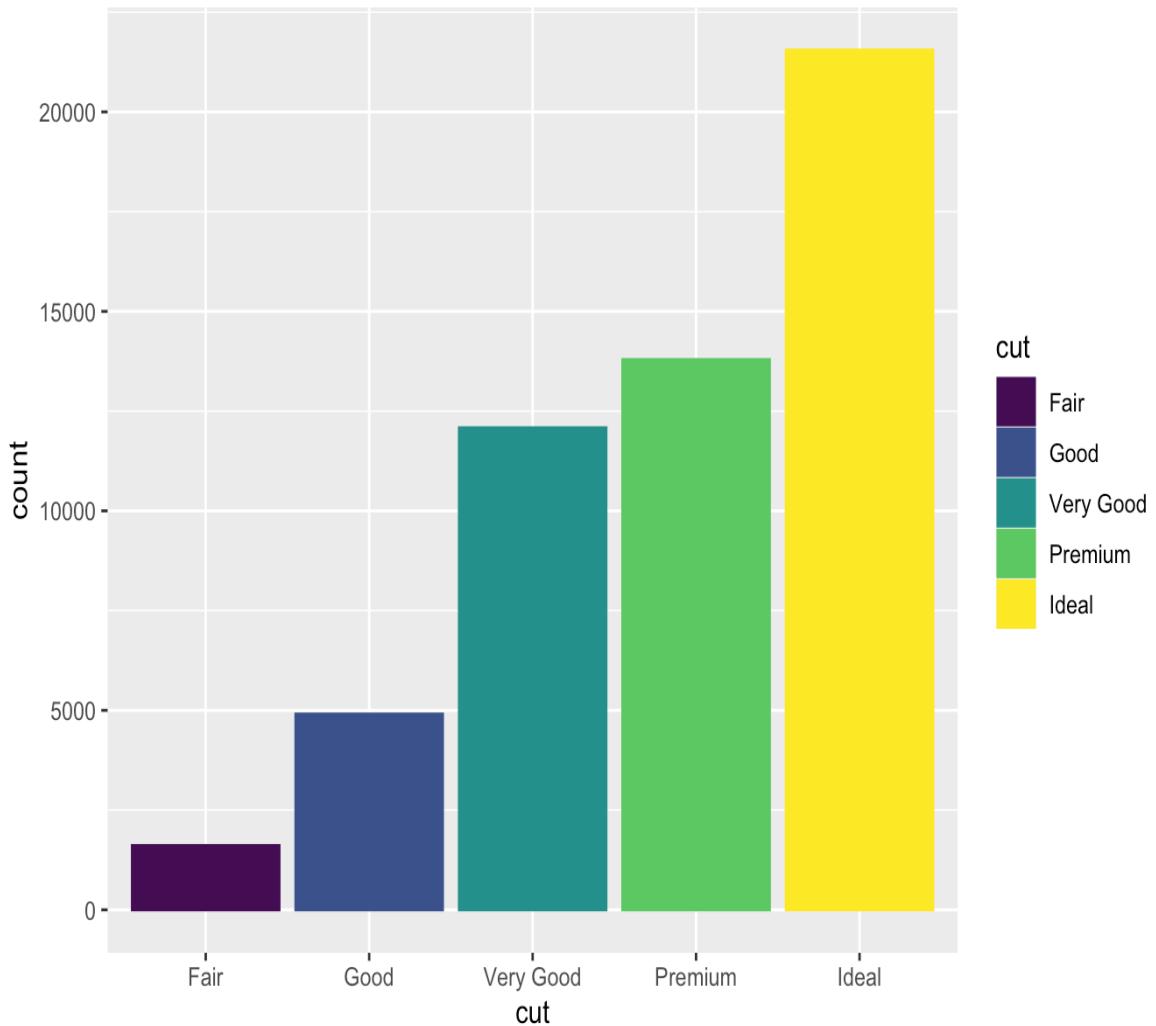


The aesthetics that each geometry understands can be found directly in the help file of the geometry (type `?geom_scatterplot` and look for the expected aesthetics in the help file).

# Bar Plots

For a quick view of the frequencies of the available cuts in `diamonds` we can construct a bar plot

```
ggplot(diamonds) + geom_bar(aes(x = cut, colour = cut, fill = cut))
```

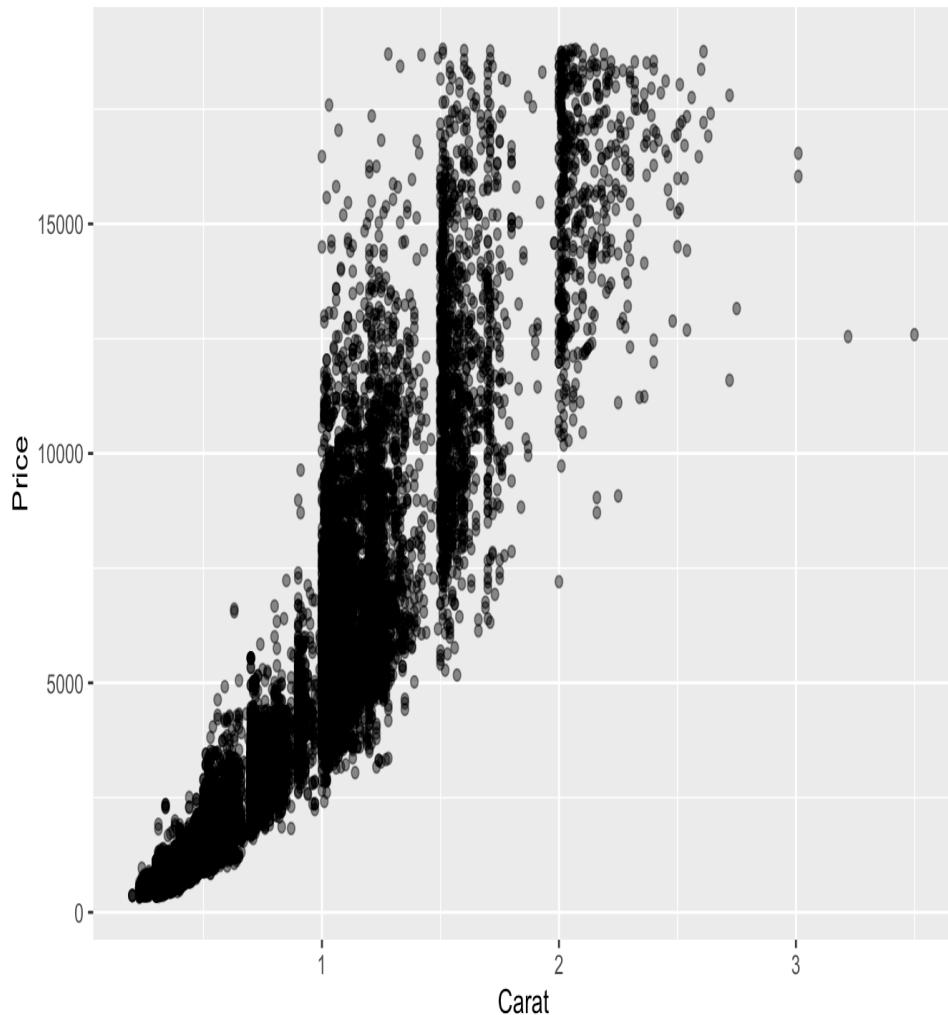


# Faceting

Now suppose that you would want to have a scatter plot of `price` versus `carat` for each value of `cut`. One way to get this is to subset the data you pass into the `ggplot()` function and produce one scatter plot for each value of `cut`. For example, using the `filter` function from `dplyr`:

```
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
ggplot(diamonds %>% filter(cut == "Ideal")) +
  geom_point(aes(x = carat, y = price), alpha = 0.5) +
  labs(title = 'Diamonds are forever...', x = 'Carat', y = 'Price')
```

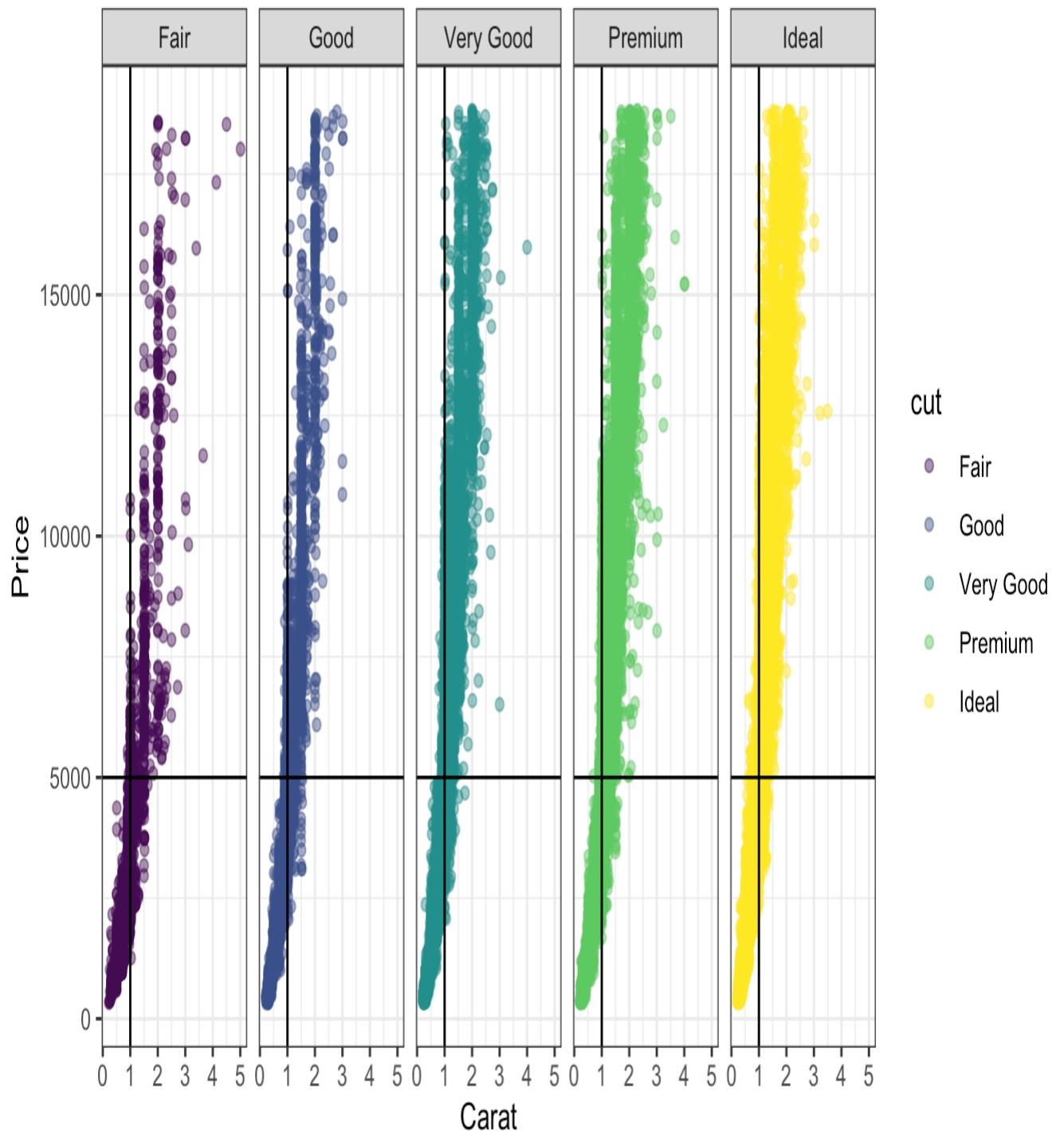
Diamonds are forever...



A better way is to use faceting. Using the `my_sc` object we constructed earlier, this is as simple as

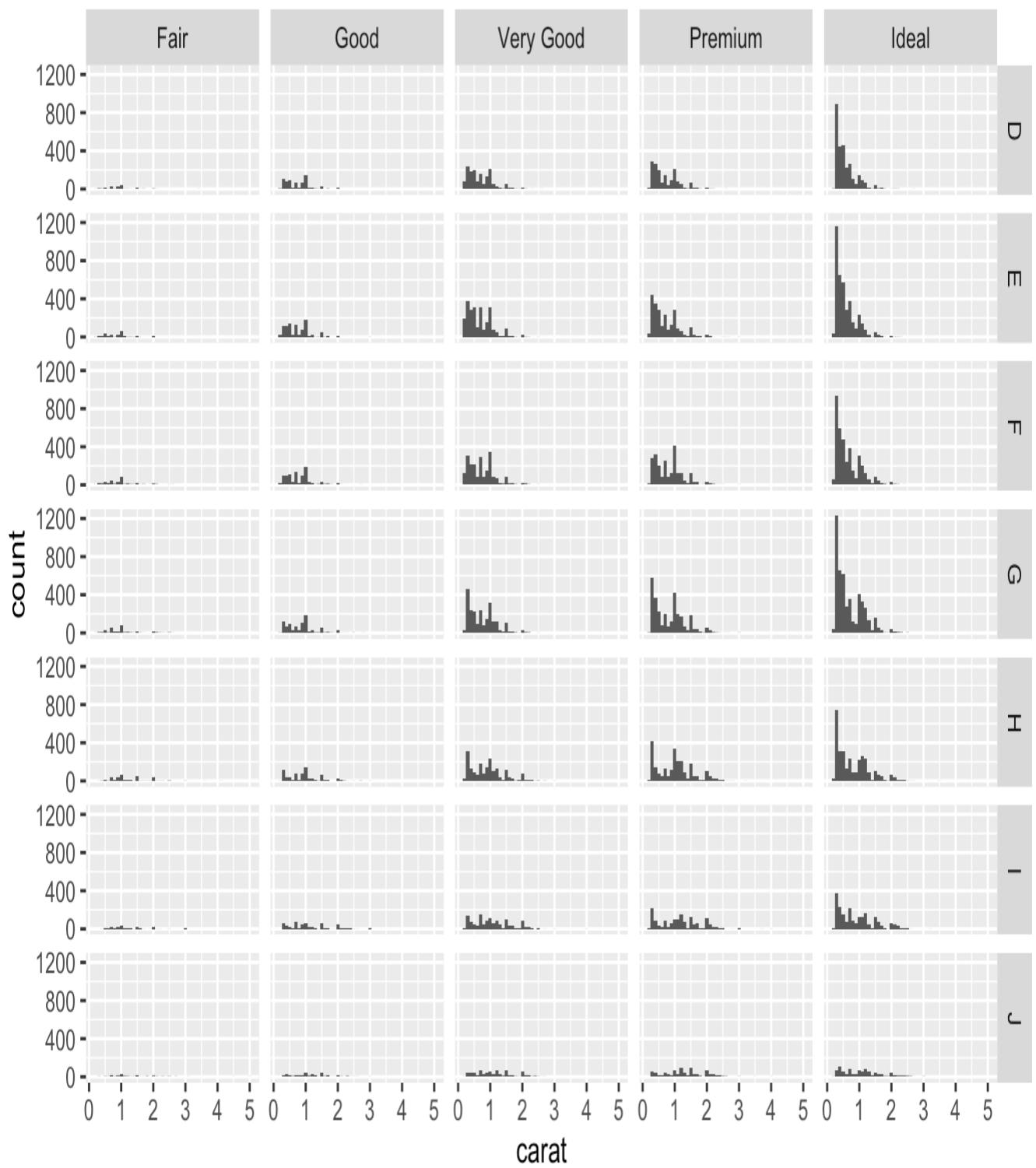
```
my_sc + facet_grid(~ cut)
```

## Diamonds are forever...



We can facet with more than one variables. For example, if we want the carat histogram for each combination of colour and cut we do

```
my_hist + facet_grid(color ~ cut)
```



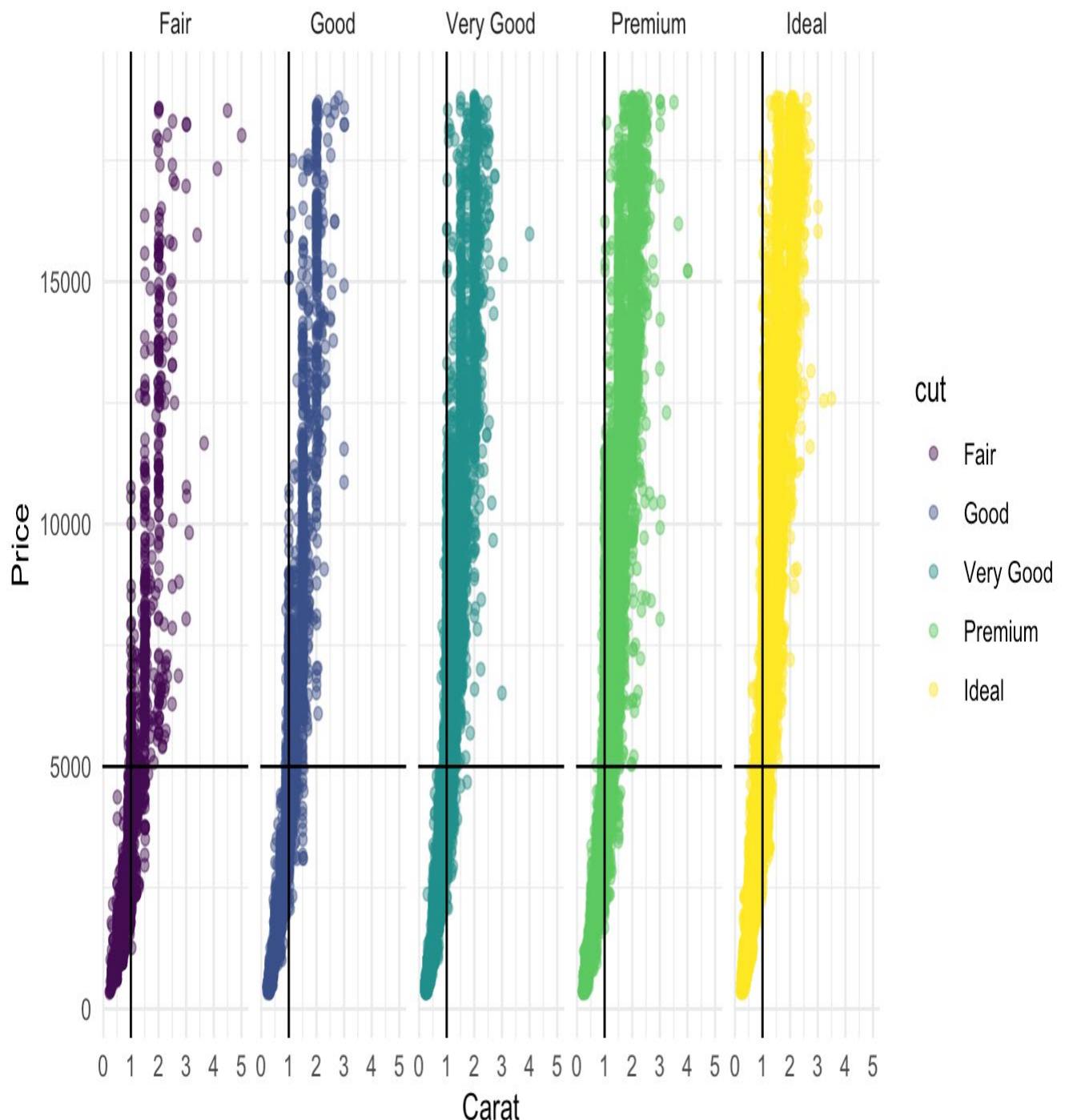
Take a look at `?facet_grid` and `?facet_wrap` for more information about faceting, and the several available arguments for producing a wealth of grids.

# Themes

The `theme_*` family of functions of `ggplot2` allows you to quickly change the look and feel of your graphics. `ggplot2` ships with some default themes. We have already seen `theme_bw()`. Here are a few more

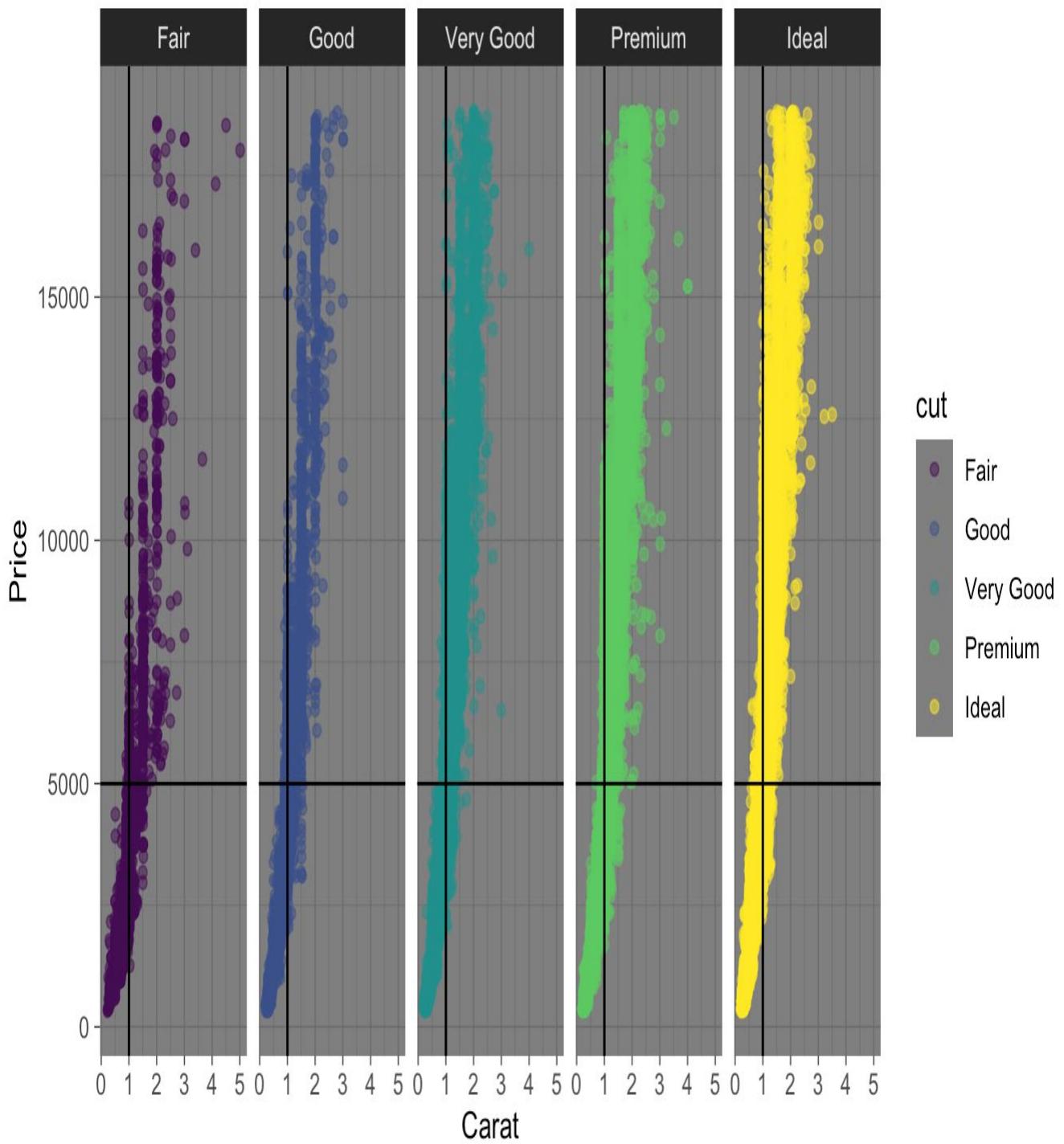
```
my_sc + facet_grid(~ cut) + theme_minimal()
```

Diamonds are forever...



```
my_sc + facet_grid(~ cut) + theme_dark()
```

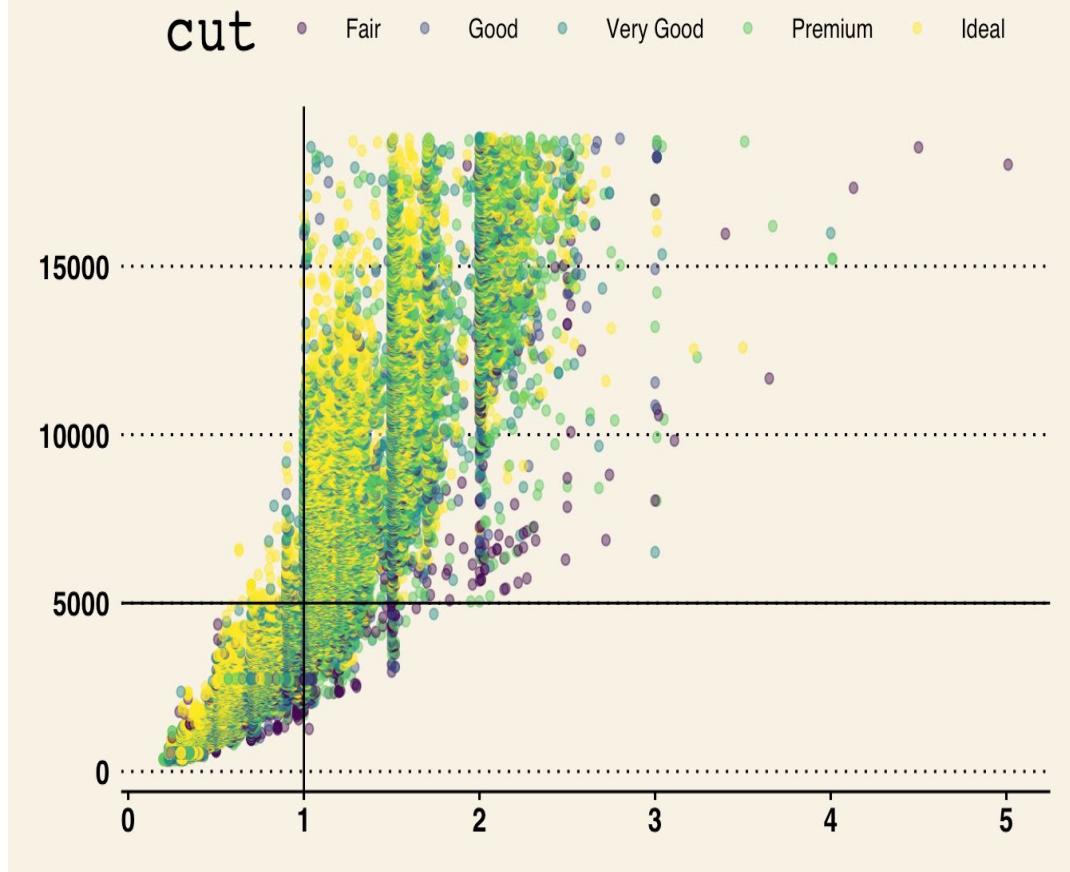
Diamonds are forever...



There is also the [ggthemes](#) R package that provides some extra themes. For example, after you install [ggthemes](#), you can do

```
library(ggthemes)
my_sc + theme_wsj()
```

# Diamonds are forever...



to get graphics closely resembling the graphics theme used in *The Wall Street Journal*.

## Useful Links and Resources

- [The ggplot2 page](#)
- [ggplot2: Elegant graphics for data analysis](#): An online book by Hadley Wickham focusing on graphics and ggplot2
- [Cédric Scherer's blog post "A ggplot2 tutorial for beautiful plotting in R"](#)
- [RStudio's data visualization with ggplot2" cheat sheet](#)
- ["Graphics for communication" chapter of R for Data Science](#)

## References

- Wickham, H., & Grolemund, G. (2017). *R for data science: Import, tidy, transform, visualize, and model data* (1st ed.). O'Reilly Media.
- Wilkinson, L. (1999). *The grammar of graphics* (1st ed.). Springer.
- Wilkinson, L., & Wills, G. (2005). *The grammar of graphics* (2nd ed.). Springer.

# ST2195 Programming for Data Science

## Graphics and Data Visualisation in Python



\*\* Note: The code chunks below should be run in the following order \*\*

## Plotting in Python

In Python, **matplotlib** and **seaborn** are two main Python libraries for plots. **matplotlib** is a basic plotting library. **seaborn** is a Python data visualization library based on matplotlib, and it provides some “prettier” and sometimes more interesting plots. We use both **matplotlib** and **seaborn** in this notebook, although we mainly use **matplotlib** here. We import the libraries as follows:

```
import matplotlib.pyplot as plt  
import seaborn as sns  
  
# Set the colour palette to colour-blind friendly  
sns.reset_orig()  
my_palette = sns.color_palette("colorblind")  
plt.style.use('seaborn-colorblind')
```

We will first start with the basics of **matplotlib**, which includes how to create a space for the plot, and how to make a basic plot. We will then talk about how to create different plots including:

- Line plot
- Bar plot
- Boxplot
- Scatter plot
- Heat map

using **matplotlib** and **seaborn**. Before we introduce how to use the plotting libraries, let us first load the data for visualisation.

# Import and Clean the Data

We use GDP per capita data from `DBnomics` and wine chemistry composition data from `scikit-learn` in this notebook. The data are already downloaded and stored in three csv files: `gdp.csv`, `gdp_wide` and `wine.csv` and we load the data to Python with the following code:

```
import numpy as np
import pandas as pd
import json
from datetime import datetime

gdp = pd.read_csv("gdp.csv")
# convert the year column from string to datetime object
gdp['year'] = gdp['year'].apply(lambda x : datetime.strptime(x, '%Y-%M-%d'))

# get the GDP per capita data for each country
# reset_index so that each Series has indexes 0,1,2,3...
gdp_uk = gdp[gdp['country']=='UK'].reset_index()
gdp_fr = gdp[gdp['country']=='FR'].reset_index()
gdp_it = gdp[gdp['country']=='IT'].reset_index()
gdp_de = gdp[gdp['country']=='DE'].reset_index()

gdp_wide = pd.read_csv("gdp_wide.csv")
# set the row labels as countries
gdp_wide.set_index('country', inplace= True)
# set the column labels (years) with the type int
gdp_wide.columns = gdp_wide.columns.astype(int)

wine = pd.read_csv("wine.csv")
```

`gdp_uk`, `gdp_fr`, `gdp_it`, `gdp_de` store the GDP per capita time series for a corresponding country. For example

```
print(gdp_uk)

   index          year    value country
0      0 2000-01-01 00:01:00  27130.0     UK
1      1 2001-01-01 00:01:00  27770.0     UK
2      2 2002-01-01 00:01:00  28250.0     UK
3      3 2003-01-01 00:01:00  29060.0     UK
4      4 2004-01-01 00:01:00  29560.0     UK
5      5 2005-01-01 00:01:00  30210.0     UK
6      6 2006-01-01 00:01:00  30810.0     UK
```

7	7	2007-01-01	00:01:00	31280.0	UK
8	8	2008-01-01	00:01:00	30940.0	UK
9	9	2009-01-01	00:01:00	29460.0	UK
10	10	2010-01-01	00:01:00	29830.0	UK
11	11	2011-01-01	00:01:00	29960.0	UK
12	12	2012-01-01	00:01:00	30190.0	UK
13	13	2013-01-01	00:01:00	30660.0	UK
14	14	2014-01-01	00:01:00	31290.0	UK
15	15	2015-01-01	00:01:00	31780.0	UK
16	16	2016-01-01	00:01:00	32060.0	UK
17	17	2017-01-01	00:01:00	32430.0	UK
18	18	2018-01-01	00:01:00	32640.0	UK
19	19	2019-01-01	00:01:00	32870.0	UK

`gdp_wide` stores the four time series above in a table:

	2000	2001	2002	2003	...	2016	2017	2018	2019
country					...				
DE	28910.0	29370.0	29290.0	29100.0	...	34610.0	35380.0	35720.0	35840.0
FR	28930.0	29290.0	29410.0	29440.0	...	31770.0	32380.0	32860.0	33270.0
IT	27430.0	27950.0	27960.0	27850.0	...	26020.0	26490.0	26780.0	26920.0
UK	27130.0	27770.0	28250.0	29060.0	...	32060.0	32430.0	32640.0	32870.0

[4 rows x 20 columns]

`wine` stores the chemistry composition of wines in a table, with the last column provides the information of which class the wines belong to:

	alcohol	malic_acid	ash	...	od280/od315_of_diluted_wines	proline	target
0	14.23	1.71	2.43	...		3.92	class_0
1	13.20	1.78	2.14	...		3.40	class_0
2	13.16	2.36	2.67	...		3.17	class_0
3	14.37	1.95	2.50	...		3.45	class_0
4	13.24	2.59	2.87	...		2.93	class_0

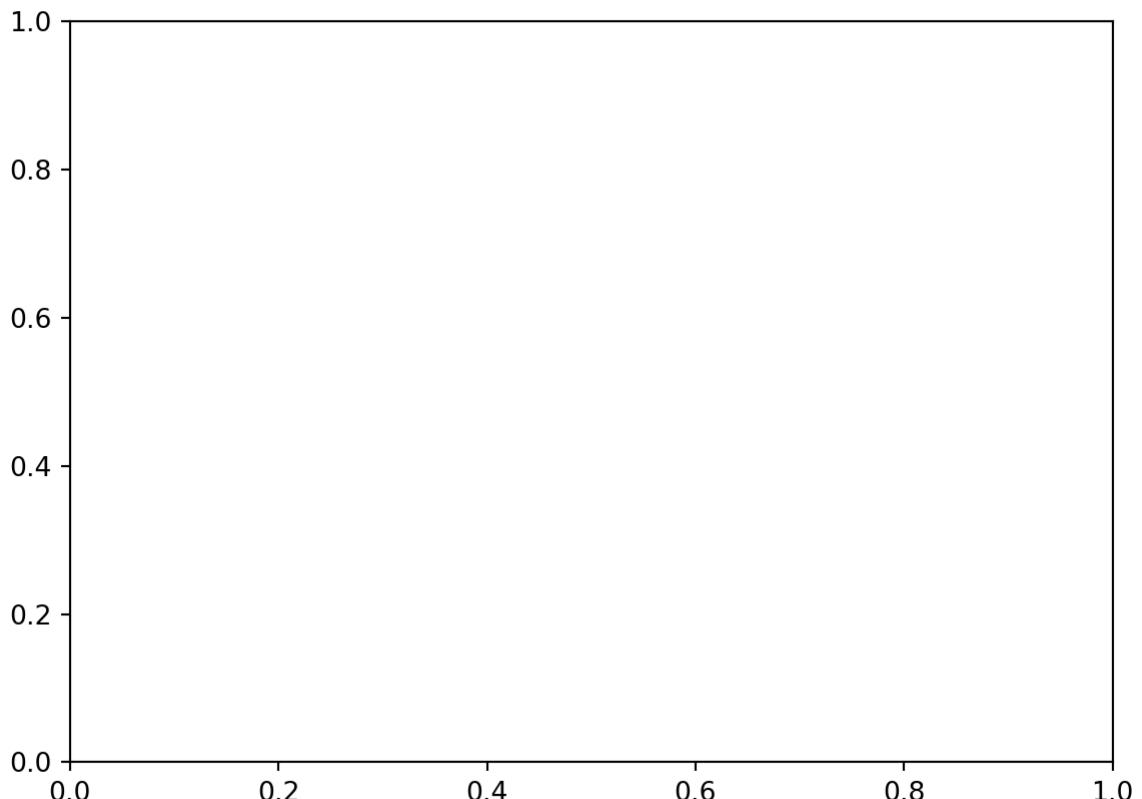
[5 rows x 14 columns]

# matplotlib Basics

## Creating an Empty Plot

We first learn how to create a empty figure. It can be done by:

```
fig, ax = plt.subplots()  
plt.show()
```



this return a `Figure` and an `Axes` object. By manipulating these objects, we can plot and set the properties of the plot. `plt.show()` is used to make sure the plot is shown (although the plot may still be shown without calling `plt.show()`).

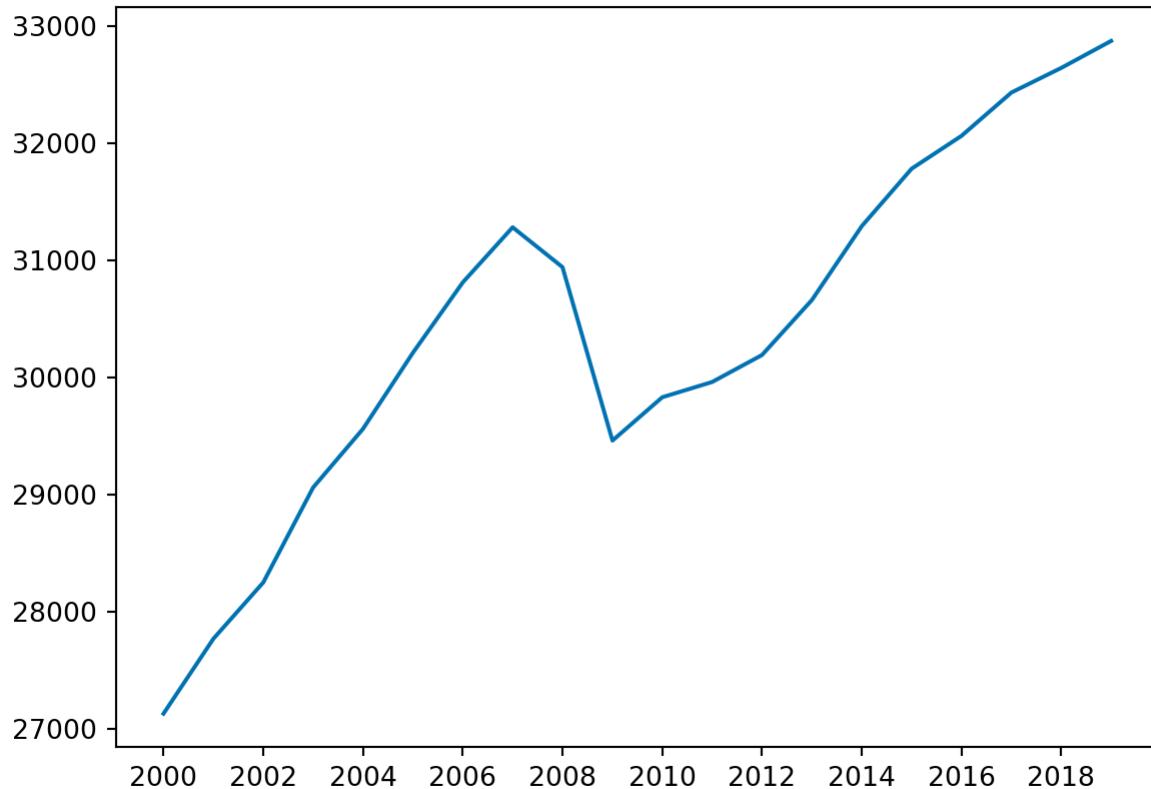
In this notebook we focus on manipulating the plots using `Axes` object `ax` or `plt` but not the `Figure` object `fig`. If you want to learn how you can use `Figure` object to manipulate the plots, please see [here](#).

## Plotting Using `ax`

After the empty plot is created, we can plot the data by using the `Axes` object `ax` and the method `plot()`. Here plot the GDP per capital time series data:

```
fig, ax = plt.subplots()
```

```
ax.plot(gdp_uk['year'], gdp_uk['value'])
plt.show()
```



## Plotting Multiple Subplots

Similar to R, we can have multiple subplots in one figure. This can be done by specifying the number of rows/columns of the subplot grid in the `subplots()` function. It returns a figure and a *collection* of axes objects. We can plot the data on each of the subplots via the axes, and here we use different GDP per capita time series to illustrate it:

```
fig, ax = plt.subplots(2, 2, figsize = (15,10)) # ax is an array of an array (2x2)

ax[0][0].plot(gdp_uk['year'], gdp_uk['value']) # top left
ax[0][0].title.set_text('U.K')

ax[0][1].plot(gdp_fr['year'], gdp_fr['value']) # top right
ax[0][1].title.set_text('France')

ax[1][0].plot(gdp_de['year'], gdp_de['value'], label = 'Germany') # bottom left
```

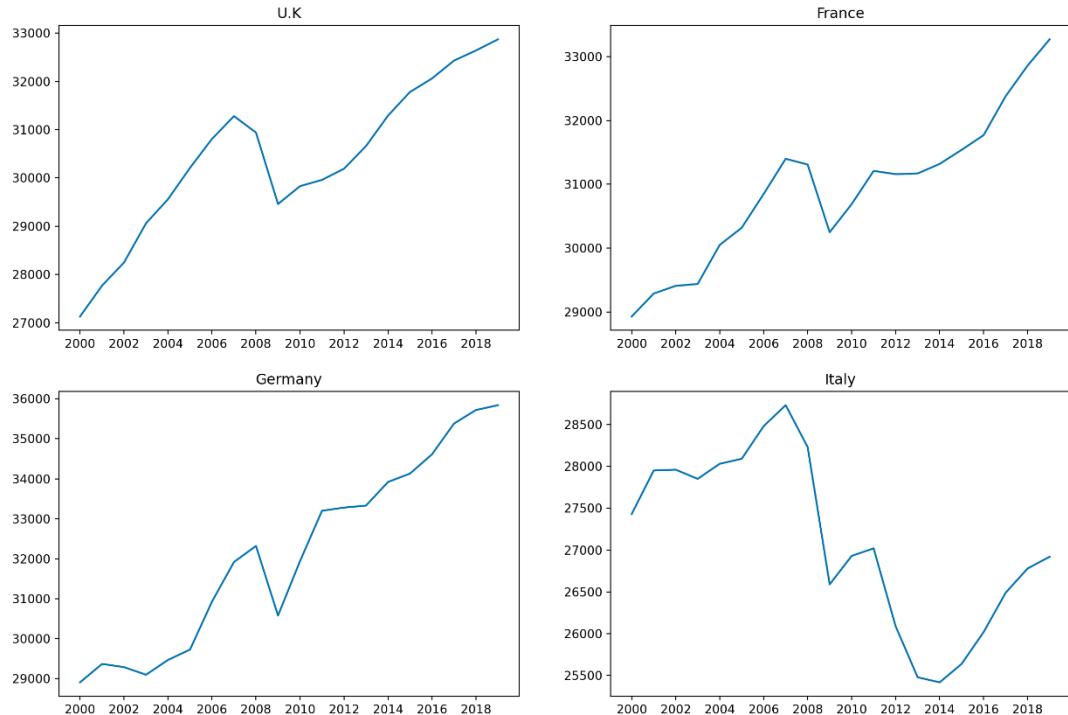
```

ax[1][0].title.set_text('Germany')

ax[1][1].plot(gdp_it['year'], gdp_it['value'], label = 'Italy') # bottom right
ax[1][1].title.set_text('Italy')

plt.show()

```



The argument `figsize` in `subplots()` determines the size of the figure.

## Plotting Multiple Lines on the *Same* Plot

Similar to R, we can plot multiple lines on the same subplot. This can be done by calling `plot()` via the same axes (here `ax[0][0]`) with different data (as illustrated below).

`ax[0][0].legend()` creates the legend for the lines for the top left subplot. We can add label, title, etc. to the subplot by manipulating via `ax`:

```

fig, ax = plt.subplots(2, 2, figsize = (15,10))

ax[0][0].plot(gdp_uk['year'], gdp_uk['value'], label = 'U.K.') # the label 'U.K.' will be shown on the legend

ax[0][0].plot(gdp_fr['year'], gdp_fr['value'], '--', label = 'France') # '--' for dashed line

ax[0][0].legend()

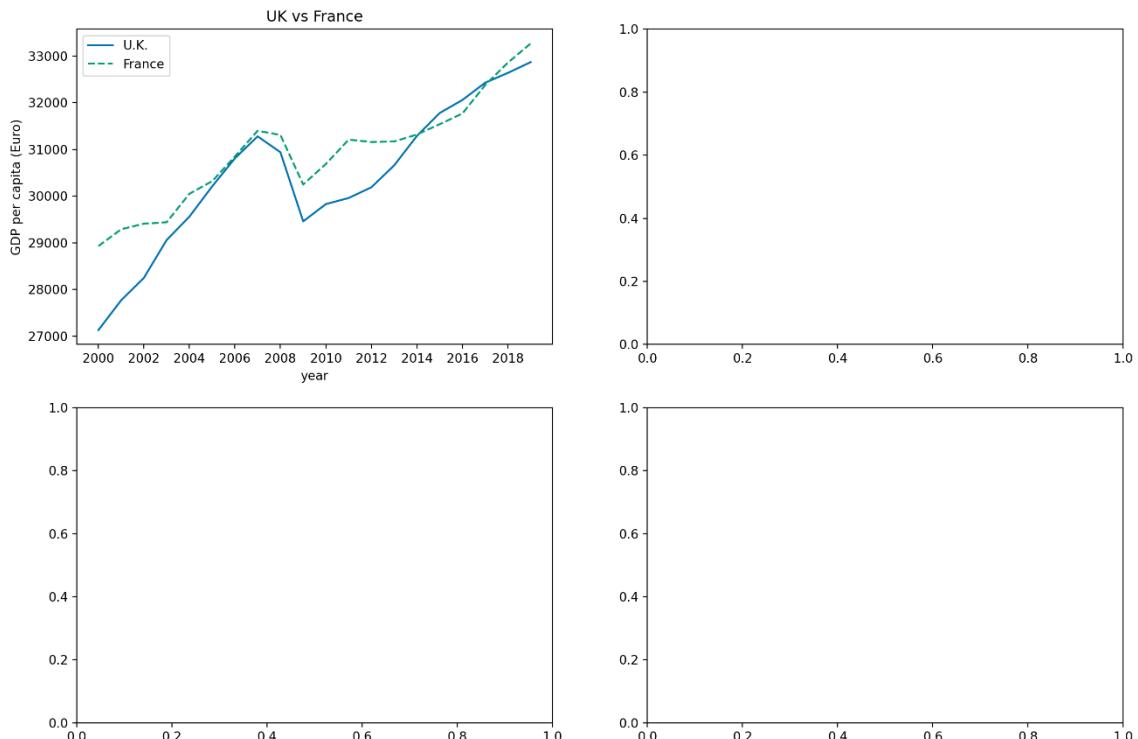
```

```

ax[0][0].set_xlabel("year")
ax[0][0].set_ylabel("GDP per capita (Euro)")
ax[0][0].title.set_text('UK vs France')

plt.show()

```



## Line Plot With plt

Above we have shown how to plot using `ax`, as it is handy to specify which subplot to plot by working on `ax`. If we only have one subplot, we can work on either `plt` or `ax`. They will give you the same figure, although the syntax is different. Below we show how the same plot can be made by `plt` (first plot) and `ax` (second plot):

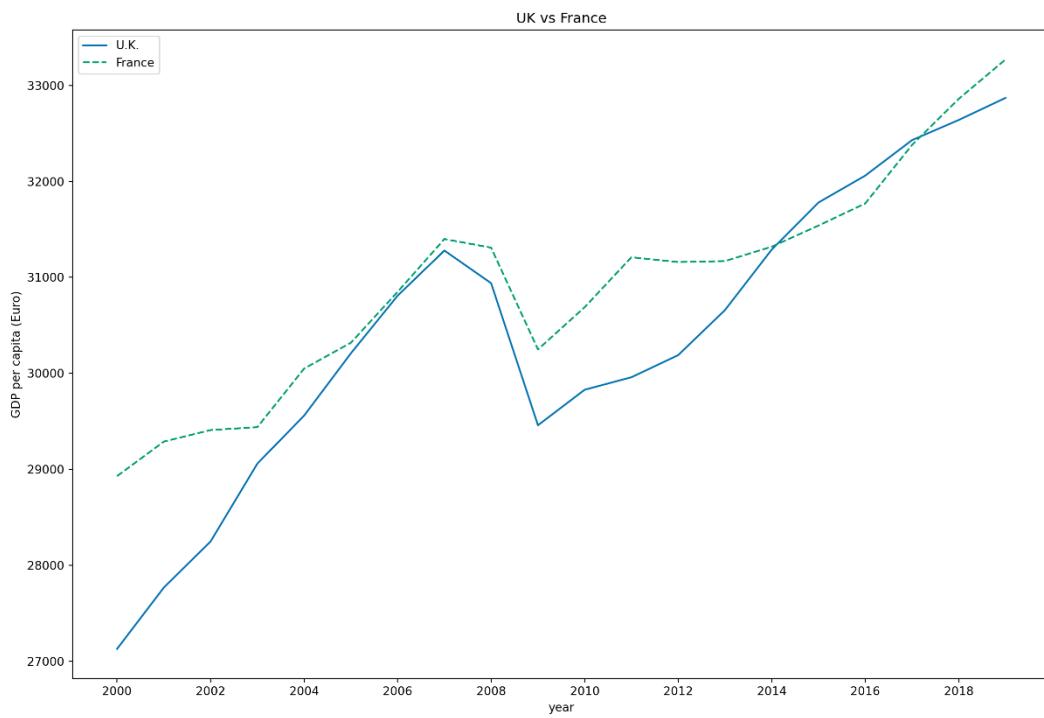
```

plt.plot(gdp_uk['year'], gdp_uk['value'], label = 'U.K.')
plt.plot(gdp_fr['year'], gdp_fr['value'], '--', label = 'France')
plt.legend()

plt.xlabel('year')
plt.ylabel('GDP per capita (Euro)')
plt.title("UK vs France")

```

```
plt.show()
```

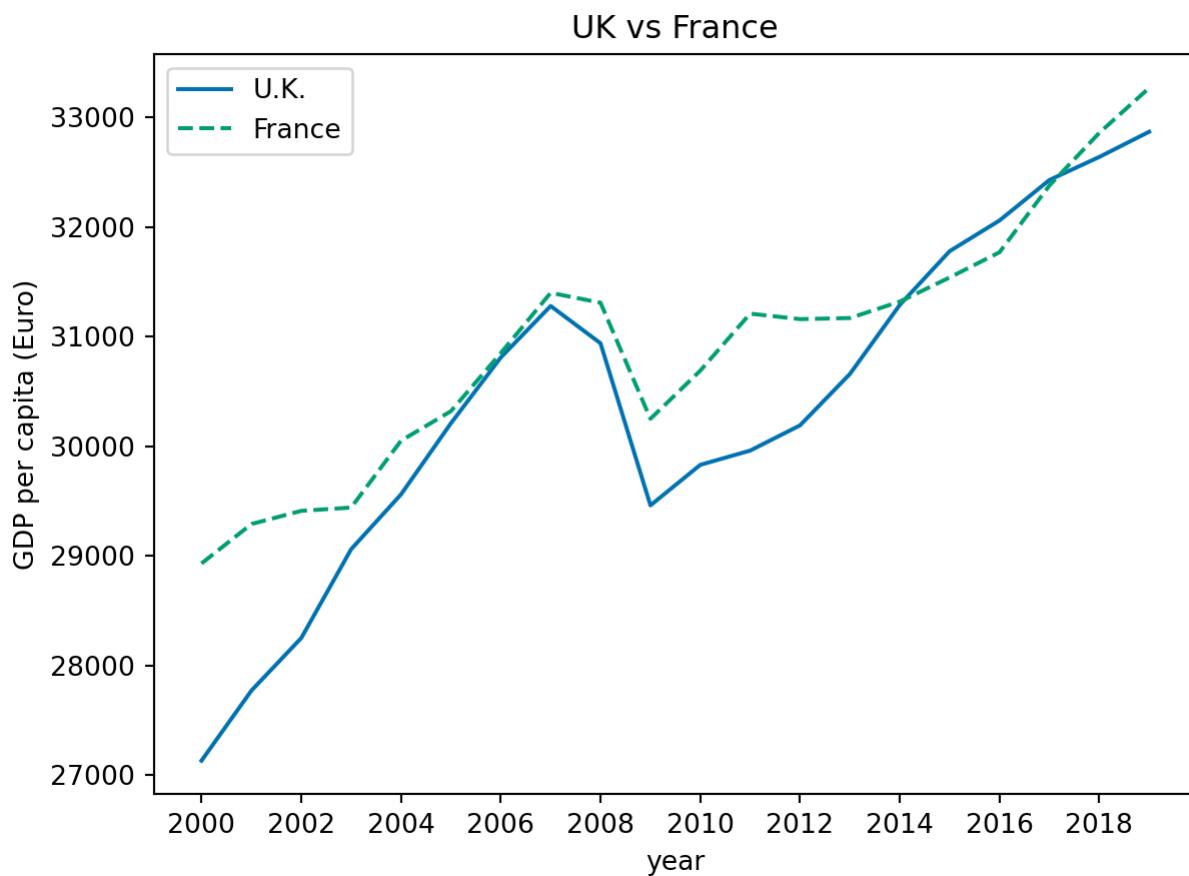


```
fig, ax = plt.subplots()

ax.plot(gdp_uk['year'], gdp_uk['value'], label = 'U.K.')
ax.plot(gdp_fr['year'], gdp_fr['value'], '--', label = 'France')
ax.legend()

ax.set_xlabel('year') # note it was plt.xlabel('year') above
ax.set_ylabel('GDP per capita (Euro)')

ax.title.set_text("UK vs France") # note it was plt.title("UK vs France") above
plt.show()
```



## Line Plot With `pandas` Data Structure

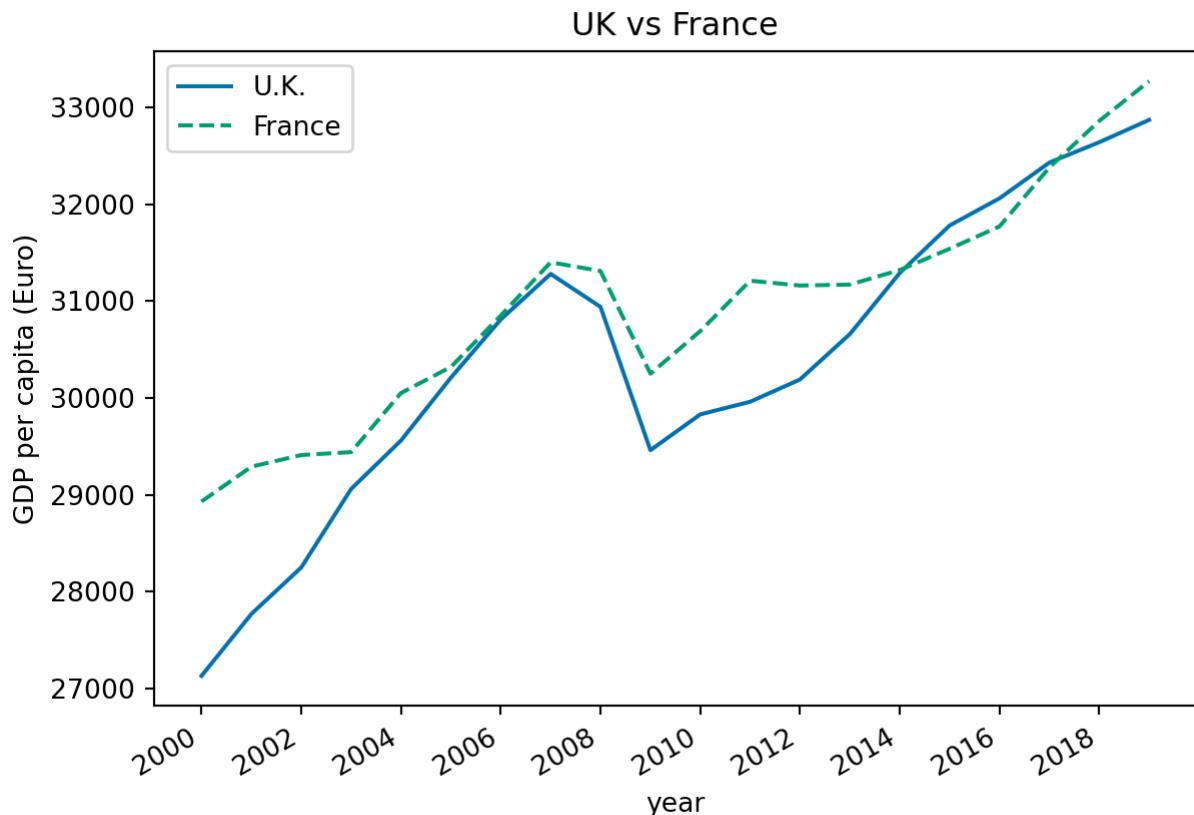
Alternatively, you can plot by using the method `plot()` from `pandas Series` or `DataFrame`. Behind the scene, `pandas` calls the plot functions from `matplotlib`, so you can consider `pandas`' `plot()` method is just a convenient shortcut to plot. As you can see below, plotting via `pandas` `plot()` method can create the same plot (slight different in figure size and the x label rotation for this particular example) as above when we use `matplotlib` `plot()` function. Sometimes, though, it is more convenient to plot via `pandas`. See the side by side bar plot example later in this page.

```
fig, ax = plt.subplots()

gdp_uk.plot(x = 'year', y = 'value', ax = ax)
gdp_fr.plot(x = 'year', y = 'value', style = '--', ax = ax)
ax.legend(["U.K.", "France"])

ax.set_xlabel('year')
ax.set_ylabel('Euro')
ax.set_ylabel('GDP per capita (Euro)')
ax.title.set_text("UK vs France")
```

```
plt.show()
```



## Bar Plot

Here we continue to plot the GDP per capital data, but now we are plotting against different countries rather than time. We can plot the bars vertically or horizontally:

```
fig, ax = plt.subplots(1, 2, figsize=(15, 4))

# ===== vertical bars =====

# ax = ax[0] indicates its the first subplot

ax[0].bar(gdp_wide[2019].index, gdp_wide[2019], alpha = 0.6, # the argument alpha = 0.6 makes the bars slightly transparent
           width = 0.5) # the argument width = 0.5 makes the bars thinner
ax[0].set_ylabel('GDP per capita (Euro)')
ax[0].set_xlabel('Countries')
ax[0].title.set_text("2019")

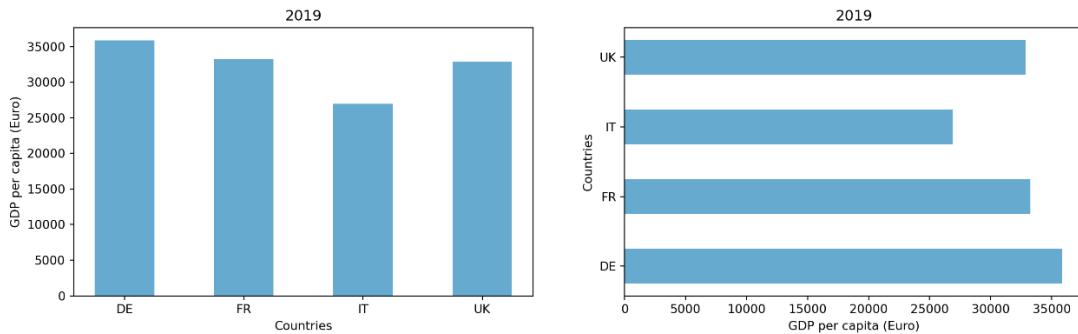
# ===== horizontal bars =====
```

```

# ax = ax[1] indicates its the second subplot
ax[1].barh(gdp_wide[2019].index, gdp_wide[2019], alpha = 0.6,
            height = 0.5) # note here we use "height" to make the bars thinner
ax[1].set_xlabel('GDP per capita (Euro)')
ax[1].set_ylabel('Countries')
ax[1].title.set_text("2019")

plt.show()

```



## Bar Plot With pandas

Like the line plots, we can plot bar plot using `DataFrame plot()` instead. Note the `plot()` arguments are different.

```

fig, ax = plt.subplots(1, 2, figsize=(15, 4))

# ====== vertical bars ======
gdp_wide[2019].plot.bar(x = 'country', y = 'value', rot = 0, # rot = 0: what if you
                        omit it?

                        ax = ax[0], # note we need to give ax as an argument
                        legend = False, alpha = 0.6) # legend = False: not to show
                        legend

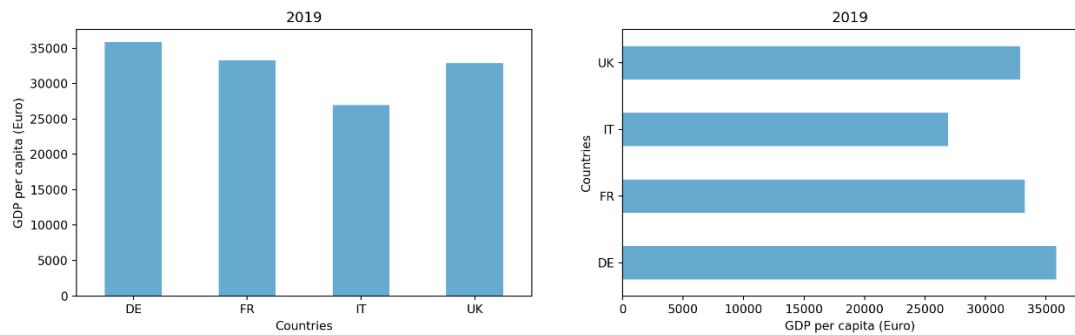
ax[0].set_ylabel('GDP per capita (Euro)')
ax[0].set_xlabel('Countries')
ax[0].title.set_text("2019")

# ====== horizontal bars ======
gdp_wide[2019].plot.barh(x='country', y='value', ax = ax[1], legend = False, alpha
= 0.6)

ax[1].set_xlabel('GDP per capita (Euro)')
ax[1].set_ylabel('Countries')
ax[1].title.set_text("2019")

```

```
plt.show()
```



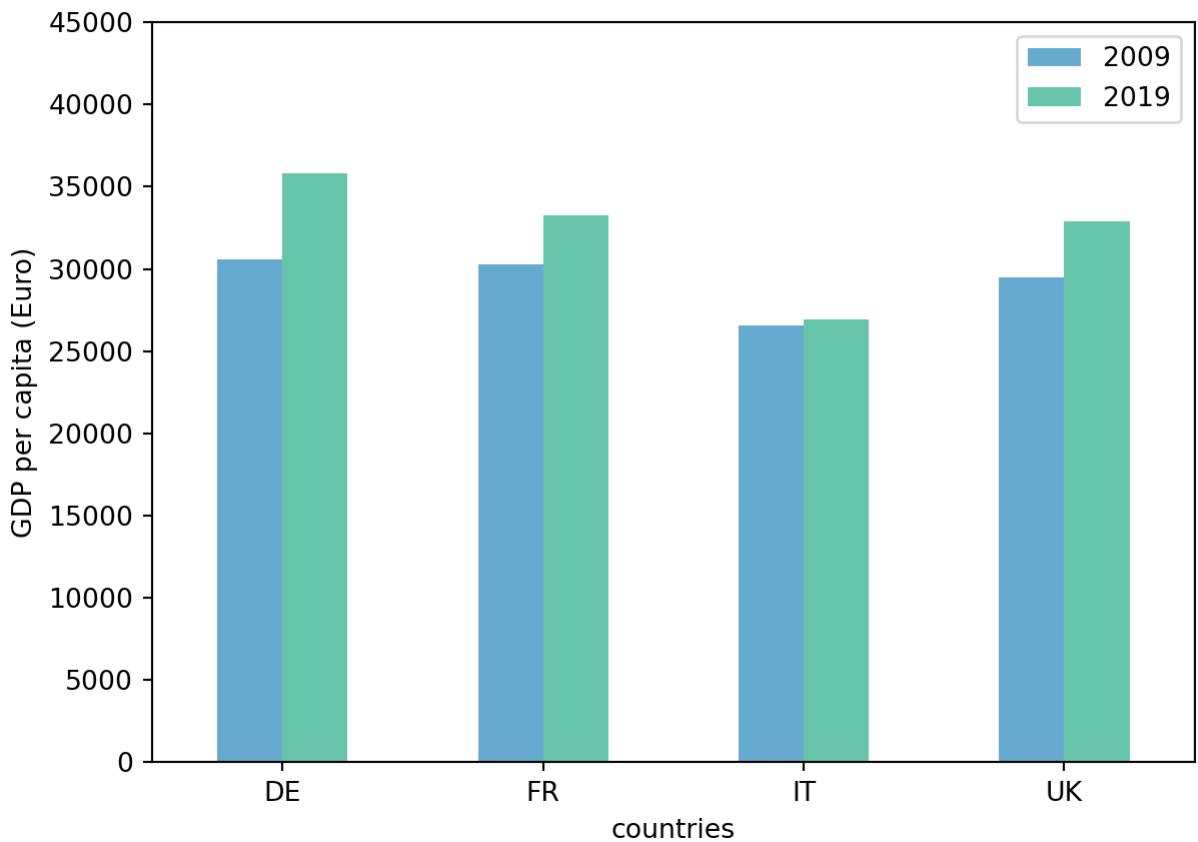
## Side By Side barplots

We can have side by side bars by provide using `DataFrame plot()` function. You could use `matplotlib plot()` but it is easier with `DataFrame plot()`. Here we plot the GDP per capital from year `2009` and `2019` side by side:

```
ax = gdp_wide[[2009,2019]].plot.bar(rot=0, alpha = 0.6)

ax.set_yticks([0, 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000])
ax.set_ylabel('GDP per capita (Euro)')
ax.set_xlabel('countries')

plt.show()
```



Note here we did not call `fig, ax = plt.subplots()` and then pass `ax` into `Dataframe plot()`. Instead we use the `ax` created from `Dataframe plot()`. We could of course use `fig, ax = plt.subplots()` and then pass `ax` into `Dataframe plot()` (as shown below), but we do not have to.

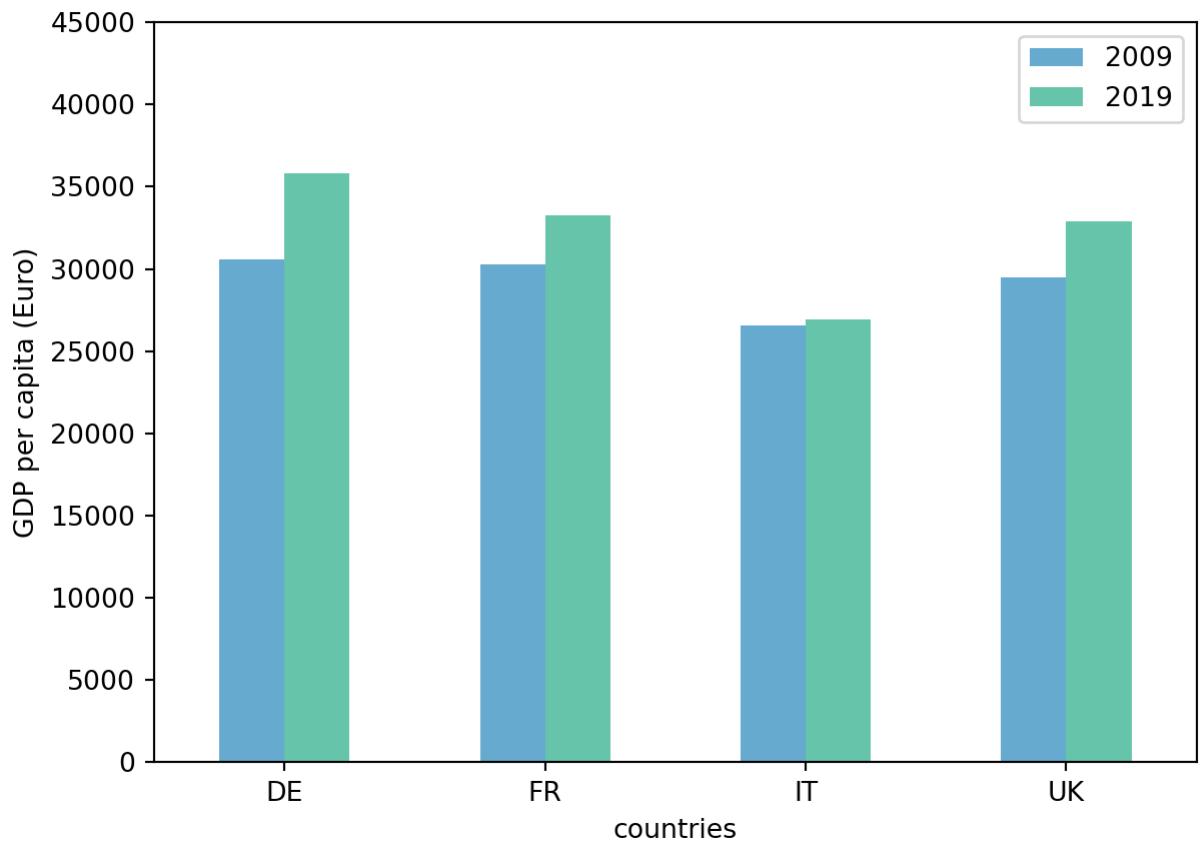
```
fig, ax = plt.subplots()

gdp_wide[[2009,2019]].plot.bar(rot=0, alpha = 0.6, ax = ax)

ax.set_ylim(top = 45000) # make ylim max to be larger so that the legend and the bars are not overlapping

ax.set_ylabel('GDP per capita (Euro)')
ax.set_xlabel('countries')

plt.show()
```



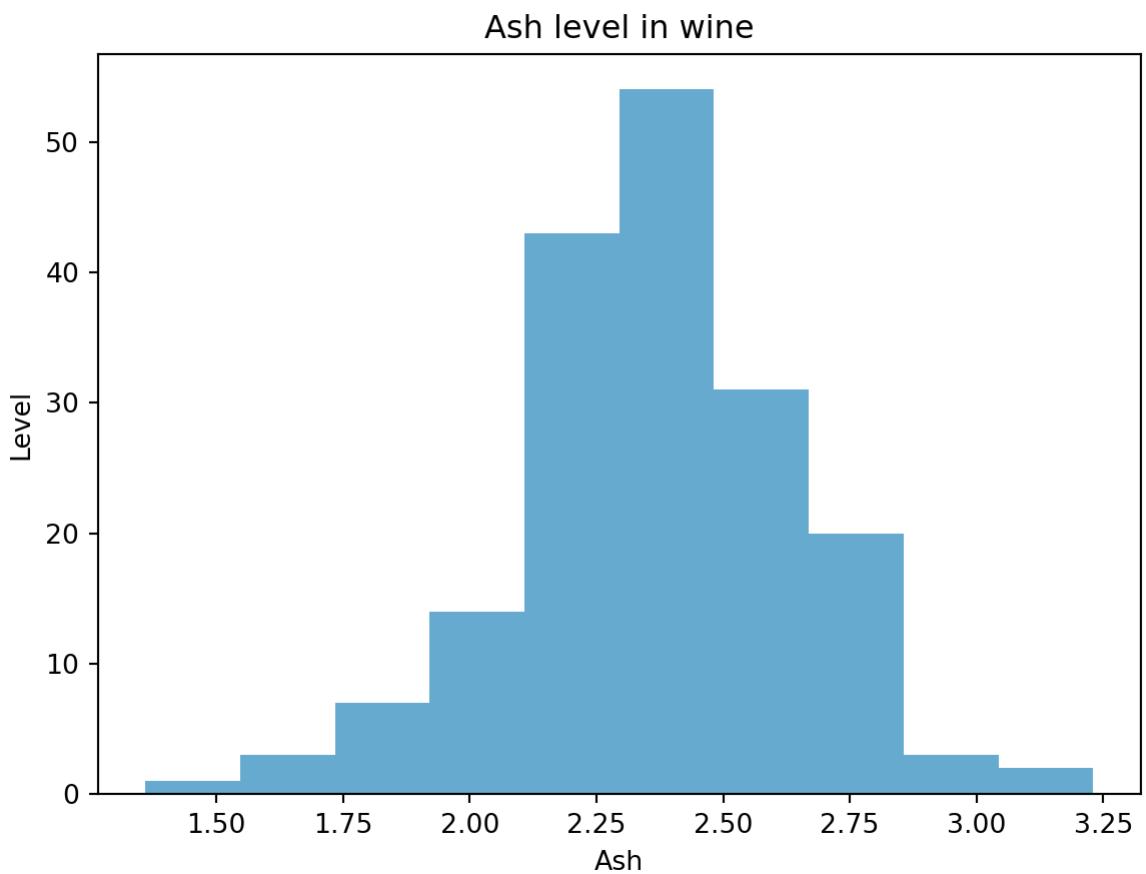
## Histogram: Showing the Distribution of a Variable

Below we create a histogram via `plt` to show the frequency of observations (here are the wines) that fall into a given `ash` level interval. You can use `ax` instead if you want to (although the syntax for setting labels and title is not the same).

```
fig, ax = plt.subplots()

plt.hist(wine['ash'], 10, alpha = 0.6)
plt.xlabel('Ash')
plt.ylabel('Level')
plt.title("Ash level in wine")

plt.show()
```

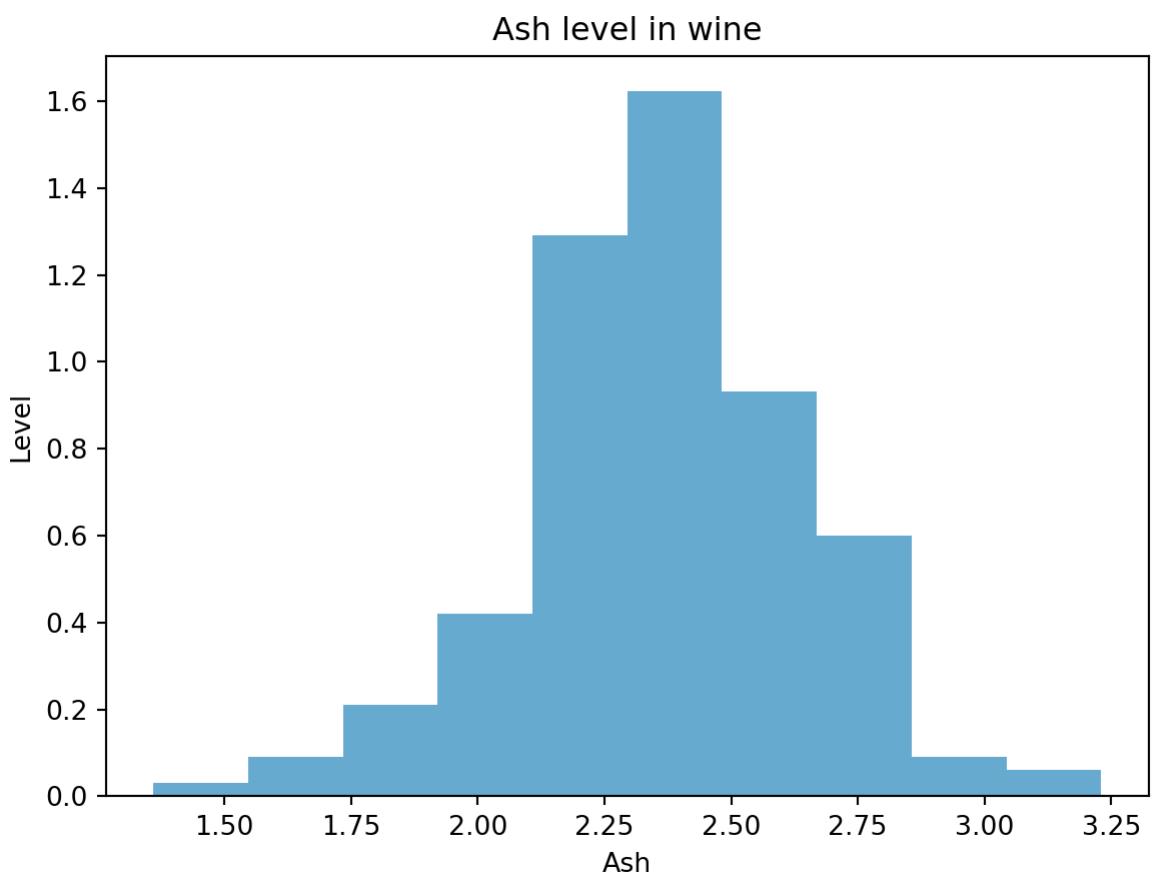


In order for the histogram to represent distribution (i.e. the area of the histogram is 1), add the argument `density = 1`:

```
fig, ax = plt.subplots()

plt.hist(wine['ash'], 10, density = 1, alpha = 0.6)
plt.xlabel('Ash')
plt.ylabel('Level')
plt.title("Ash level in wine")

plt.show()
```



## Boxplot and variation: showing the distribution of a variable

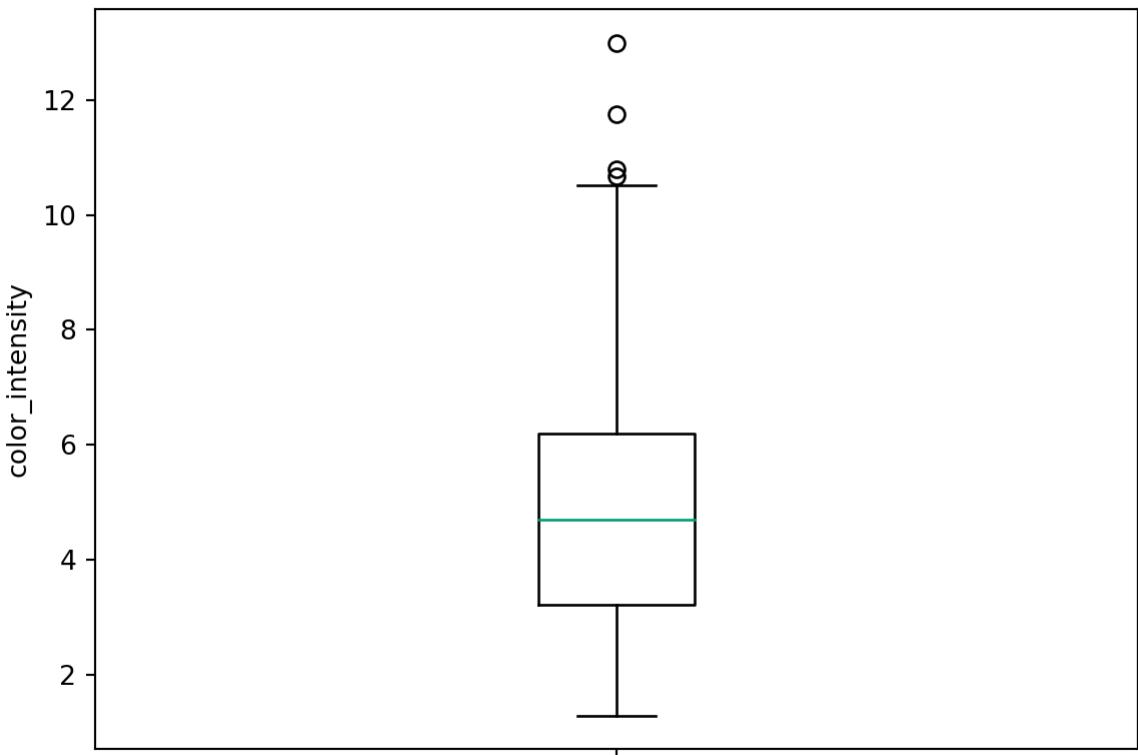
In Python boxplot (and its variations) can be created via **matplotlib** or **seaborn**. We will use the boxplots to compare the chemistry composition of wines with different class labels.

We start at looking at the colour intensity regardless the class label:

```
fig, ax = plt.subplots()

ax.boxplot(wine['color_intensity'])
ax.set_ylabel('color_intensity')
ax.set_xticklabels([''])

plt.show()
```

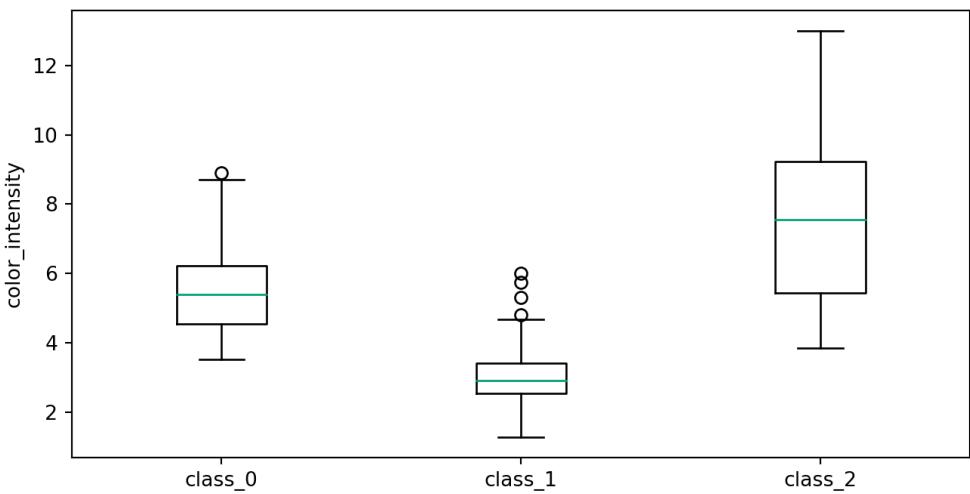


## Side By Side (or grouped) Boxplot

Often, it is more useful to have boxplots side by side to compare the same variables conditional on some factors. For example, here we want to see the colour intensity of wines given the type of the wine. With **matplotlib**, we need to first convert our long `wine['color_intensity']` data to wide, and then create the boxplot:

```
fig, ax = plt.subplots(figsize = (8, 4))

ax.boxplot([wine['color_intensity'][wine['target']=='class_0'],
            wine['color_intensity'][wine['target']=='class_1'],
            wine['color_intensity'][wine['target']=='class_2']])
ax.set_ylabel('color_intensity')
ax.set_xticklabels(['class_0', 'class_1', 'class_2'])
plt.show()
```



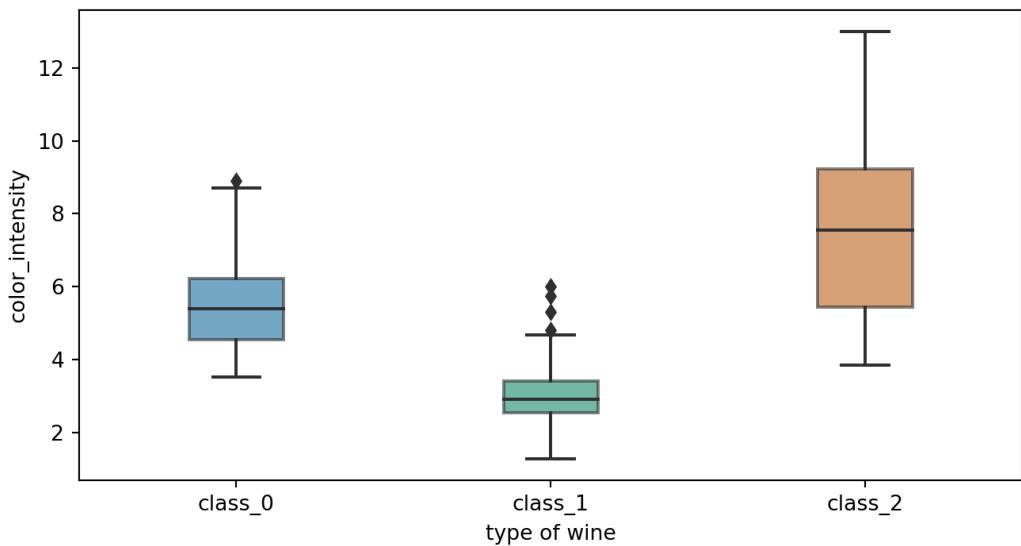
Side by side boxplot can be created more easily via **seaborn**. With **seaborn**, we can use the original `wine DataFrame` ('`target`' column in the `wine DataFrame` contains the class label information):

```
fig, ax = plt.subplots(figsize = (8, 4))

sns.boxplot(data = wine, x = 'target', y = 'color_intensity', width = 0.3,
            boxprops = dict(alpha=0.6)) # note in seaborn, alpha (and other parameters) is set in a different way

ax.set_xlabel("type of wine")

plt.show()
```



## Boxplot Variation

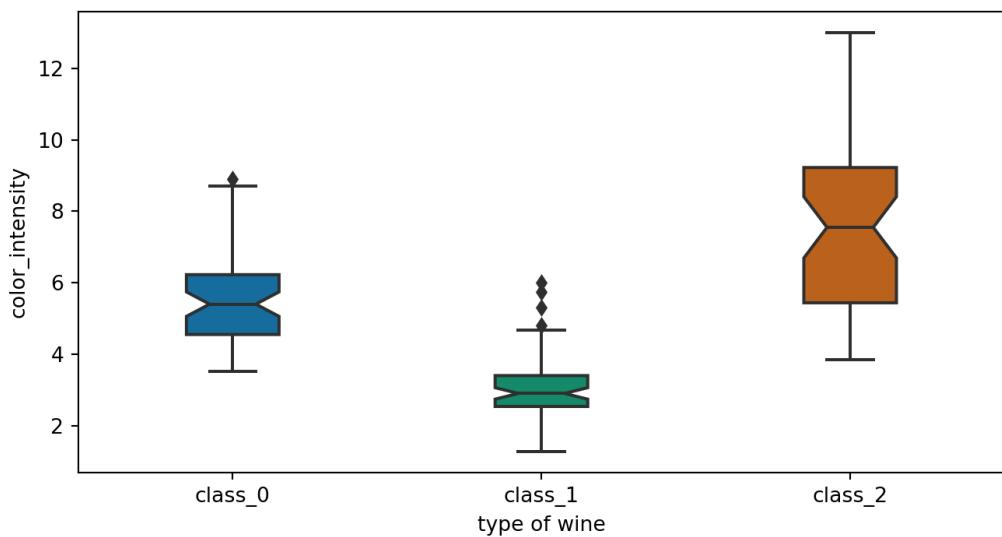
To create a notched boxplot, add the argument `notch = True`:

```
plt.subplots(figsize=(8, 4))

ax = sns.boxplot(data = wine, x = 'target', y = 'color_intensity', notch = True, width = 0.3)

ax.set_xlabel("type of wine")

plt.show()
```



To create a violin plot, we use the `violinplot()`. Below boxplot is plotted as well to show you how you can have subplots with **seaborn**:

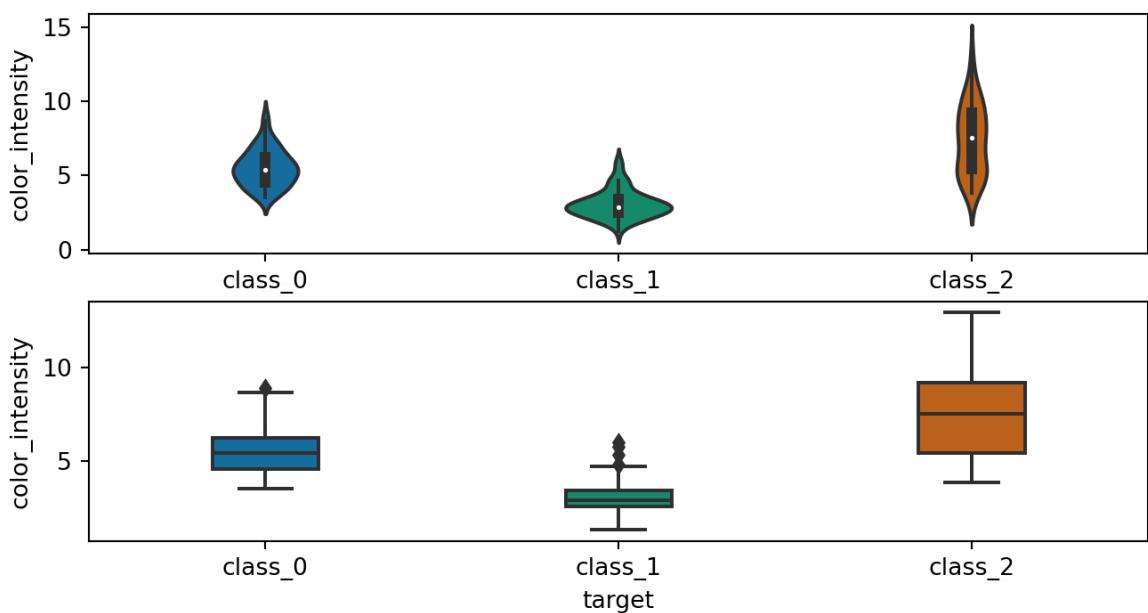
```
fig, ax = plt.subplots(2, 1, figsize=(8, 4))

sns.violinplot(data = wine, x = 'target', y = 'color_intensity', ax = ax[0], width = 0.3)

sns.boxplot(data = wine, x = 'target', y = 'color_intensity', ax = ax[1], width = 0.3)

ax[0].set_xlabel("type of wine")

plt.show()
```

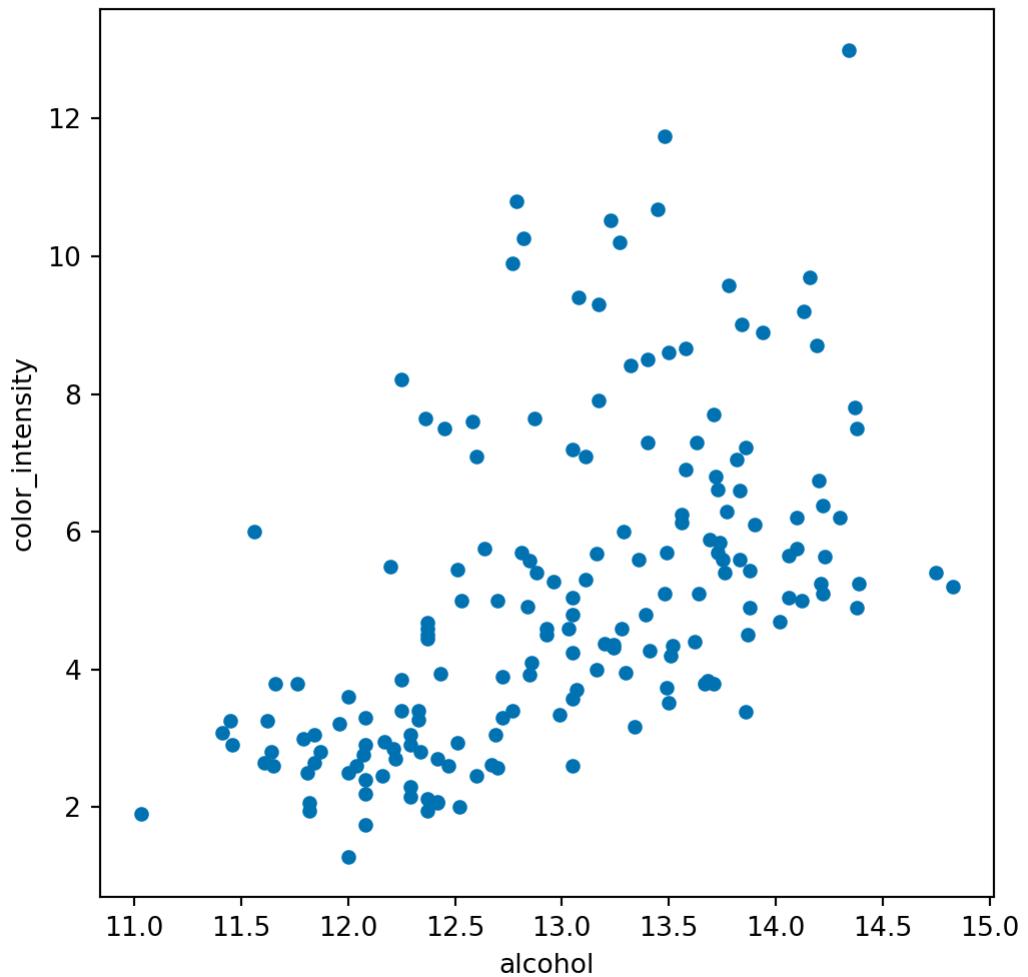


## Scatter: Showing Relations Between Two Variables

Below we create a scatter plot via `DataFrame.plot()` to show how the two chemistry composition levels relate to each other (you can use `matplotlib` instead if you want to).

```
ax = wine.plot.scatter('alcohol', 'color_intensity', figsize=(6, 6)) # use a square
plot size

plt.show()
```

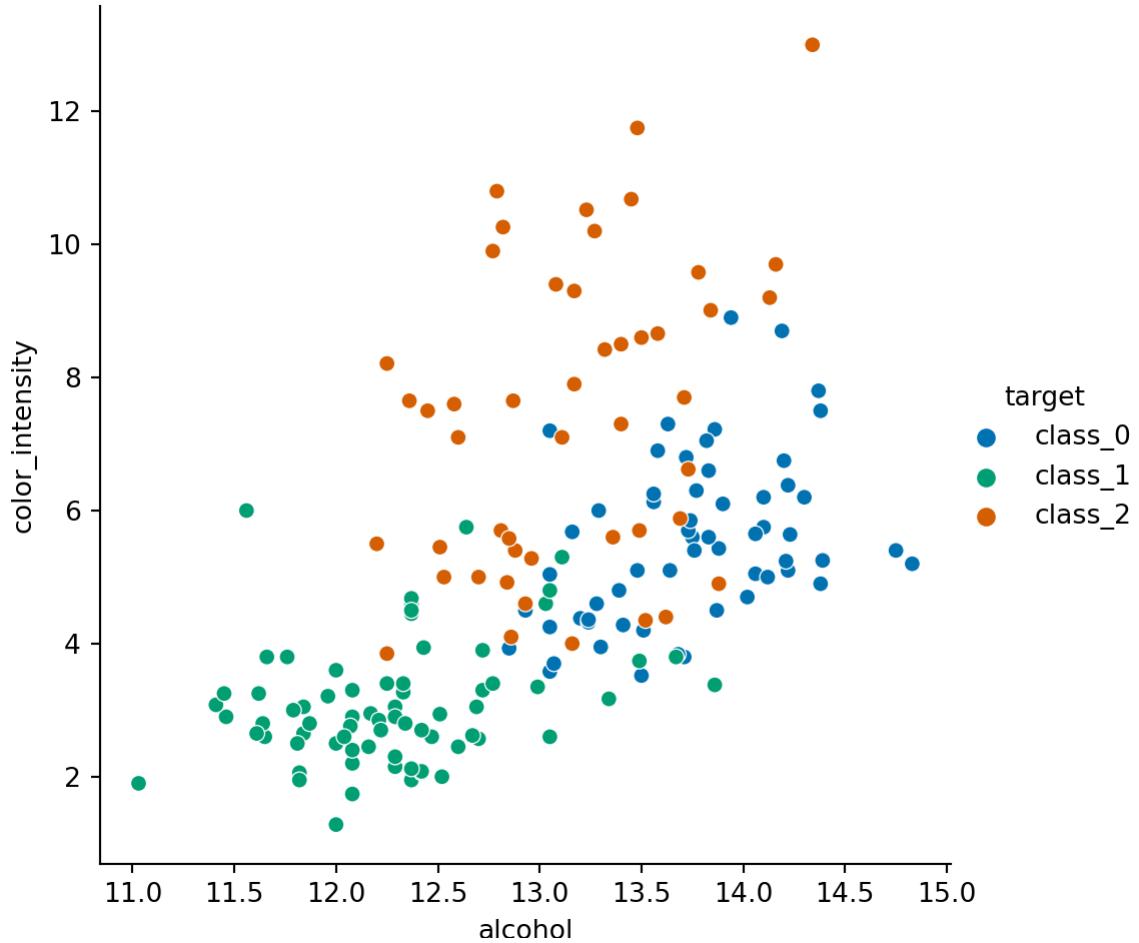


## Scatter Plot With **seaborn**

With **seaborn**, we can use `relplot()` to plot the scattered plot. We can set the optional argument `hue`, `style`, `size`, etc to show an extra dimension of the data.

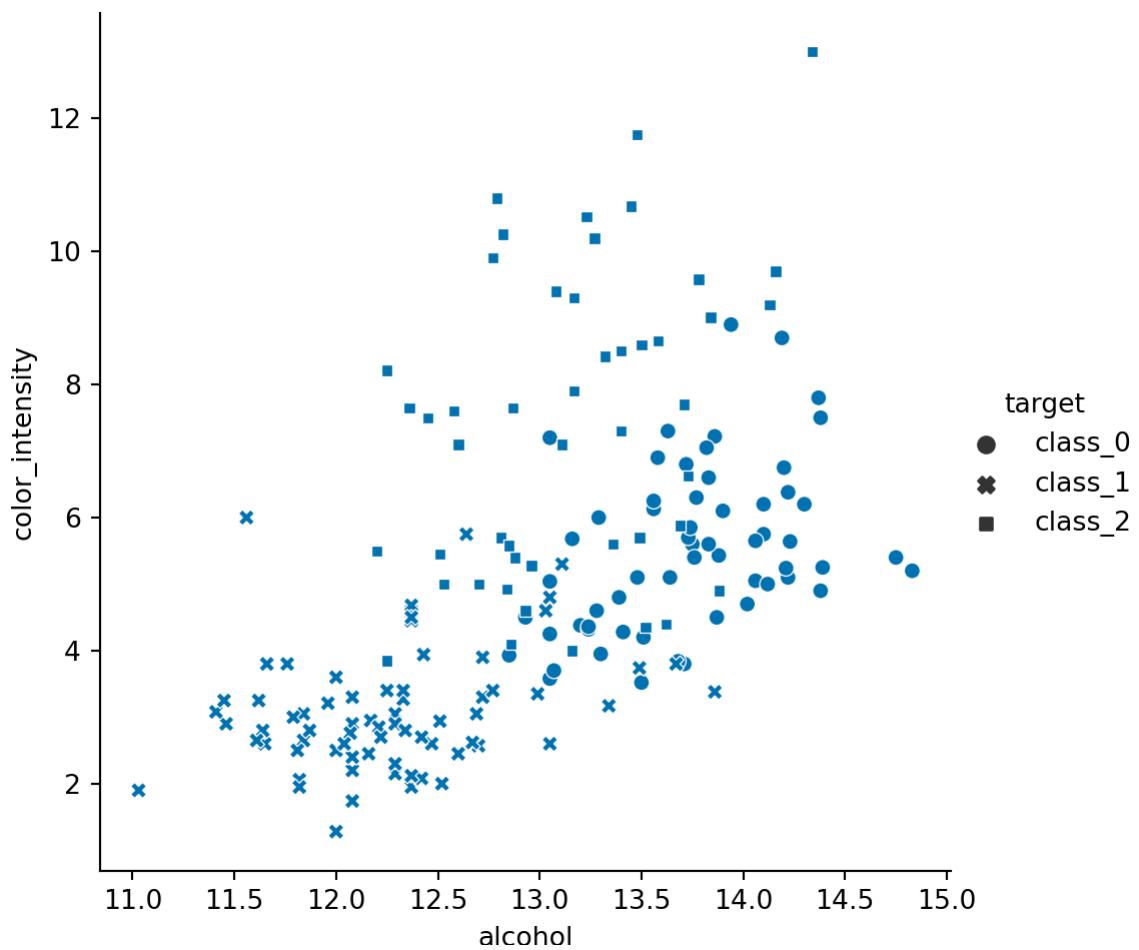
For example, here we plot the `color_intensity` against `alcohol`, with the colour of scattered points depends on which group the wines belong to by providing the optional argument `hue`:

```
sns.relplot(x = 'alcohol', y = 'color_intensity', hue = "target", data = wine)
plt.show()
```



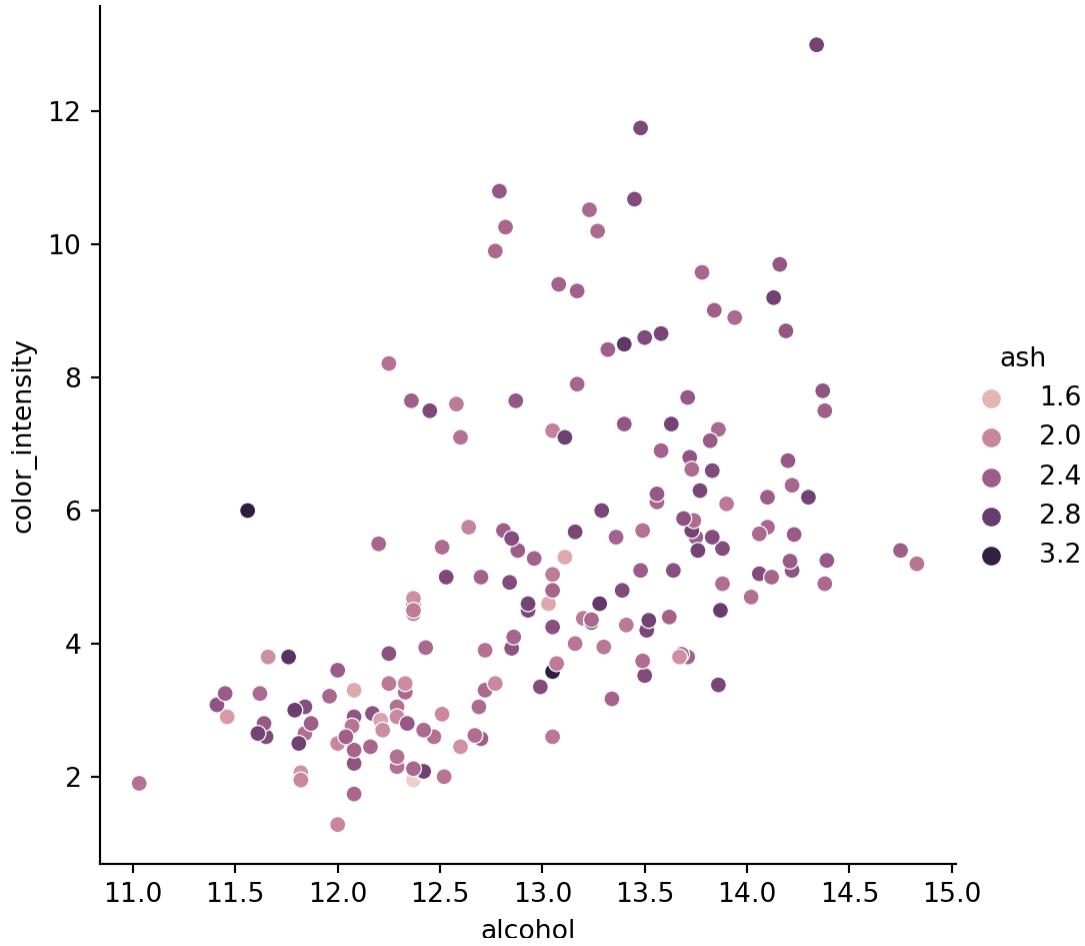
We can also use the shape instead of colour to represent the group membership by `style`:

```
sns.relplot(x = 'alcohol', y = 'color_intensity', style="target", data = wine)  
plt.show()
```



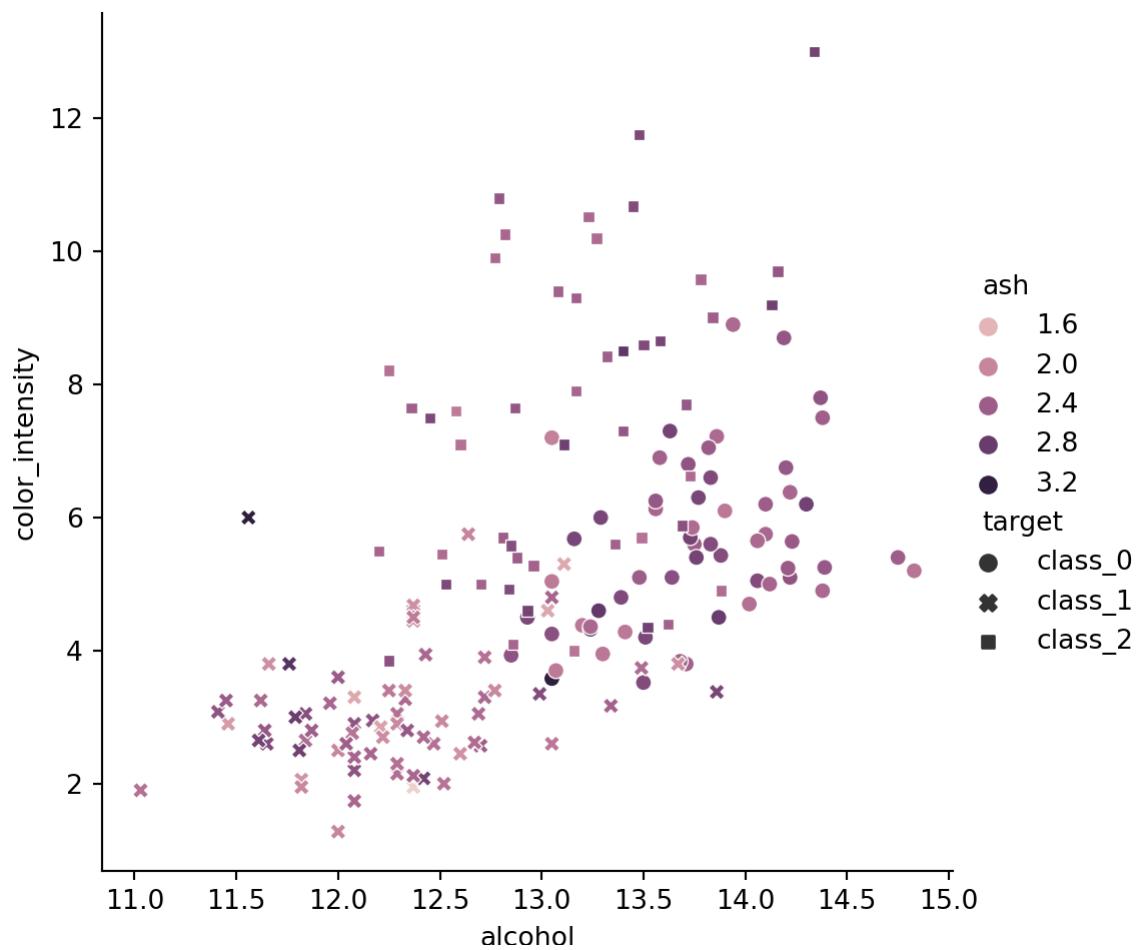
The optional argument `hue` can also be used for continuous variable. Here the colour of the points depends on the value of `ash`

```
sns.relplot(x = 'alcohol', y = 'color_intensity', hue = "ash", data = wine)  
plt.show()
```



We can also have a graph showing four different variables by using both `hue` and `style`:

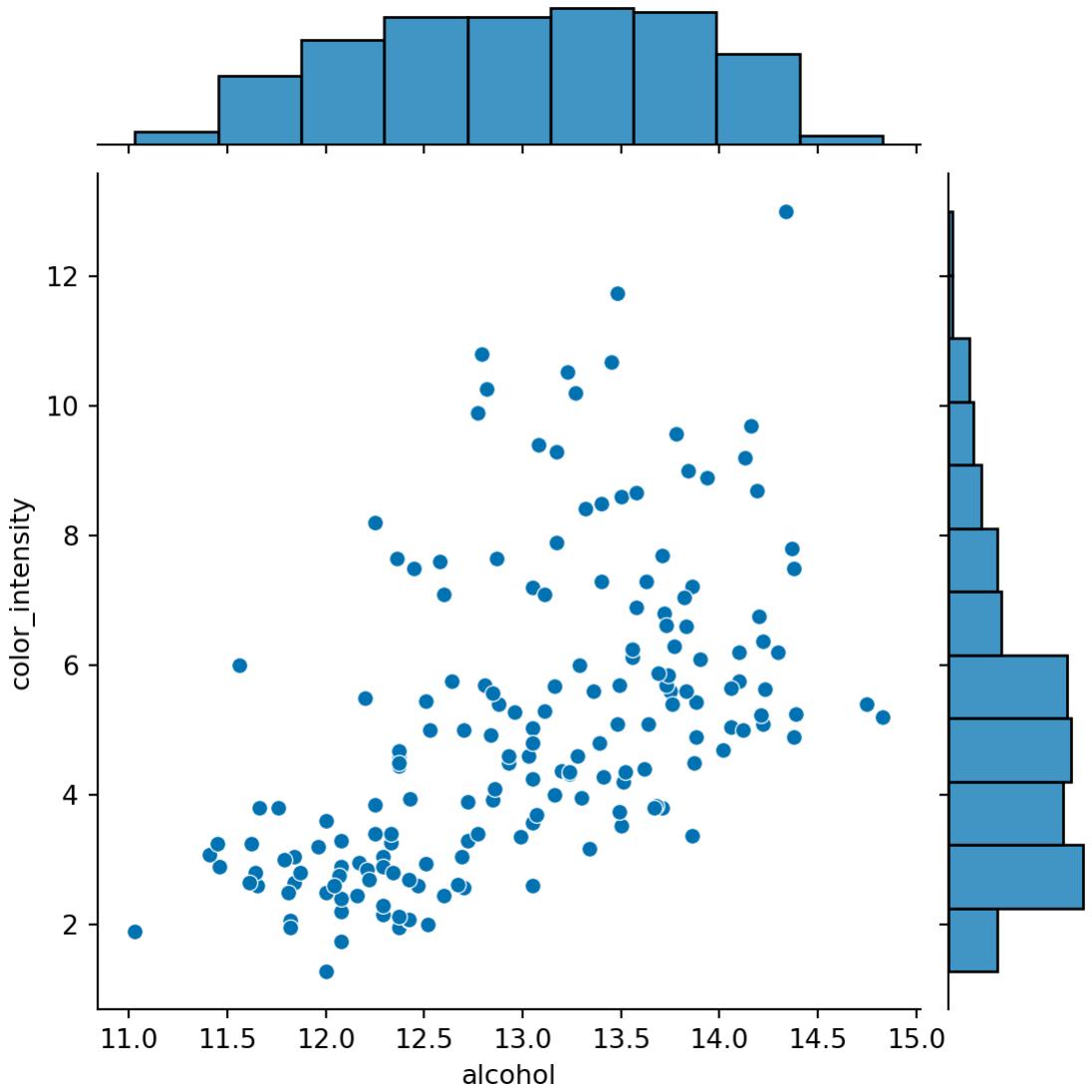
```
sns.relplot(x = 'alcohol', y = 'color_intensity', hue = "ash", style = 'target', data = wine)
plt.show()
```



## More on Scatterplots With `seaborn`

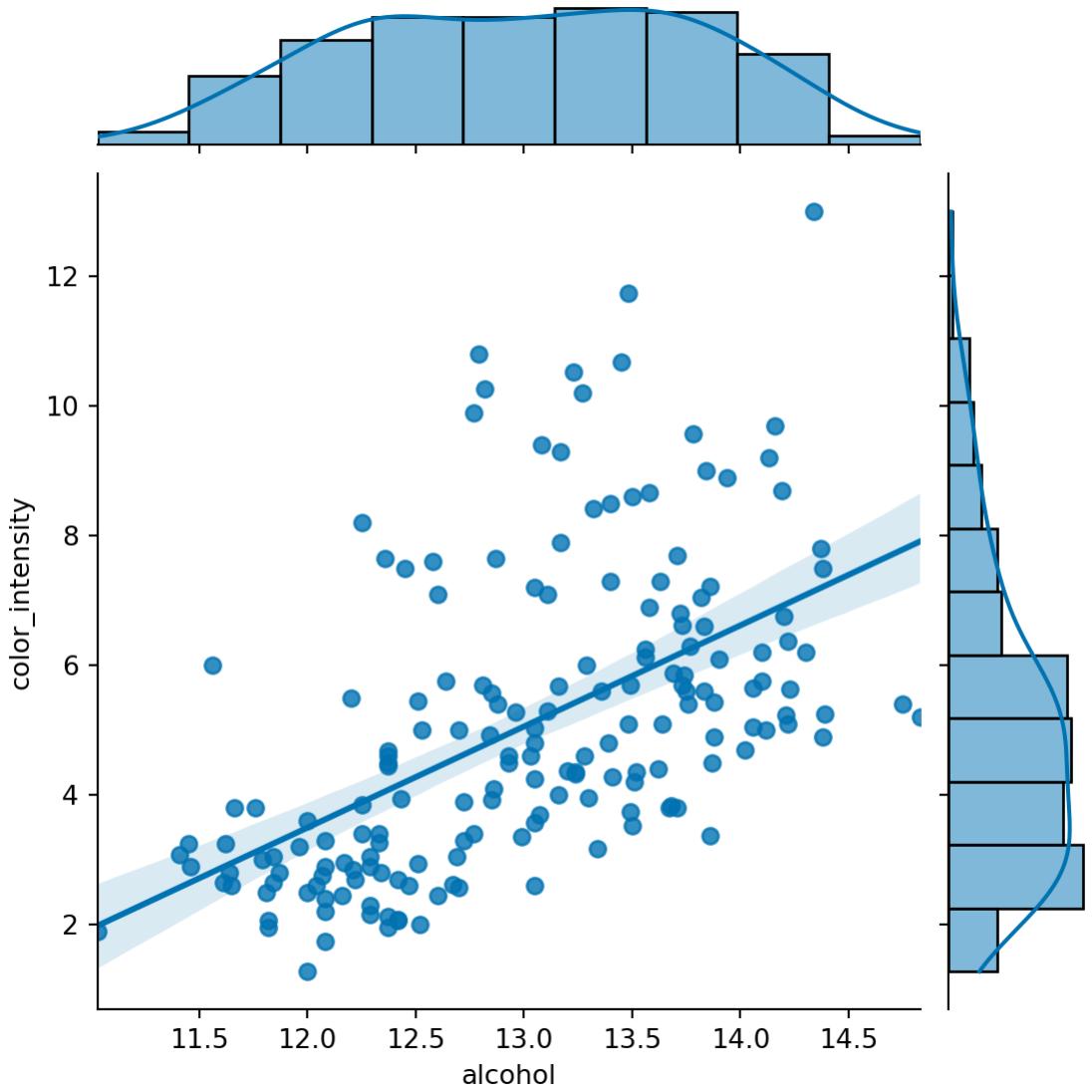
With **seaborn**, you can do some fancier scatter plots. For example, showing the marginal histogram together the scatterplot:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine)
plt.show()
```



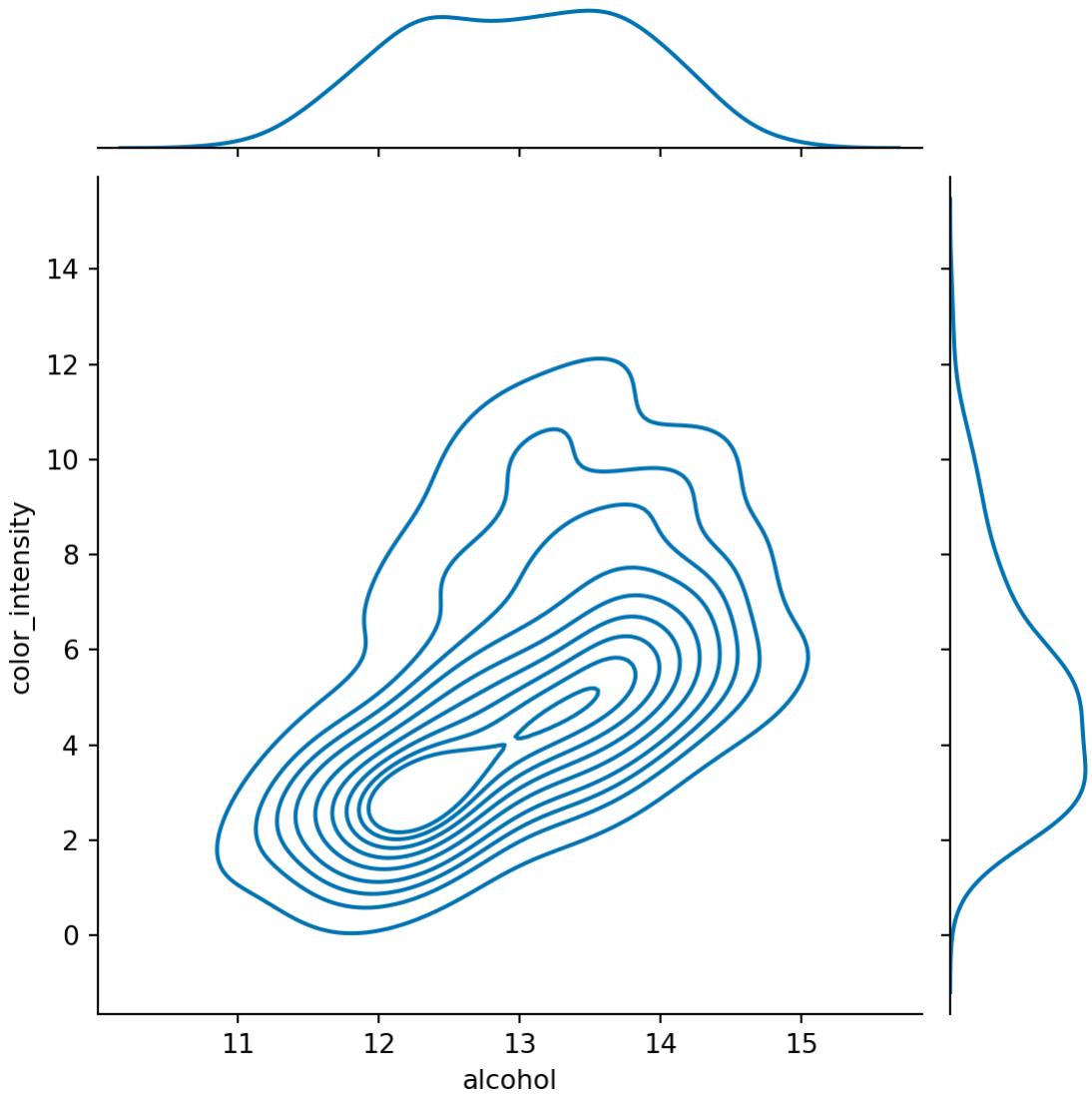
We can also add a linear regression fit and univariate KDE curves using `kind='reg'`:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, kind="reg")
plt.show()
```



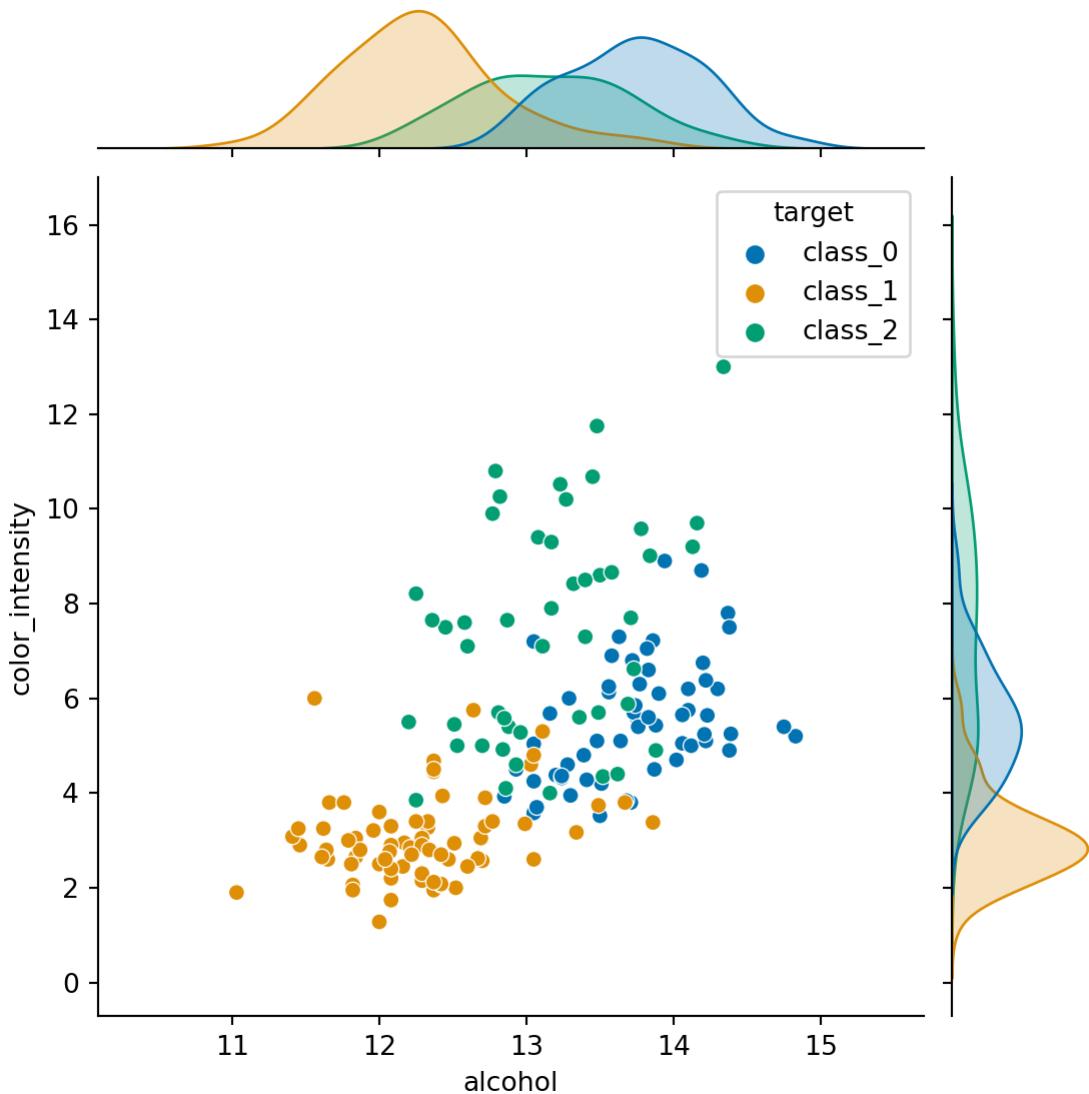
... or with the bivariate and univariate Kernel density estimation:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, kind = 'kde')
plt.show()
```

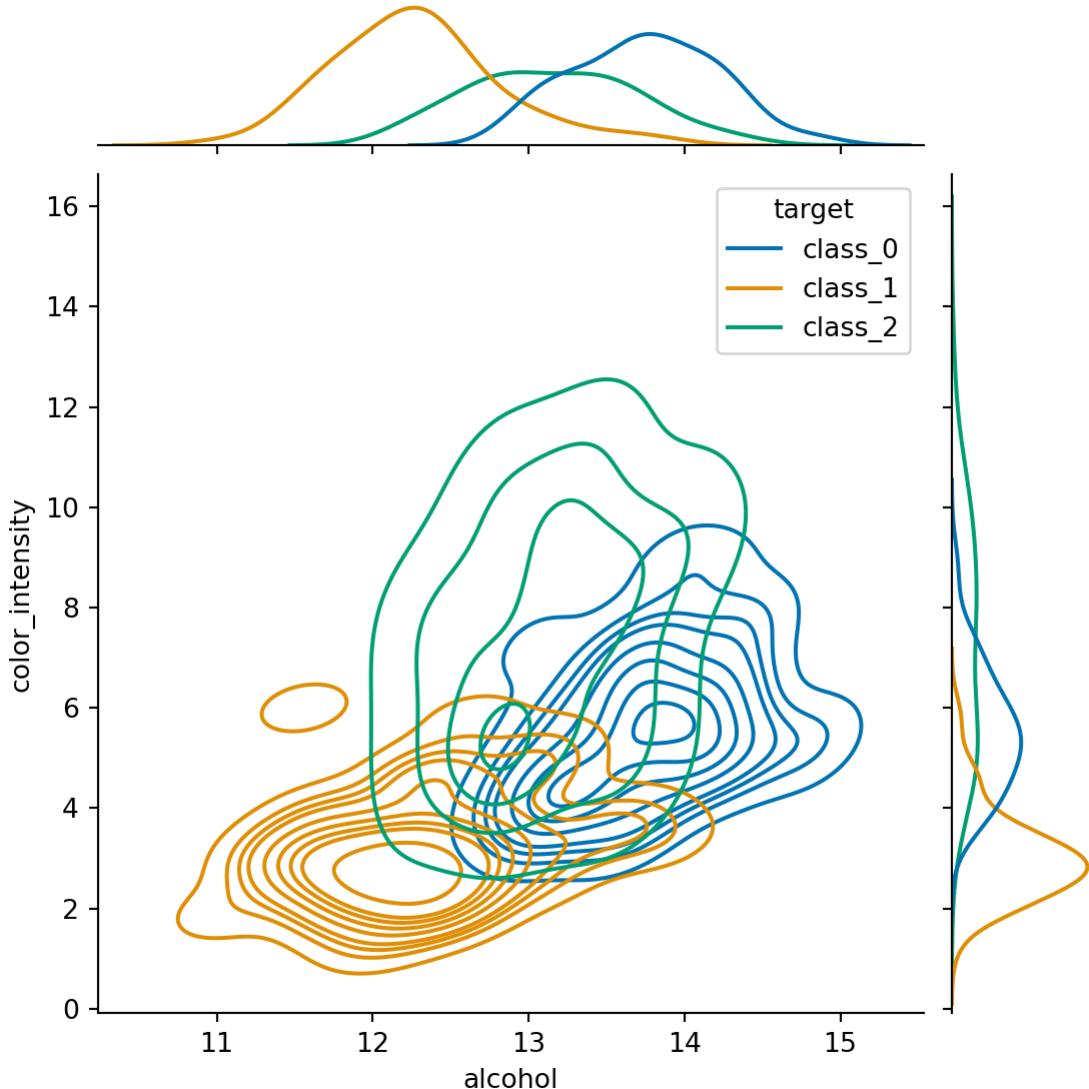


You can have show the scatter points in different colours according to the label as well:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, hue = "target", palette = my_palette[:3])  
plt.show()
```



```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, kind = 'kde', hue = "target", palette = my_palette[:3])  
plt.show()
```



See [here](#) to see what other scatter plot you can create with **seaborn**.

## Heat Map

To create a heatmap, we can use `sns.heatmap()`. Here we create a heatmap to show the correlation of the wine chemistry composition:

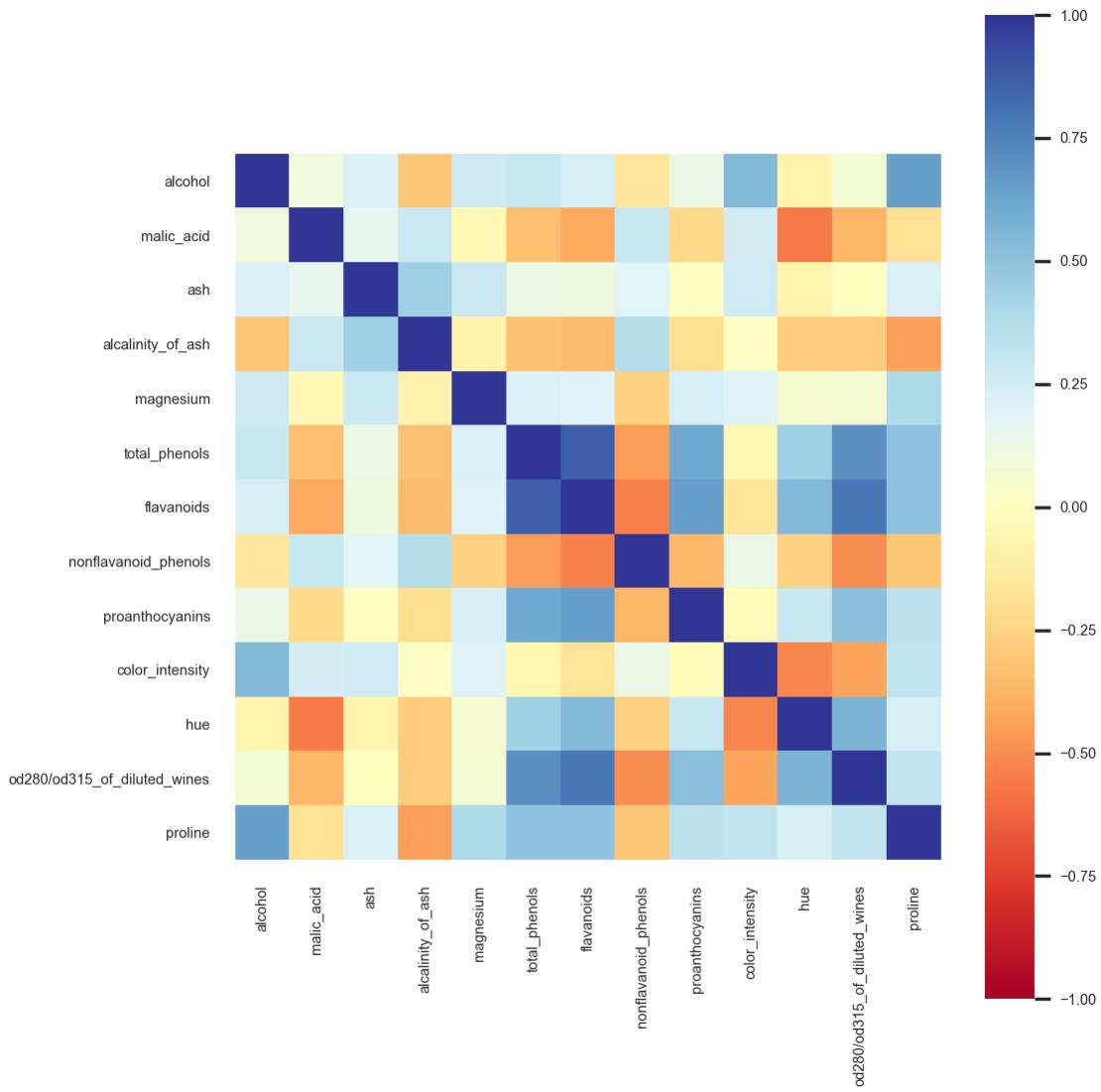
```

sns.set(font_scale=0.5)

ax = sns.heatmap(wine.corr(),
                  center= 0, vmin = -1, vmax = 1, cmap= "RdYlBu", # this set the colour of the heatmap
                  square=True) # set the heatmap to be square shape

ax.figure.tight_layout() # this makes sure all labels are shown in the plot
plt.show()

```



The argument `cmap= "RdYlBu"` sets the palette, with red corresponds to lowest value, yellow corresponds to the middle value and blue corresponds to the highest value. `center = 0` sets 0 as the middle value, and `vmax = 1` and `vmax = -1` set the maximum and minimum values to be 1 and -1. To set heatmap with different colours, see [here](#).

## Exporting Plots

Similar to R, you can save the figure as a separate file. In Python, you can do it by using the function `savefig()`:

```
fig, ax = plt.subplots()

gdp_uk.plot(x = 'year', y = 'value', ax = ax)
gdp_fr.plot(x = 'year', y = 'value', style = '--', ax = ax)
ax.legend(["U.K.", "France"])

ax.set_xlabel('year')
```

```
ax.set_ylabel('Euro')
ax.set_ylabel('GDP per capita (Euro)')
ax.title.set_text("UK vs France")

plt.savefig('gdp_uk_france.png') # you can find the file in the folder where you run this line of code
```

## Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapter 8.
- [Matplotlib tutorial](#)
- [seaborn jointplot](#)

# Network Visualisations

\*\* Note: The code chunks below should be run in the following order \*\*

## Networks

A network  $\langle G \rangle$  consists of a set  $\langle V \rangle$  of vertices and a set  $\langle E \rangle$  of edges. An edge indicates a relationship between two vertices. Networks provide standard tools for visualisation in data science and related fields. Below are some examples of kinds of real world network data that can be represented by networks:

- Social networks: Representing people and their social interactions. Examples include looking into Facebook, LinkedIn, email exchanges, etc.
- Knowledge networks: Representing entities and their relationships. For example Google knowledge graph, Bing Satori, Freebase, Yago, Wordnet, etc.
- Collaboration networks: Representing people and their collaborations. Examples include Co-authorships from dblp, Google Scholar, Microsoft Academic search, etc.
- Product purchase networks: Representing who bought what. Example: Amazon product purchases
- Reviews: Ratings of products or services provided by users. Example: TripAdvisor
- Road networks, communication networks.

There are several reasons for visualising networks:

- For exploratory data analysis by visual inspection of network drawings
- For visual detection of network structures, e.g. community detection in social networks, tree or feed-forward structures.
- For communicating network properties

For more on networks you can read Robinson et al. (2015).

## Networks in Python

The standard Python package for the creation, manipulation, and study of the structure of networks is **networkx**. A tutorial for it can be found [here](#). To install **networkx** type the following in the terminal or Anaconda prompt:

```
pip install networkx
```

Below are some standard layout styles for networks produced by the **networkx** package:

- `circular`: Position nodes on a circle
- `random`: Position nodes uniformly at random in the unit square
- `shell`: Position nodes in concentric circles
- `spring`: Position nodes using the Fruchterman-Reingold force-directed algorithm
- `spectral`: position nodes using the eigenvectors of the network Laplacian

We will illustrate the above in a real world dataset. In next two subsections, we will explore and pre-process the data and then we will sample from the above styles.

# GitHub Organisations Data

As a working example, we will study the activities of users across different GitHub organizations by a representation of data as a network. The vertices of the network will represent organisations and an edge between two vertices will indicate that there is at least one user who initiated an event to at least one repository in each of the two corresponding organizations. The edge weights are defined as the number of such users. We will use the data retrieved from GitHub archive for 2 March 2015 that are contained in the attached file [2015-03-02.csv](#).

We will need to do a fair amount of pre-processing to bring the data in the desired format. This can be seen as a good data wrangling exercise. First, we initialise Python and load the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# load the data
fdate = '2015-03-02'
df = pd.read_csv(fdate + '.csv', parse_dates=['created_at'])
df.head()
```

	created_at	id	actor.login	org.login	repo.name	type
0	2015-03-02	2615962109	pdgago	simbiotica	simbiotica/charicharts	CreateEvent
1	2015-03-02	2615962111	micmania1	NaN	micmania1/silverstripe-cms	PushEvent
2	2015-03-02	2615962112	xsb	bitpay	bitpay/insight-api	WatchEvent
3	2015-03-02	2615962113	dkorolev	KnowSheet	KnowSheet/Bricks	PullRequestReviewCommentEvent
4	2015-03-02	2615962114	edwin-pers	NaN	edwin-pers/neuromancer	PushEvent

The dataset contains several variables indicating the user (`actor.login`) and the GitHub organisation (`org.login`) as well as other variables such as date created, ID, name of the repository and type of event. To check which types of events exist we use the code below:

```
# unique event types
df['type'].unique()
array(['CreateEvent', 'PushEvent', 'WatchEvent',
       'PullRequestReviewCommentEvent', 'IssueCommentEvent',
       'PullRequestEvent', 'DeleteEvent', 'CommitCommentEvent',
       'ForkEvent', 'IssuesEvent', 'GollumEvent', 'MemberEvent',
       'ReleaseEvent', 'PublicEvent'], dtype=object)
```

Next, we want to transform the data to triplets: (actor, organization, repository). Moreover, we remove events of type `WatchEvent`, drop duplicates and also remove missing values from `org.login`.

```
df = df[(df['type']!='WatchEvent')]
df = df[['actor.login', 'org.login', 'repo.name']]
```

```

df = df.drop_duplicates()
df = df[df['org.login'].notnull()]
df.head()

```

	actor.login	org.login	repo.name
0	pdgago	simbiotica	simbiotica/charicharts
3	dkorolev	KnowSheet	KnowSheet/Bricks
5	dmolsen	pattern-lab	pattern-lab/patternengine-php-twig
7	LordSputnik	BookBrainz	BookBrainz/bookbrainz.org
16	ScottNZ	OpenRA	OpenRA/OpenRA

Next, we want to focus on the most frequently occurring organisations, say the top 100. We will do so using the function `groupby()`.

```

org_actor_df = df.groupby(['org.login','actor.login']).agg('count').reset_index()
toporgs = org_actor_df.groupby(['org.login'])['org.login'].agg('count')
toporgs = toporgs.sort_values(ascending=False)[:100].keys().tolist()
toporgs[:10]
['mozilla', 'apache', 'facebook', 'google', 'angular', 'docker', 'elasticsearch', 'iojs', 'dotnet', 'owncloud']

```

The first line in the code above takes all possible pairs of `org.login` and `actor.login` and applies the operation `.agg('count')` that counts the frequency of each pair. The second line takes the frequency of each organisation and stores them into `toporgs`. Finally we sort the organisations in terms of frequency of appearance and print the top 10 of them.

The next step is almost taking us to our objective as we track the organisations where each user initiated an event and record all possible pairs between them. For example, if a user accessed three organisations, we will record the three possible pairs between them using the function `combinations()` from the Python package `itertools`. Note also that the line `for name, group in actor_df:` loops over all groups (having their name as well) contained in the `groupby` object `actor_df` that aggregates over each user.

```

from itertools import combinations

actor_df = df.groupby('actor.login')

dfs = pd.DataFrame(columns=['Actor', 'Org1', 'Org2'])

for name, group in actor_df:
    orgs = group['org.login'].unique()
    orgs = [val for val in orgs if val in toporgs]
    if len(orgs)>1:
        actor_edge = pd.DataFrame(data=list(combinations(orgs, 2)), columns=['Org1', 'Org2'])
        actor_edge['Actor'] = name
        dfs = dfs.append(actor_edge)

```

```

dfs = pd.concat([dfs, actor_edge])
dfs.head()

```

	Actor	Org1	Org2
0	0xc0170	ARMmbed	mbedmicro
0	0xdeadcafe	projectkudu	aspnet
0	130s	ros-planning	ros-industrial-consortium
0	13abylon	arangodb	SleepyDragon
0	1Power	mongodb	rails

Each row in the `dfs` data frame corresponds to a user initiating an event to two different organisations (vertices), in other words it defines an edge to the network. We are almost ready to proceed; the final steps produce a data frame that allows for a more efficient way to construct the network by counting the number of times each edge (pair) occurs, also known as *weight* of the edge, and records it. We also print the top 10 edges in terms of their weight.

```

dfs = dfs.groupby(['Org1', 'Org2']).agg('count').reset_index().rename(columns={'Actor': 'weight'})
dfs = dfs.sort_values(by='weight', ascending=False)
dfs.head()

```

	Org1	Org2	weight
187	odoo-dev	odoo	17
186	odoo	odoo-dev	9
133	iojs	joyent	5
105	google	GoogleCloudPlatform	5
190	openshift	GoogleCloudPlatform	4
10	GoogleCloudPlatform	openshift	4
61	chef	opscode-cookbooks	3
27	OCA	odoo	3
8	GoogleCloudPlatform	docker	3
56	babel	facebook	3

# Network Visualisations

Now we proceed to the part of actually producing the network. First we load the **networkx** package.

```
import networkx as nx
```

We can now create the network by adding edges one by one:

```
G = nx.Graph()

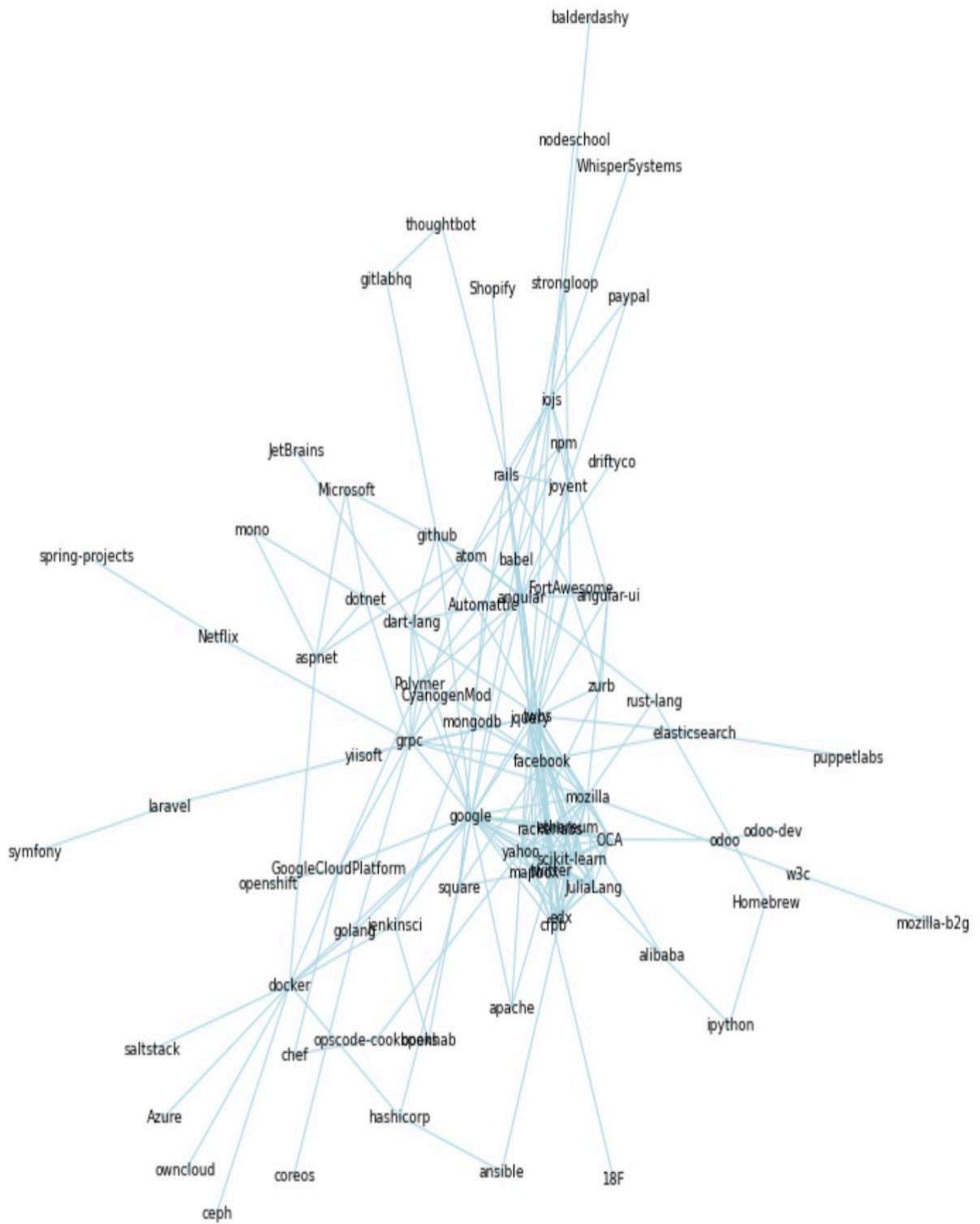
for index, row in dfs.iterrows():
    G.add_edge(row['Org1'], row['Org2'], weight=row['weight'])

# remove isolated vertices (if any)
remove = [node for node,degree in G.degree() if degree ==0]
G.remove_nodes_from(remove)
```

Finally, we can now produce a network with the **networkx** package. We start with the **spring** style after setting some network options regarding its size and colour:

```
#Setting size and colours
options = {
...     'node_color': 'lightblue',
...     'edge_color': 'lightblue',
...     'node_size': 1,
...     'width': 1,
...     'alpha': 1.0,
... }

# Producing the network
plt.subplots(figsize=(10,10))
pos=nx.spring_layout(G)
nx.draw(G,pos=pos,font_size=9,**options)
nx.draw_networkx_labels(G,pos=pos,font_size=9,**options)
plt.tight_layout()
plt.axis('off');
plt.show()
```



```
plt.subplots(figsize=(10,10))

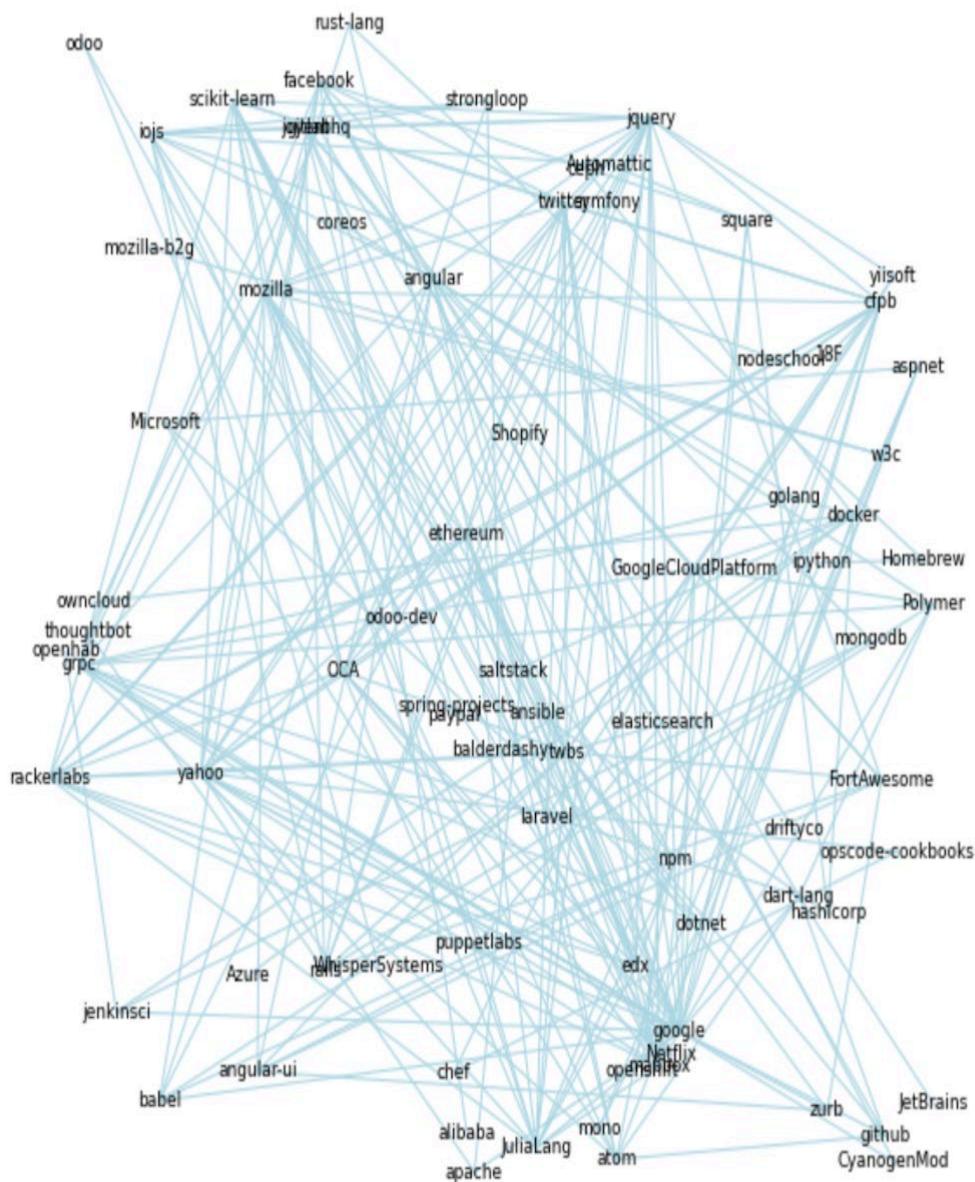
pos=nx.random_layout(G)

nx.draw(G,pos=pos,font_size=9,**options)

nx.draw_networkx_labels(G,pos=pos,font_size=9,**options)

plt.tight_layout()
```

```
plt.axis('off');  
plt.show()
```



```
plt.subplots(figsize=(10, 10))

pos = nx.circular_layout(G)

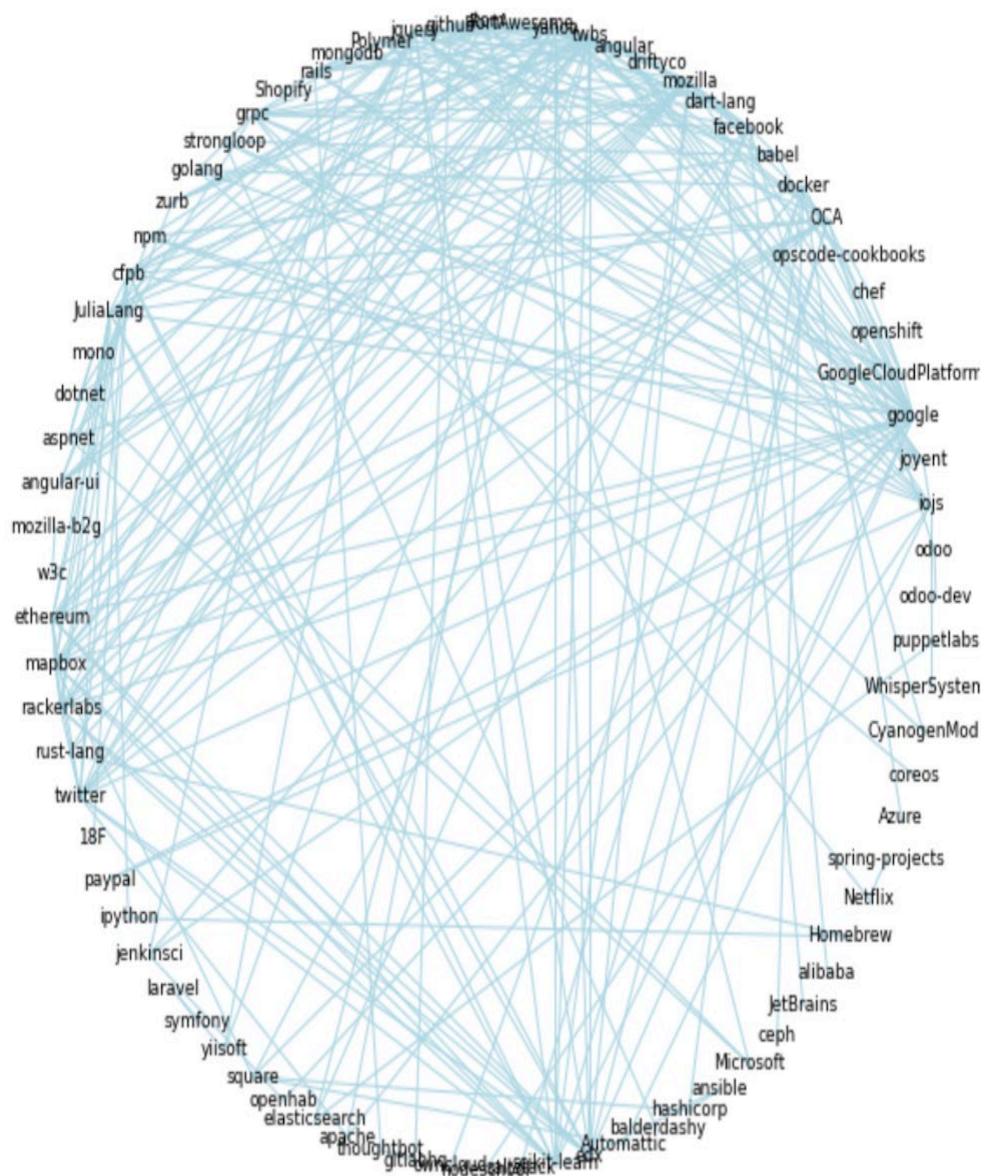
nx.draw(G, pos=pos, font_size=9, **options)

nx.draw_networkx_labels(G, pos=pos, font_size=9)

plt.tight_layout()

plt.axis('off')

plt.show()
```



## More Network Layouts From Graphviz

The package **networkx** has more visualisation options (feel free to explore) and can also be combined with **Graphviz** (graph visualization software), which is a package of open-source tools initiated by AT&T Labs Research for drawing networks specified in DOT language scripts. More information can be found [here](#). To install **Graphviz** follow the instructions [here](#). To incorporate **Graphviz** into Python, the **Pygraphviz** interface to the **Graphviz** package can be used, see [here](#) for more information. To install **Pygraphviz**, type one of the following in the Anaconda prompt or terminal:

```
conda install -c conda-forge python-graphviz
```

or

```
pip install pygraphviz
```

Alternatively the **pydot** interface can be used (more information [here](#)).

```
pip install pydot
```

## Networks in R

The most well-used package for manipulating network data in R is the **igraph** R package, which provides methods to handle and visualize network data, as well as methods for computing key network summaries (e.g. node centrality and connectivity metrics), methods for community detection and basic methods for the visualization of network data. **igraph** and its capabilities are also available in Python through the **python-igraph** module.

In addition to the visualization capabilities **igraph** provides, there is also a range of dedicated R packages for the visualization of networks. Marked examples are the **vizNetwork** R package that can be used to produce interactive network visualizations (see the [vizNetwork pages](#) for examples), the **ggnetwork** that provides **ggplot2** geometries for network data (see [ggnetwork's vignettes](#) for examples), and the **networkD3** R package.

[Katya Ognyanova's blog post](#) provides a lot of resources about visualizing static and dynamic networks in R.

The **sna** R package also provides a range of advanced summaries and data-analytic tools for social network analysis.

## Useful Links and Resources

- [A tutorial on networkx](#)
- [The graphviz project](#)
- [Katya Ognyanova's blog post on visualizing static and dynamic networks in R](#)

## References

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.

# ST2195 Programming for Data Science

## Block 9 Machine Learning Frameworks

### VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:  
<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-9>

### Data Science Workflow

Recall the workflow of a data science project that typically involves the following steps.

1. Importing the data, cleaning them and preparing them for use (data wrangling)
2. Get some basic level of understanding on the data by exploration and visualisation.
3. Consider several models as well as combinations thereof to separate signal from noise.
4. Compare models and inform future decisions.

In this and the next week we will touch upon the last two steps. Note that this essentially constitutes what is also known as *machine learning*, one of the most substantial parts of data science. Here we will briefly mention some key concepts; for more on that consider taking the course **ST3189 Machine Learning**.

### Machine Learning: Definition and Applications

The recent increase in collecting and using data has resulted in widespread interest in data analysis and *machine learning*. There is no unique definition of machine learning; different sources offer different explanations. One version of those defines machine learning as a set of methods that can automatically detect patterns in data and use those patterns to make predictions (Murphy, 2013).

There are numerous real-world applications of machine learning. One of its big successes is image recognition, which is widely used nowadays for operations such as scanning barcodes in a supermarket. Another big area of success is movie and product recommendation systems on platforms such as Netflix or Amazon, internet search engines etc. Other areas are fraud detection in transactions or identifying spam emails. This is by no means an extensive list with machine learning emerging in several areas such as medical imaging, biology and genetics, artificial intelligence and robotics and recently finance.

# Elements of a Machine Learning Model

A *machine learning model*, or *learner*, can be described as an algorithm that inputs data and outputs actions or predictions based on its parameters. The model is designed to automatically produce an output from input data. In order to calibrate a model we need to apply it to some data and set up a *cost function* that measures how far the predictions from the observations are. The overall aim in machine learning is to identify the parameters and the models that minimise this cost function. Nevertheless, this task is more complex than an optimisation problem as we do not necessarily want to optimise the performance on the observed data but on future versions of them.

The parameters of a machine learning model can involve choices made on the data pre-processing steps. For example, the data could be scaled (subtract the sample mean and divide with the standard deviation), and missing values can be imputed with different strategies. Some of the strategies for imputing missing values are listed below:

1. For each missing point impute the mean, median or the mode of each variable based on the observed data.
2. For each missing point impute a randomly sampled value from the observed data.
3. As in 2 above, but rather than sampling with equal probabilities, sample based on a histogram
4. Do not impute and remove the cases with missing values entirely.

From a programming point of view, when data pre-processing steps are merged with machine learning models, we get a new more extended model termed as *machine learning pipeline*.

## Types of Machine Learning Models

Depending on the type of problem, different types of machine learning models can be used to make predictions. The main types are supervised and unsupervised learning, although recently other types of learning have emerged such as reinforcement learning. We give a brief description of those below:

- *Supervised learning*: For cases where there is a single variable that we want to predict, termed as *target* or *response* or *dependent variable*, based on several other ones, termed as *features* or *covariates* or *independent variables*.
- *Unsupervised learning*: For cases when there is no single target or response variable, where we are interested in exploring the variables together with equal importance on all of them. In such cases we may be interested in identifying groups of variables that share something in common, known as *factor analysis*, or groups of individuals that have something in common, known as *clustering*. Analysis of *networks* also falls into this category.

In the next two weeks we will focus on supervised learning by working through several examples. We can further divide supervised learning into two categories depending on the type of the response variable: When the response is a continuous variable the task is termed as *regression*, whereas when it is categorical it is termed as *classification*. There are several machine learning models for those tasks, below is just an indicative list:

- Linear regression (regression only)
- Logistic regression (classification only)
- Lasso/Ridge regression (regression only)
- Penalised logistic regression (classification only)
- Regression and classification trees
- Random forests

- Gradient boosting
- Deep or shallow neural networks
- Gaussian process regression and classification
- Support vector machines (Classification only)

Most of these methods are covered in the course **ST3189: Machine Learning**. Here we will just use them.

The most standard cost function used for Regression task is the *mean squared error (MSE)*. This is obtained by subtracting each prediction from the actual point, taking the square of this difference and averaging all these squares. For the case of classifications see the notes of next week.

## Machine Learning Workflow

### Train, Validation and Test Sets

The procedure in machine learning is determined to a large extent by (usually) randomly splitting the data into two or three sets.

1. *Train set*: Data to be used to train each model. After training the models we can contrast their predictions of the target against the actual observations in this set. This provides the *train error* which is a measure of the *in sample* performance of each model.
2. *Validation set*: The trained models from the previous step can also provide predictions for the validation set based the features of this set. Comparing predictions against actual points for the target provides the *test error*, which is a measure of the *out of sample* performance. We typically prefer the model with the lowest test error.
3. *Test set*: It turns out that the test error calculated on the validation set is good for choosing models but downward biased if the aim is to estimate the resulting test error. This is because the choice of the optimal models is made in sight of the supposedly unseen data of the validation set. This is why we often set aside some entirely unseen data in order to apply our chosen model in the end and obtain a more reliable value for the test error.

The above choice often depends on the sample size; if it is not big enough the validation and test sets are often merged into a single set called test set.

### Cross-Validation

A potential issue with having a single train-validation-test split is that we may end of having a fortunate or unfortunate split for some models due to chance. To limit this possibility, this process can be done repeatedly, in other words we can use *cross-validation*. The repeated partitions can be done in various ways.

One of the most widely used techniques is *K-fold* cross-validation: The data is randomly split into K subsamples. Then, each of these subsamples is set as test set in turn, with the remaining K-1 subsamples being used for training purposes. This process is repeated K times and the results are aggregated, e.g. averaged. See below for a schematic representation.

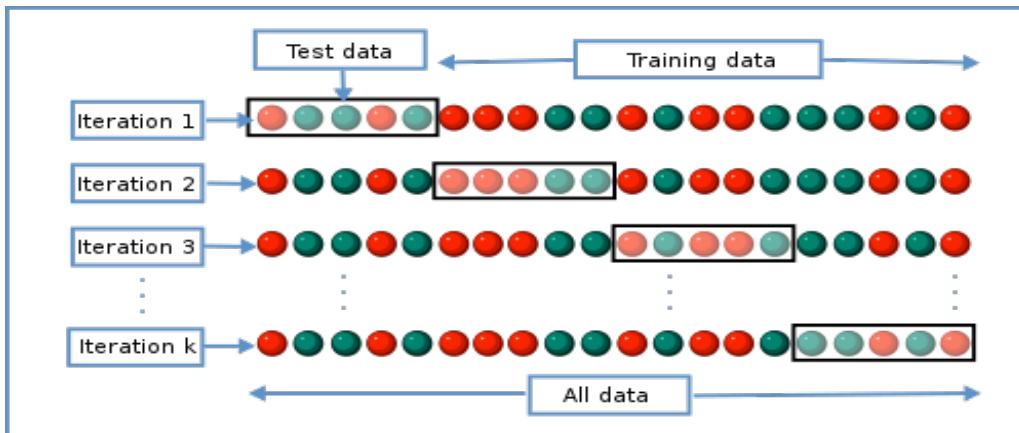


Diagram of K-fold cross-validation. Source: Wikipedia.

Other examples include repeated random subsampling (randomly split in train and test sets many times and aggregate), *leave-one-out* or *Jackknife* (K-fold cross-validation with K=1), etc.

## Machine Learning Steps

To sum up the machine learning framework involves the following steps (resembling the Data Science workflow we outlined earlier):

1. Import and pre-process the data (data wrangling). The decisions made in these steps can be thought of as model parameters.
2. Consider different models/learners, train them and compare them with the validation set.
3. Obtain predictions from the optimal model/learner chosen in the previous steps and check its performance.

## Useful Links and Resources

- - [ISLR book for a gentle introduction to machine learning](#)

## References

Murphy, K. P. (2013). *Machine learning : A probabilistic perspective*. MIT Press.

# Machine Learning Frameworks in R



\*\* Note: The code chunks below should be run in the following order \*\*

## Data

In this page the aim is to illustrate the main concepts of machine learning as well as working with *pipelines*. We will do so using the **mlr3** R package and work on the Boston dataset, which consists of the following 14 variables:

Variable	Description
crim	per capita crime rate by town
zn	proportion of residential land zoned for lots over 25,000 square feet
indus	proportion of non-retail business acres per town.
chas	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
nox	nitric oxides concentration (parts per 10 million)
rm	average number of rooms per dwelling
age	proportion of owner-occupied units built prior to 1940
dis	weighted distances to five Boston employment centres
rad	index of accessibility to radial highways
tax	full-value property-tax rate per \$10,000
ptratio	pupil-teacher ratio by town
black	$1000(Bk - 0.63)^2$ where Bk is the proportion of Black ethnicity residents by town
lstat	% lower status of the population
medv	median value of owner-occupied homes in \$1000's

See Harrison & Rubinfeld (1978) for more information on the aim the data were used initially. Here we will focus on developing competitive machine learning techniques to predict the value of an owner-occupied home in Boston, based on the features (variables) above. This is a supervised learning task.

As a first step, we load and inspect the data from the **MASS** library.

```
library('MASS')
df<-Boston
head(df)

  crim zn indus chas    nox      rm    age      dis rad tax ptratio   black lstat
1 0.00632 18  2.31     0 0.538 6.575 65.2 4.0900     1 296  15.3 396.90  4.98
2 0.02731  0  7.07     0 0.469 6.421 78.9 4.9671     2 242  17.8 396.90  9.14
3 0.02729  0  7.07     0 0.469 7.185 61.1 4.9671     2 242  17.8 392.83  4.03
4 0.03237  0  2.18     0 0.458 6.998 45.8 6.0622     3 222  18.7 394.63  2.94
5 0.06905  0  2.18     0 0.458 7.147 54.2 6.0622     3 222  18.7 396.90  5.33
6 0.02985  0  2.18     0 0.458 6.430 58.7 6.0622     3 222  18.7 394.12  5.21
  medv
```

```

1 24.0
2 21.6
3 34.7
4 33.4
5 36.2
6 28.7

```

To illustrate the use of a machine learning pipeline, we will also change the first value of the `crim` variable into a missing value. The use of pipelines will then be essential. We use the `skimr` package to produce a report on the data; the presence of the missing value is confirmed there.

```

library(skimr)
df$crim[1] <- NA
skim(df)

```

#### Data summary

Name	df
Number of rows	506
Number of columns	14

#### Column type frequency:

numeric	14
---------	----

Group variables	None
-----------------	------

#### Variable type: numeric

skim_variable	n_missin	complete_rate	mean	sd	p0	p25	p50	p75	p100hist
e	g	e							
crim	1	1	3.62	8.61	0.01	0.08	0.26	3.68	88.98
zn	0	1	11.36	23.32	0.00	0.00	0.00	12.50	100.0
indus	0	1	11.14	6.86	0.46	5.19	9.69	18.10	27.74
chas	0	1	0.07	0.25	0.00	0.00	0.00	0.00	1.00
nox	0	1	0.55	0.12	0.38	0.45	0.54	0.62	0.87
rm	0	1	6.28	0.70	3.56	5.89	6.21	6.62	8.78
age	0	1	68.57	28.15	2.90	45.02	77.50	94.07	100.0
dis	0	1	3.80	2.11	1.13	2.10	3.21	5.19	12.13
rad	0	1	9.55	8.71	1.00	4.00	5.00	24.00	24.00
tax	0	1	408.2	168.5	187.0	279.0	330.0	666.0	711.0
ptratio	0	1	18.46	2.16	12.60	17.40	19.05	20.20	22.00
black	0	1	356.6	91.29	0.32	375.3	391.4	396.2	396.9
lstat	0	1	12.65	7.14	1.73	6.95	11.36	16.96	37.97
medv	0	1	22.53	9.20	5.00	17.02	21.20	25.00	50.00

# Machine Learning Pipelines in mlr3

The **mlr3** provides a general platform to train and evaluate a wide range of machine learning models. It also allows to set up machine learning pipelines in a very convenient manner. Below we will demonstrate the key steps for doing so.

## Training and Evaluating a (Machine) Learner

The first step is to set up a *task* such as doing regression or classification on a particular dataset and specifying the response variable.

```
library('mlr3')
task <- TaskRegr$new('boston', backend=df, target = 'medv')
measure <- msr('regr.mse')
```

Next, a learner needs to be chosen. We start with linear regression `regr.lm` which is essentially the `lm()` function in R.

```
library('mlr3learners')
learner_lm <- lrn('regr.lm')
```

If we were to try and fit `lm()` to the data, the cases with missing values will be removed entirely which may not be the best way to proceed. Other learners may even return an error. So it is essential to either remove the cases with missing values or perform some imputation. Below we go with `imputemean`, i.e. imputing the mean of the variable's other values and include this as part of the learner. Other learners can be constructed based on other options and compared to each other, for example:

- `imputemedian` to impute the median of variable's other values
- `imputesample` to impute a sample from variable's other values
- `imputehist` to impute a sample from the histogram of variable's other values

In order to join together the operations of imputing missing values and fitting the linear regression model, **mlr3** creates a pipe for each of these tasks and connects them with the operator `%>>%` from **mlr3pipelines**. The resulting pipeline is termed as a *graph*, denoted by `gr_lm` in the code below, and can be converted in a learner by using `GraphLearner.new()`:

```
library('mlr3pipelines')
gr_lm <- po('imputemean') %>>%
  po(learner_lm)
glrn_lm <- GraphLearner$new(gr_lm)
```

Now we can view the combined learner `glrn_lm` as a separate machine learning algorithm. To evaluate its performance we can proceed by

1. Training it in the training data,
2. Extracting its predictions for the test data
3. Evaluating its performance by contrasting the predictions against the actual data via the model evaluation metric (`mse` in our case)

This can all be done with the code below that returns the value of the `mse`.

```

set.seed(1)

train_set <- sample(task$nrow, 0.7 * task$nrow)

test_set <- setdiff(seq_len(task$nrow), train_set)

glrn_lm$train(task, row_ids = train_set)

glrn_lm$predict(task, row_ids = test_set)$score()

regr.mse

27.2937

```

One of the advantages of working with pipelines, as above, is that they are quite convenient for guarding against issues such as *data leaking*. In other words it ensures that the imputation is done based entirely on information contained on the training dataset; this would not be the case if we were to impute the value in the beginning on the basis of the entire dataset.

## Ridge Regression: Tuning Hyperparameters

In the previous example there were no hyperparameters but this is rather the exception than the rule when dealing with machine learning models. Let us consider another example, namely *ridge regression*. The learner `regr.glmnet` is now used, which essentially calls the `glmnet` package under the assumption that is installed. In other words, there is no need to include `library(glmnet)` but if `glmnet` is not installed, an error will be returned. Setting `alpha = 0` does ridge regression whereas `alpha = 1` corresponds to *lasso regression*. If you are curious about ridge and lasso regression you can check the relevant section in James et al. (2014), although this is beyond the scope of this course. We will proceed with `alpha = 0` but feel free to experiment with `alpha = 1`. In both cases, a value for `lambda` needs to be provided and we will use `lambda = 0.03` below for illustration purposes.

As before we incorporate missing values imputation in the procedure. We also add the operation of scaling the features (subtracting their sample mean and dividing with their standard deviation) which is generally recommended in ridge regression. Again, this is something that can cause data leaking issues if it was to be done on the entire dataset but if done in the following way it will only use information from the training data.

```

learner_ridge <- lrn('regr.glmnet')

learner_ridge$param_set$values <- list(alpha = 0, lambda = 0.03)

gr_ridge <- po('scale') %>>%
  po('imputemean') %>>%
  po(learner_ridge)

glnr_ridge<- GraphLearner$new(gr_ridge)

glnr_ridge$train(task, row_ids = train_set)

glnr_ridge$predict(task, row_ids = test_set)$score()

regr.mse

27.27267

```

The above give a slightly lower `mse` than before. But it relied on the artificial value `lambda = 0.03` which may not represent the best option. To determine its optimal value we can try several different values and choose the optimal via cross-validation. This process is known as *tuning hyperparameters*.

First we need to create a slightly different version of the previous learner. More specifically the values of `lambda` will not be fixed beforehand but it will be a *free* parameter to be optimised. As before, we create a pipeline by incorporating imputation and scaling. This gives another learner, termed `glnr_ridge2`.

```
learner_ridge2 <- lrn('regr.glmnet')
learner_ridge2$param_set$values <- list(alpha = 0)
gr_ridge2 <- po('scale') %>>%
  po('imputemean') %>>%
  po(learner_ridge2)
glnr_ridge2 <- GraphLearner$new(gr_ridge2)
```

The next steps are the following:

- Specify the hyperparameters and the range of their values to be explored
- Determine how the search will be done, for example grid search
- Determine how many times (evaluations) the procedure will be repeated

The above can be done (in that order) with the following code. In the end we combine everything into a new pipeline `at_ridge` which is trained on the training data.

```
library('mlr3tuning')
library('paradox')
# Set up tuning environment
tune_lambda <- ParamSet$new(list(
  ParamDbl$new('regr.glmnet.lambda', lower = 0.001, upper = 2)
))
tuner<-tnr('grid_search')
terminator <- trm('evals', n_evals = 20)
#Put everything together in a new learner
at_ridge <- AutoTuner$new(
  learner = glnr_ridge2,
  resampling = rsmp('cv', folds = 3),
  measure = measure,
  search_space = tune_lambda,
  terminator = terminator,
  tuner = tuner
)
#Train the learner on the training data
at_ridge$train(task, row_ids = train_set)
```

After training the ridge machine learning pipeline, the next step is to evaluate its performance. Below we just report it `mse`. Further inspection of the training, for example finding out the optimal value of `lambda` can be done with `at_ridge$model`.

```

at_ridge$predict(task, row_ids = test_set)$score()

regr.mse
27.18366

```

The result is a slightly lower `mse` which is not convincing. A similar learner is offered by *lasso regression* and can be fit using the previous code and simply setting `alpha = 1`; it is recommended to do it as an exercise.

## Random Forests

So, next, we move away from a linear regression setting and try *random forests*, a very successful technique in [Kaggle](#) competitions. Random forests are known to do very well in finding non-linear associations as well as synergies between the features. For (entirely optional) reading on random forests, the James et al. (2014) textbook is a good place to start. To specify a random forest pipeline the code is almost identical and only differs in the learner (`regr.ranger`) and its parameters. Again, the code below assumes that the R package `ranger` is installed in your computer.

```

learner_rf <- lrn('regr.ranger')

learner_rf$param_set$values <- list(min.node.size = 4)

gr_rf <- po('scale') %>>%
  po('imputemean') %>>%
  po(learner_rf)

glrn_rf <- GraphLearner$new(gr_rf)

tune_ntrees <- ParamSet$new(list(
  ParamInt$new('regr.ranger.num.trees', lower = 50, upper = 600)
))

at_rf <- AutoTuner$new(
  learner = glrn_rf,
  resampling = rsmp('cv', folds = 3),
  measure = measure,
  search_space = tune_ntrees,
  terminator = terminator,
  tuner = tuner
)

at_rf$train(task, row_ids = train_set)

```

To get the prediction we use the same code as before:

```

at_rf$predict(task, row_ids = test_set)$score()

regr.mse
15.12726

```

We see that the improvement offered by random forests is substantial in this case.

# Benchmarking

An alternative and easier-to-code way to compare the different learners is by the `benchmark()` function which automates the train/predict steps. It requires to setup a benchmark design as an input which can be set by the function `benchmark_grid()`. The latter function tabulates the different comparisons to be made that consist of combinations of tasks, different ways to resample and different learners. Below, we stick to regression task of the Boston data, choose three-fold cross-validation and the four learners we checked so far (the code below may take a while to run):

```
set.seed(123) # for reproducible results

# list of learners

lrn_list <- list(
  glrn_lm,
  glrn_ridge,
  at_ridge,
  at_rf
)

# set the benchmark design and run the comparisons

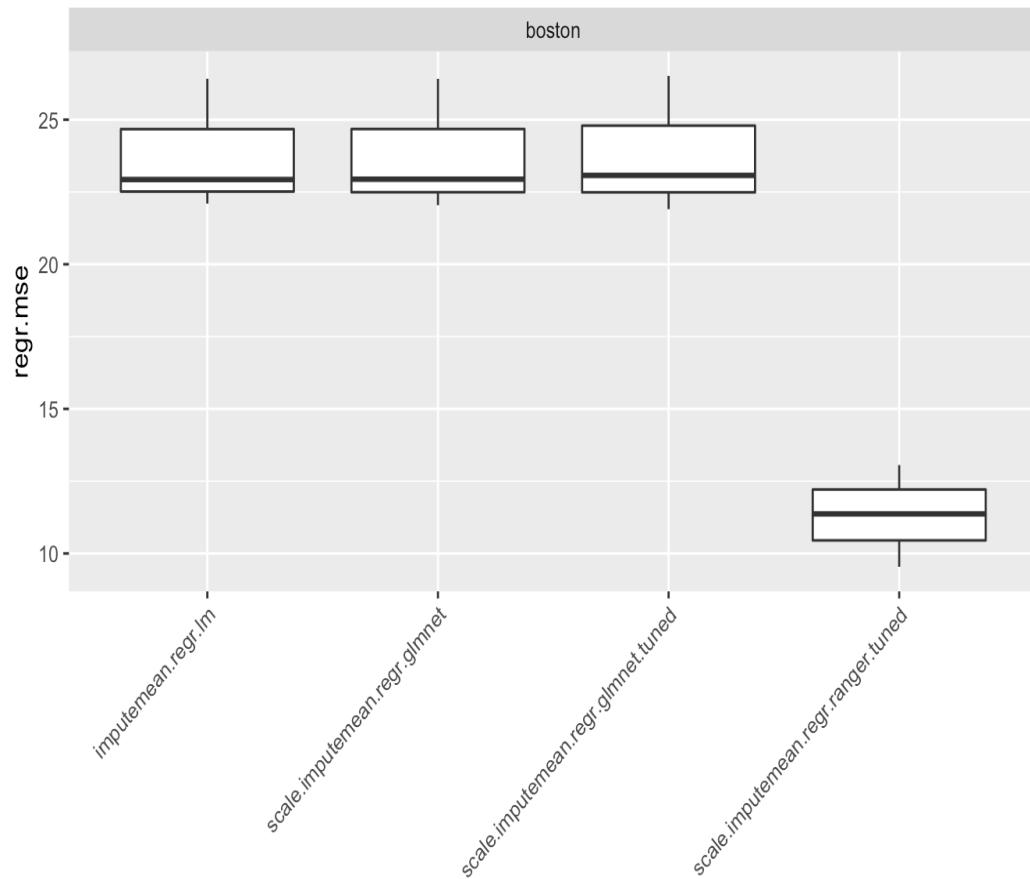
bm_design <- benchmark_grid(task = task, resamplings = rsmp('cv', folds = 3), learners = lrn_list)

bmr <- benchmark(bm_design, store_models = TRUE)
```

A nice way to visualise the comparisons made is with boxplots. We can also see the overall `mse` for each learner using `bmr$aggregate(measure)`.

```
library('mlr3viz')
library('ggplot2')

autoplot(bmr) + theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```
bmr$aggregate(measure)
  nr      resample_result task_id
1: 1 <ResampleResult[21]> boston
  learner_id
                    imputemean.regr.lm
2: 2 <ResampleResult[21]> boston
                    scale.imputemean.regr.glmnet
3: 3 <ResampleResult[21]> boston scale.imputemean.regr.glmnet.tuned
4: 4 <ResampleResult[21]> boston scale.imputemean.regr.ranger.tuned
  resampling_id iters regr.mse
1:           cv     3 23.81437
2:           cv     3 23.79866
3:           cv     3 23.83034
4:           cv     3 11.32159
```

The results above confirm the clear dominance of random forests over regression methods in the Boston dataset.

## Parallelisation

Training pipelines may end up taking a lot of time. One way to reduce this is by splitting the procedure into multiple jobs and execute them in parallel, simultaneously. This procedure is known as *parallelisation* and allows for significant gains in terms of computing power and substantial reduction in the execution times.

The package `mlr3` is compatible with the `future` package for parallelisation, thus making it very easy to parallelise. All you have to do is essentially install the package `future` and simply run the following line in the beginning of your code

```
future::plan()
```

## Useful Links and Resources

- [Book for the `mlr3` package](#)
- [Function Reference for the `mlr3` package](#)
- [Project `mlr` with several resources](#)
- [The `caret` package; an alternative to `mlr3`](#)

## References

Harrison, D., & Rubinfeld, D. L. (1978). Hedonic prices and the demand for clean air. *Journal Environmental Economics and Management*, 5, 81–102.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2014). *An introduction to statistical learning: With applications in R*. Springer.

# Categorical Variables in Machine Learning

As already mentioned categorical variables take values in a predetermined finite set of categories. An example of a categorical variable is whether an email is spam or not. Below we will briefly discuss how to handle such variables in machine learning (mostly supervised learning) depending on whether they are forming the target variable or feature variables.

## Categorical Predictions

In order to illustrate the concepts let us consider binary classification with the target being  $\{Y\}$  taking values of 1: a person will default on its credit card or 0: no default. Typically machine learning models, also called *classifiers* in this case, provide probabilities of  $\{Y=1\}$  for each individual given their set of features. Often a threshold is selected for these probabilities, above which we classify the case as 1 otherwise it is classified as 0. Without loss of generality, let us assume that this threshold is 0.5. In our example, the individuals with probability higher than 0.5 are classified as projected to default, otherwise they are considered safe. In the presence of the data we can then check the actual outcome, i.e. whether they did default or not. When contrasting projected classifications and outcomes, there are four different cases:

- *True positives*: Projected to default and defaulted
- *False positives*: Projected to default but did not default
- *True negatives*: Did not project to default and did not default
- *False negatives*: Did not project to default but defaulted

The rate of *true positives*, correctly identified is also known as *sensitivity* whereas the rate of *true negatives* correctly identified is also known as *specificity*. These quantities can be summarised in the so called *confusion matrix*. Below we see an example of a confusion matrix, taken from James et al. (2014), that corresponds to the threshold of 0.5:

		<i>True default status</i>		Total
		No	Yes	
<i>Predicted default status</i>	No	9,644	252	9,896
	Yes	23	81	104
Total	9,667	333	10,000	

Confusion matrix of the previous example based on a probability threshold of 0.5; Table 4.4 in James et al. (2014).

Based on the above matrix we see that overall classification accuracy is  $(9644+81)/10000$ , or else 97.25%. This may seem high but if we focus on the 333 individuals who defaulted only 81 were identified. In other words the sensitivity is  $81/333=0.24$ , which suggests that 76% were missed. On the other hand the model is very successful in classifying the negatives, i.e. those less likely to default, as the specificity is  $9644/9667=0.99$ . Note however that the confusion matrix depends on the choice of threshold whose choice may also be determined by the context. In this case the aim may be to identify individuals that are likely to default and take up an action to prevent it. In that sense we may be willing to tolerate some false negative, in other words

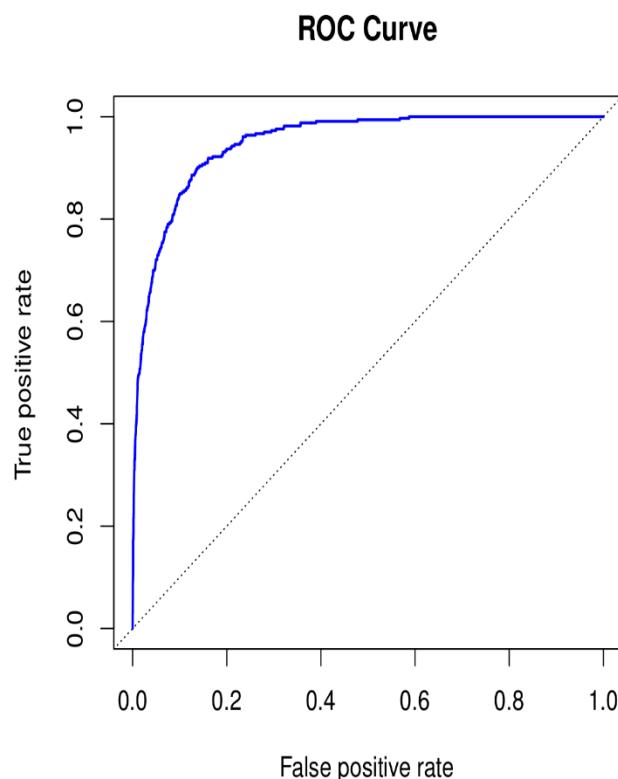
triggering false alarms indicating people are in danger when they are not, if this results in higher chances of identifying people in danger. This can be calibrated by choosing another threshold. Below is the confusion matrix from the same example for a threshold of 0.2:

		True default status		Total
		No	Yes	
Predicted default status	No	9,432	138	9,570
	Yes	235	195	430
Total		9,667	333	10,000

Confusion matrix of the previous example based on a probability threshold of 0.2; Table 4.5 in James et al. (2014).

Now we get a sensitivity of  $195/333=0.59$  and a specificity of  $9432/9667=0.97$ , which is better for the purposes of this classification task.

In other cases however setting a fixed threshold is not so straightforward and we are interested in getting a measure that does not depend in its choice. This is provided by the *receiver operating characteristics (ROC) curve* which is constructed by taking a range of thresholds and plotting the sensitivity (true positive rate) versus one minus the specificity (false positive rate). It turns out that the *area under the ROC curve*, known as AUC, corresponds to the probability of correctly classifying a positive and a negative case. In our example the ROC curve is shown below and corresponds to an AUC of 0.95.



ROC curve for the previous example with AUC of 0.95. Figure 4.8 in James et al. (2014)

In the ROC plot above the ideal ROC curve would come as close as possible to the top left corner, which would correspond to high true positive rate (high sensitivity) and low false positive rate (low specificity). The dotted line corresponds to a classifier that is essentially guessing at random and classifies correctly with 50% chance.

## Categorical Features

There are two ways to handle categorical variables as features (inputs) in regression or classification tasks. The first way is to assign a numerical score on their values and treat them as continuous. This is clearly suboptimal from an interpretation viewpoint as the assignment of numerical scores is arbitrary. In the case of nominal variables (e.g. 1:red, 2:yellow, 3:green) there is no reason why one of these colours should have bigger values than the others. But even in ordinal variables (e.g. 1:low, 2:medium, 3:high) there is no guarantee why the distance between medium and low is the same as that of between high and medium. So we may as well have used the numbers (1, 3, 7) or any other ordered triplet of real numbers. Despite this, the use of numerical scores (quantification) is still being used as it is convenient and result in fewer features in the model.

A more meaningful option is to create *dummy variable(s)* from each categorical variable. Dummy variables take values of either 0 or 1, indicating the presence (1) or absence (0) of a category. Typically the number of dummy variables for each categorical variable is equal to the number of categories (say  $k$ ) minus 1; one category is treated as the reference category. Let us consider the (1:red, 2:yellow, 3:green) example, where we can set the green category as reference and create two dummy variables: one taking value 1 if the colour is red and 0 otherwise, and another one taking the value 1 if the colour is yellow and 0 otherwise. The reason for creating  $k-1$  dummy variables, rather than  $k$ , is that the  $k$ -th dummy variable is redundant in the presence of the others. Still on the (1:red, 2:yellow, 3:green) example, if we know whether the colour is red (or not) and whether it is yellow (or not), then we can also infer if it is green or not. In other words, if the dummy variable for red is 1 or if the dummy variable for yellow is 1, the colour is not green. But if both of the dummy variables for red and yellow are 0, then the colour is red.

Now consider a regression model where the target is  $\backslash(y)$  and the dummy variables for red and yellow are the features. The coefficient of the red dummy variable will then reflect the mean difference (in the values of  $\backslash(y)$ ) between cases corresponding to red and green colours. Similarly the coefficient of the yellow dummy variable will reflect the mean difference (in the values of  $\backslash(y)$ ) between cases corresponding to yellow and green colours. The constant in the regression equation will reflect the mean value of  $\backslash(y)$  when both dummy variables are zero or, in other words, when the colour is green.

Hence, we can setup a regression model with a categorical feature in that way and the regression coefficients will reflect the mean values of  $\backslash(y)$  in each of the categories. All we need to do is construct these dummy variables, use them as inputs, and then proceed as in the case of continuous inputs. Note also that the choice of the reference category only affects the interpretation of the regression coefficients. In terms of predictive performance, models with dummy variables extracted on the basis of different reference categories will be equivalent.

## Useful Links and Resources

- [ISLR book for a gentle introduction to machine learning](#)

## References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2014). *An introduction to statistical learning: With applications in R*. Springer.

# ST2195 Programming for Data Science

## Machine Learning Frameworks in Python



\*\* Note: The code chunks below should be run in the following order \*\*

### Python Setup

We start by activating the necessary Python packages. In addition to the standard use of `pandas` and `matplotlib`, we will be using several features from the `scikit-learn` or `sklearn` package which is the default option for machine learning in Python. The `scikit-learn` package is an open-source collection of simple and efficient tools for machine learning, built on `numpy`, `scipy`, and `matplotlib`. More details about it can be found [here](#).

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml #using openml to import data
from sklearn.metrics import plot_roc_curve
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer #transform different types
```

# Data

We will be working on the *titanic* dataset that is available from the [OpenML](#) website. The features and response are stored in separate arrays, `x_initial`, and `y` respectively. To view all the data we combine them into a single pandas data frame called `combined_dataset`.

```
# Dataset details at-
X_initial, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)
combine_dataset = pd.concat([X_initial, y], axis=1)
combine_dataset.head()
```

	pclass	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest	survived
0	1.0	Allen, Miss. Elisabeth Walton	female	29.0000	0.0	0.0	24160	211.3375	B5	S	2	NaN	St Louis, MO	1
1	1.0	Allison, Master. Hudson Trevor	male	0.9167	1.0	2.0	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON	1
2	1.0	Allison, Miss. Helen Loraine	female	2.0000	1.0	2.0	113781	151.5500	C22 C26	S	None	NaN	Montreal, PQ / Chesterville, ON	0
3	1.0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1.0	2.0	113781	151.5500	C22 C26	S	None	135.0	Montreal, PQ / Chesterville, ON	0
4	1.0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1.0	2.0	113781	151.5500	C22 C26	S	None	NaN	Montreal, PQ / Chesterville, ON	0

## Titanic pandas data frame

To illustrate concepts we will focus on five features: age, fare, embarked, sex and class. The matrix of these five features to be used in the following examples will be labelled as `X`.

```
# features from the dataset
features = ['age', 'fare', 'embarked', 'sex', 'pclass']
X = X_initial[features].copy()
```

# Pipelines: Pre-Processing Stage

We will now start to build the pipelines. To do so we will use different pre-processing steps for continuous (numerical) and categorical features. For the numerical features (age and fare) we apply imputation on the missing values with the function `SimpleImputer()` (imputing either the mean or the median of the observations of each variable) and scaling with the function `StandardScaler()`. These operations are bundled together in a pipeline called `numerical_transformer`. The function `Pipeline()` is being used where we simply define the different steps. Each step is specified within brackets, where the name of the step is specified (e.g. `name of step`) followed by the Python function used:

```
numerical_features = ['age', 'fare']

# Applying SimpleImputer and StandardScaler into a pipeline
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer()),
```

```
('scaler', StandardScaler())])
```

For categorical features, imputation is applied as before and we also use the `OneHotEncoder()` function to create the appropriate dummy variables (see notes on categorical variables in this week's material). The resulting pipeline is called `categorical_transformer`.

```
categorical_features = ['embarked', 'sex', 'pclass']

# Applying SimpleImputer and then OneHotEncoder into another pipeline
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer()),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

Having pre-processed the numerical and categorical features separately we can now merge them with the function `ColumnTransformer()`, in which we specify the two pipelines to be merged using the argument `transformers`. This creates a new pipeline called `data_transformer`.

```
data_transformer = ColumnTransformer(
    transformers=[
        ('numerical', numerical_transformer, numerical_features),
        ('categorical', categorical_transformer, categorical_features)])
```

The `data_transformer` pipeline now contains all the pre-processing steps. It can be used as the starting point for different (machine) learners as we illustrate in the next sections.

## Pipelines With Logistic Regression

We are now ready to complete our first machine learning pipeline by simply attaching a learner to the `data_transformer` pipeline from the previous section. In this section we use logistic regression since the response (`y`: survived) is binary.

```
pipe_lr = Pipeline(steps=[('data_transformer', data_transformer),
                           ('pipe_lr', LogisticRegression(max_iter=10000))])
```

The next stage is to perform the standard machine learning operations of training the pipeline and evaluating its predictive performance. We begin by splitting the data into train and test datasets with the very convenient function `train_test_split()`.

```
X_train, X_test, y_train, y_test = train_test_split(X_initial, y, test_size=0.5, random_state=1)
```

To tune the pipeline we can use cross-validation for each combination of its hyperparameters. In other words we can use *grid search cross-validation* that can be done with the function `GridSearchCV()`. This requires setting up a parameter grid, which is a dictionary containing the names of each hyperparameter followed by its range of values. The hyperparameters to be tuned below are the imputation strategies used in `SimpleImputer()`. In the case of numerical variables there are two options, mean or median. For categorical variables one option is to impute the most frequent value of each variable (`most_frequent`), whereas another option is to impute the

missing value to be the reference category (`constant` with filled value being left to the default of 0). The best possible combination is chosen based on fitting the models on the training data.

```
param_grid = {
    'data_transformer_numerical_imputer_strategy': ['mean', 'median'],
    'data_transformer_categorical_imputer_strategy': ['constant', 'most_frequent']
}
grid_lr = GridSearchCV(pipe_lr, param_grid=param_grid)
grid_lr.fit(X_train, y_train);
```

## Pipelines With Gradient Boosting

Before we evaluate the logistic regression pipeline, let us consider also the Gradient Boosting learner. For simplicity we do not explore tuning the hyperparameters of this learner and simply use the default values for them. Hence, the code is very similar as before, we do add the argument `random_state=2` to ensure reproducible results.

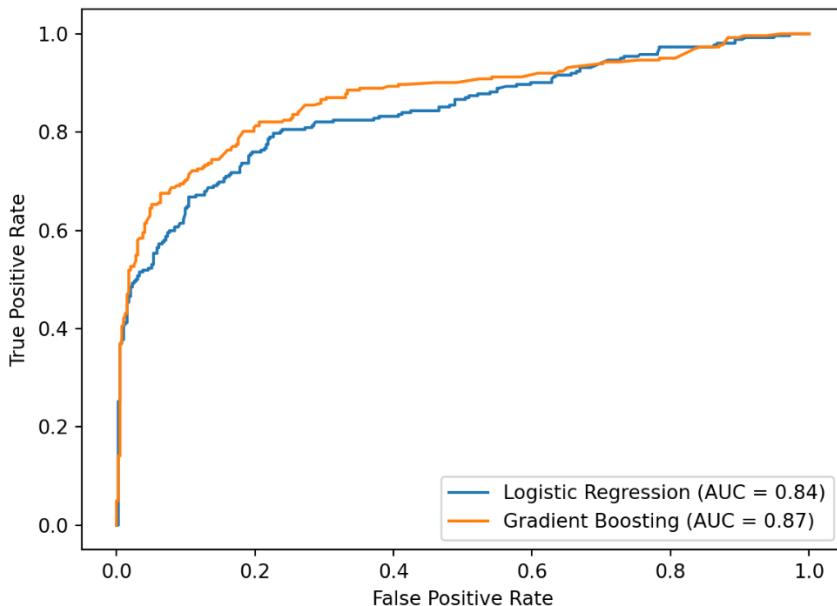
```
pipe_gdb = Pipeline(steps=[('data_transformer', data_transformer),
                           ('pipe_gdb', GradientBoostingClassifier(random_state=2))])

grid_gdb = GridSearchCV(pipe_gdb, param_grid=param_grid)
grid_gdb.fit(X_train, y_train);
```

Finally let us compare the two machine learning pipelines by overlaying their receiver operating characteristic (ROC) curves.

```
ax = plt.gca()

plot_roc_curve(grid_lr, X_test, y_test, ax=ax, name='Logistic Regression')
<sklearn.metrics._plot.roc_curve.RocCurveDisplay object at 0x14a0606d8>
plot_roc_curve(grid_gdb, X_test, y_test, ax=ax, name='Gradient Boosting')
<sklearn.metrics._plot.roc_curve.RocCurveDisplay object at 0x14a064198>
plt.show()
```



The Gradient Boosting offers a small improvement which can potentially be improved by tuning its hyperparameters on the training dataset. This is left as an exercise.

## Parallelisation

As in R, it is straightforward to use parallel computing in order to speed up the training time. The function `GridSearchCV()` has the argument `njobs`, which if set to any value other than 1 results in parallel computing. Setting `njobs=-1` results in using all the available cores. An alternative to `GridSearchCV()`, which can also be parallelised in the same way, is `RandomizedSearchCV()` that searches randomly over the possible combinations of hyperparameter values.

## Useful Links and Resources

- [OpenML](#): Website with several datasets and useful resources for machine learning
- [The scikit learn project](#)
- [More details on the Pipeline function of the sklearn package](#)

# **ST2195 Programming for Data Science**

## **Block 10 Introduction to Software Development**

### **VLE Resources**

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here:

<https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-10>

### **Software Development Phases**

It may sound surprising that the majority of time and effort when developing software for solving a problem in business and research environments is not spent in writing code! In fact, principled software development involves multiple phases, most of which require little or no coding! Software development involves at least elements of each of the following phases:

- Requirements analysis and planning
- Software design
- Implementation
- Testing
- Integration and deployment
- Maintenance

The majority of the code is written during the implementation phase.

How the above phases are combined and in what order, and how much time or effort is invested in each phase is guided by the [software development life cycle](#) (SDLC) methodology that each software development team or organization chooses to adopt.

### **Software Development Life Cycle Methodologies**

There is a range of SLDC methodologies (or models), each with its own advantages and disadvantages depending on the project at hand. The most prominent SLDCs at the time of writing are:

- *Waterfall SDLC model*: The phases are organized in a rigid way in the order given above with strong investment of time and effort in each phase and with no option to revisit past phases. Most software companies nowadays require more flexibility than

what the waterfall model provides, but it still remains a strong methodology for mission-critical projects, where mistakes in the solution can result in extreme costs (e.g. piloting software, software used in nuclear power stations, etc.)

- *Iterative SDLC model:* The key idea of this model is to develop a system through repeated iterations or the waterfall model, where at each iteration (also known as a “mini waterfall”) a sub-problem is solved. Despite the fact that the phases are again organized in a rigid way, developers can take advantage of what was learned from the iterations of earlier parts.
- *Agile SDLC model:* The agile model is in a sense the opposite of the waterfall model. Instead of organizing the phases in a rigid way, the agile model makes all phases an ongoing process that requires the involvement of developers, management and customers. Work is typically organized in two- to four-week segments (known as “sprints”), in which the responsible teams tackle the major needs of their customers and perform testing, integration and deployment on the go. The agile model is popular in start-ups where speed and flexibility (e.g. because of changes in requirements) are essential.
- *V-shaped SDLC model:* This is an evolution of the waterfall model, where each development phase before implementation is associated with a testing phase (e.g. requirement analysis and planning—acceptance testing, software design—software testing). The next phase starts only after there is a testing activity corresponding to it.
- *DevOps SDLC models:* One of the most recent SDLC models, where *development* and *operational* teams are merged together into a single team. Software developers and engineers work across the entire SDLC from development to testing, to deployment and operations, including interaction with customers.
- *Spiral SDLC model:* The spiral model is a blend of iterative and waterfall approaches, and allows teams to adopt multiple SDLC models based on the risk patterns of the project.

The [SDLC tutorial at tutorialspoint.com](#) provides an accessible overview of SDLC methodologies, and of their advantages and disadvantages. [Useful Links and Resources](#) below provide links with more details on each SDLC.

## Useful Links and Resources

- [tutorialspoint pages for the waterfall model](#): Details on the waterfall model
- [tutorialspoint pages for the iterative model](#)
- [tutorialspoint pages for the agile model](#)
- [tutorialspoint pages for the V-shaped model](#): Details on a version of the V-shaped SDLC model
- [Wikipedia's DevOps page](#)
- [tutorialspoint pages for the spiral model](#): Details on the spiral SDLC model

# Developing R packages



## Packages

Python and R packages play a key role in data science projects, not only because they are routinely being used as dependencies or tools during at least some of the software development phases but also because they are often the key outputs and deliverables of a project or part of those.

In fact, one of the reasons that Python and R are so pervasive in data science is because they benefit from diverse and active ecosystems of packages that, as we have seen in this course, provide ready methods for read and write operations, data wrangling and visualization, and for developing machine learning pipelines. As we will also see next week, there are even dedicated packages for testing software.

A package bundles together code, data, documentation, and potentially tests. This makes it easy to share with others, either by submitting to one of the official repositories (e.g. [CRAN](#) for R and [PyPi](#) for Python) or by hosting it in a repository hosting service like GitHub.

Even if you do not want to share your code, organizing it in a package always pays off. Python and R packages follow strict rules on how code is organized, documented and tested, and as a result, packaged code is easier to return to than plain text files, even after several years.

In the following sections you will learn the basic structure of R and Python packages.

## Developing R Packages

### Package Skeleton

Suppose that you wrote a function `circle()` that takes as input the radius of a circle and returns a list of class "circle" with the radius, and a function `area.circle()` that takes as input objects of class "circle" and calculates the area of the circle

```
circle <- function(radius) {  
  shape <- list(radius = radius)  
  class(shape) <- "circle"  
  shape  
}
```

```
#' @export
area.circle <- function(object) {
  pi * object$radius^2
}
```

You may notice the commented-out line `#' @export` before `area.circle()`. Ignore it for now; this is special terminology, whose importance will become apparent later in this page.

You immediately realize that there is potential for an R package that provides functions and methods to compute the area of circles and other 2D shapes, called **areacalc**. How can you go about it?

The first step in writing an R package is to set up the basic *skeleton* for the package directory. The conventions that should be used for a valid R package are detailed in [Section 1.1 of the “Writing R extensions” resource](#). While you can always go ahead and create the required directories and files manually, many of those tasks are nowadays abstracted away through high-level, specialized utilities such as the ones provided by the **usethis** R package. Assuming that you have already installed **usethis**, the following code chunk sets up the skeleton for **areacalc** under the directory `"~/Repositories/"` (feel free to replace this with any *existing* directory in your system):

```
library("usethis")
create_package("~/Repositories/areacalc/")
✓ Creating '~/Repositories/areacalc/'
✓ Setting active project to '~/Repositories/areacalc'
✓ Creating 'R/'
✓ Writing 'DESCRIPTION'
Package: areacalc
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
  * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
         pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
✓ Writing 'NAMESPACE'
✓ Changing working directory to '~/Repositories/areacalc/'
```

The output above details the steps that `create_package()` took while setting up the package skeleton for the **areacalc** R package. Now, the directory `~/Repositories/areacalc` looks like (using the `dir_tree()` function of **fs** R package)

```
library("fs")
dir_tree("~/Repositories/areacalc")
~/Repositories/areacalc
├── DESCRIPTION
└── NAMESPACE
└── R
```

This is the most basic structure for an R package. There is `DESCRIPTION` file, whose contents were automatically generated by `create_package()` earlier (see the output from `create_packages()` above), a `NAMESPACE` file which defines the package name space, and a directory called `R`, which will host the R code we will put in the package.

## Adding New Functionality

A convenient way to add new functionality in the R package is by using `usethis`'s `use_r()` function. For example, the following code chunk will create a empty script called `cicle.R` under `~/Repositories/areacalc/R/` and open it for editing

```
use_r("circle")
✓ Setting active project to ' ~/Repositories/areacalc'
• Modify 'R/circle.R'
• Call `use_test()` to create a matching test file
```

Go ahead and copy and paste in that script the code chunk where the functions `circle()` and `area.circle()` are defined above and hit save. Feel free to ignore the last piece of advice in the output about calling `use_test()` for now; we will come back to this in the following week.

Congratulations! You have just created your first bare-bones R package! You can use the `load_all()` function of the [devtools](#) to load your new functions

```
library("devtools")
Loading required package: usethis
load_all("~/Repositories/areacalc")
Loading areacalc
ci <- circle(1)
area.circle(ci)
[1] 3.141593
```

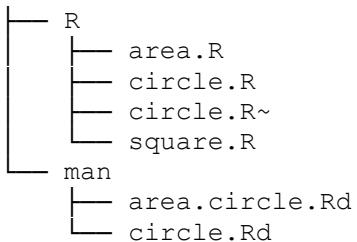
Now, as an exercise, include a new script called `square.R` in `areacalc` with the following two functions to compute the area of a square

```
square <- function(side) {
  shape <- list(side = side)
  class(shape) <- "square"
  shape
}

#' @export
area.square <- function(object) {
  object$side^2
}
```

`areacalc`'s directory structure should look like

```
dir_tree("~/Repositories/areacalc")
~/Repositories/areacalc
├── DESCRIPTION
├── LICENSE
└── LICENSE.md
└── NAMESPACE
```



and you should be able to do

```

load_all("~/Repositories/areacalc")
Loading areacalc
sq <- square(3)
area.square(sq)
[1] 9

```

## Name Spaces and Methods

Packages have their own name spaces and, as such, are great places for object-oriented programming. Name spaces are a fairly advanced topic for which you can find more details in the [section about name spaces in the “Advanced R” book](#); loosely, a namespace provides a context for looking up the value of an object associated with a name. As such they are vital for *encapsulating* the package and making it self-contained, ensuring that other packages do not interfere with your code.

Back to **areacalc**, using the [S3 OOP system](#) we can easily implement a *generic method* called `area()` that will calculate the area of a shape, depending on whether the object we pass to the generic method is of class "circle" (e.g. as constructed by the `circle()` function) or "square" (e.g. as constructed by the `square()` function).

The names we used for the functions `area.circle()` and `area.square()` in **areacalc** were far from accidental. In fact, the particular naming convention `generic_method.class()` we used is S3-specific and makes `area.circle()` and `area.square()` *methods* ready to be used by a *generic method* `area()`!

In order to implement the generic method, we first create a new script under `~/Repositories/areacalc/R/` (I called it `area.R`) with the following very simple generic function specification

```

area <- function (object, ...) {
  UseMethod("area")
}

```

Then, we can call the `document()` function from the **devtools** R package to update the package’s name space. `document()` uses the methods from the [roxygen2](#) package to populate the NAMESPACE file.

```

document("~/Repositories/areacalc")
Updating areacalc documentation
Loading areacalc
Writing NAMESPACE
Writing NAMESPACE

```

The `#' @export` statements before the definitions of `area.circle()` and `area.square` told **roxygen2** to generate the table of exported methods. Your `NAMESPACE` file now looks like

```
# Generated by roxygen2: do not edit by hand

S3method(area,circle)
S3method(area,square)
```

in essence telling R that **areacalc** provides two S3 methods for the generic `area()` for objects of class "circle" and "square".

Now we can do

```
ci <- circle(1)
sq <- square(1)
area(ci)
area(sq)
```

Notice here that the generic method `area()` automatically decides which specific method to dispatch (`area.circle()` or `area.square()`) depending on the class of the object we pass to it ("circle" or "square", respectively). If you want to add support for a new shape, say a rectangle, you just need to write the `rectangle()`, and `area.rectangle()` functions, add `#' @export` before the definition of the `area.rectangle()`, and rerun `document()`.

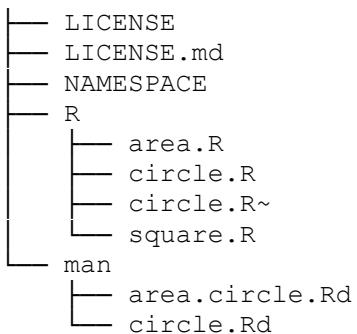
## DESCRIPTION File

The DESCRIPTION file is where we specify the package title, version, authors, licence, and much other information, including other packages that the package depends on. Go ahead and edit **areacalc**'s description file to add an informative title and your details. In my copy, I have edited the title and person specification, set the version to `0.0.1`, and used the `use_mit_license("John Doe")` function to populate the License attribute with the details of the MIT licence. After all this, my DESCRIPTION file looks like

```
Package: areacalc
Title: Methods to Define and Compute the Area of Various Shapes
Version: 0.0.1x
Authors@R:
  person(given = "John",
          family = "Doe",
          role = c("aut", "cre"),
          email = "john.doe@nowhere.com")
Description: Provides methods to define various shapes in terms of their
defining characteristics and compute their area.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
```

**areacalc**'s directory structure is now

```
dir_tree("~/Repositories/areacalc")
~/Repositories/areacalc
└── DESCRIPTION
```



## Building Packages

If we want to share the package or submit it in a repository we can do

```

build("~/Repositories/areacalc")
✓  checking for file '~/Repositories/areacalc/DESCRIPTION'
- preparing 'areacalc':
✓  checking DESCRIPTION meta-information
- checking for LF line-endings in source and make files and shell scripts
- checking for empty or unneeded directories
  Removed empty directory 'areacalc/man'
- building 'areacalc_0.0.1.tar.gz'

[1] "~/Repositories/areacalc_0.0.1.tar.gz"

```

As the output suggests, the package directory structure is now in the compressed file `areacalc_0.0.1.tar.gz`, which can be shared as is for installation and use by other users.

The `use_github()` function of the **usethis** R package allows you to upload the package in GitHub. This can also be done “less manually” by using the GUIs that RStudio provides.

## Installing

If you want to install **areacalc** version 0.01 and have it ready for use in future R sessions you simply do `install("~/Repositories/areacalc")` if you want to install directly from the package directory structure, or

`install.packages("~/Repositories/areacalc_0.0.1.tar.gz")` if you want to install from the compressed file you’ve built in earlier.

## Publishing

The easiest way to release an R package is using the `release()` from the **devtools** R package, which will guide you through the steps required for release and make recommendations about what you are expected to have done prior to release; see [section on releasing R packages in the ‘R packages’ book](#). Of course, **areacalc** is not ready yet for release. More work is needed to documenting the package, and it is also useful to incorporate some tests to make sure that the methods we developed work as expected.

# Useful Links and Resources

- [\*R packages\* book by Hadley Wickham](#): An authoritative account about how to build and publish R packages.
  - [Writing R extensions resource](#): The definitive guide for creating R packages and writing R documentation.
4. [R package primer by Karl Broman](#): A minimal tutorial about writing R packages.

# Modules and Packages in Python



## Modularization

*Modular programming* is a design technique to separate the functionality of a program into some smaller independent parts for some specific tasks. As each module is for some specific tasks, it is much easier to navigate and locate the code. Different developers can also work on different parts of the program independently. We have already seen one tool to promote code modularization: functions. With the use of functions, we can break down a complex problem into some smaller sub-problems. The use of functions also facilitates code reusing. In this page, we introduce two other tools for code modularization: *modules* and *packages*.

## Modules

In Python, a *module* is a file with the suffix `.py` that contains Python code. The name of the module is the same of the name of file. For example, the module `circle` is created by writing relevant code in the `circle.py` file.

The content of the `circle.py` file is as follows:

```
'''  
This module contains functions for circle calculation  
pi = 3.14159  
def area(radius):  
    '''  
        return area of a circle given the radius. Radius is assumed to be a  
non-negative number    return pi * (radius**2)  
  
def circumference(radius):  
    '''  
        return circumference of a circle given the radius. Radius is assumed to  
be a non-negative number    return 2 * pi * radius
```

Using modules is very similar to using packages, as we have done so far. We first import the module by `import module`. For example, to import the module `circle` we write:

```
import circle
```

By using the function `help()`, we can see how to use the module:

```
>> help(circle)
Help on module circle:

NAME
    circle - This module contains functions for circle calculation

FUNCTIONS
    area(radius)
        return area of a circle given the radius. Radius is assumed to be a
non-negative number

    circumference(radius)
        return circumference of a circle given the radius. Radius is
assumed to be a non-negative number

DATA
    pi = 3.14159
```

The docstring of this module tells us that the module aims to do circle-related calculation. The `help()` function also tells us that the module has two functions for calculating the area and circumference of a circle given the radius, and an object `pi`. While the `help()` function tells us how to *use* the module and its functions, it does not tell us how the module and its functions are *implemented*. This is *abstraction* - we hide the unnecessary details from the user.

To get the attributes (which includes functions and other objects) in the module, we use `module.attribute`. For functions, we call `module.func()`. For example, if we want to use the function `area()` from the module `circle`, we need to call the function as follows:

```
circle.area(3)
28.26
```

Similarly, if you want to get the retrieve the objects from the module, you need to write `module.obj`. For example, if we want to retrieve `pi` from the module `circle`, we need to write:

```
circle.pi
3.14
```

We will get an error if we call the functions or use other objects directly, as functions and other objects *within* the module scope.

If you need to use the functions a lot, calling the functions in the module by `module.func()` may not convenient. Instead you can import the function or other objects from the module by `from module import function`. This enables you to use the them directly:

```
from circle import area, pi
print(area(3))
28.26
print(pi)
3.14
```

# Packages

Packages are a collection of modules. We have been using packages like **NumPy** and **pandas** in this course. In each package, there is at least one `__init__.py` file. The `__init__.py` files tell Python that the directories should be treated as packages. The `__init__.py` file can just be an empty file, but often it contains some initialization code for the package.

Now let's create our own package `my_math` for calculation. We put the file `__init__.py` to tell Python this is a package. We also put the files `vector.py` and `circle.py` for the vector and circle-related calculations. The structure of the files are as follows:

```
my_math/      Top-level package
    __init__.py  Initialize the my_math package
    vector.py
    circle.py
```

We can import the package `my_math` in the same way we have imported other third-party packages:

```
import my_math.circle
my_math.circle.area(3)
28.26
```

Or we could only import the module `circle`:

```
from my_math import circle
circle.area(3)
28.26
```

# Sub-Packages

A package often has a large number of files, and a hierarchical structure is often imposed by grouping the modules files into different folders and include a `__init__.py` file in the folders. Take a look of the source code of **NumPy** from [GitHub](#) and you can see that modules are organized via folders. For example, the modules related to random sampling are under the sub-folder `numpy/random` and the modules related to polynomial calculation are under the sub-folder `numpy/polynomial`. For each sub-folder, there is another `__init__.py`. Each sub-folder therefore is a sub-package. For **NumPy**, `numpy.random` is a sub-package.

Let us create another package `my_math_2`, which with the following structure:

```
my_math_2/      Top-level package
    __init__.py  Initialize the my_math_2 package
    linear_algebra/
        __init__.py  Sub-package for linear algebra
        vector.py
        matrix.py
    geometry/      Sub-package for geometry
        __init__.py
```

```
└── circle.py  
└── triangle.py
```

To use a sub-package from the package we can import it by:

```
from my_math_2.geometry import circle  
circle.area(3)  
28.26
```

The *Python Package Index (PyPI)* is a repository of software for the Python programming language. Packages are shared

## Python Package Repositories and Package Installer

Similar to CRAN in R, Python has its own software repositories. Python packages are typically installed from one of two package repositories:

- The Python Package Index (PyPI)
- Conda

PyPI is the official third-party software repository for Python, and it is the default source for packages for `pip`, which is a popular package installer for Python. The name Conda is for both the general-purpose package management system and the package repository. Unlike PyPI, Conda manages software of *any* language.

## Publishing a Package

To publish the package, see the [official Python document](#) and [PyPI](#)

## Useful Links and Resources

- [Official Python tutorial on modules and packages](#)
- [Official Python tutorial on packaging Python projects](#)

# Documenting Code



## Documentation

You have (hopefully) typed `help` or `?` at least twice in this course while studying the workings of R and Python, and, hence, can appreciate how important it is for software to be documented well. Documentation allows both developers and users to at least get an idea what a function or package does and how, or what particular objects are, without necessarily having to read the code. In fact, it would be almost impossible to write Python or R code without having access to documentation of Python or R!

## Documenting R Code

R objects are documented in files with extension `.Rd`, written in “R documentation” (`Rd`) format. `Rd` is a simple markup language much of which closely resembles (La)TeX and can be processed into a variety of formats, including LaTeX, HTML and plain text. Typically, documentation files are distinct from the script files of an R package, put under the `man/` directory in the directory structure of an R package. A clear description of how `.Rd` files should be authored and the conventions for their elements is at [the “Writing R extensions” resource](#).

The `roxygen2` R package considerably simplifies the documentation of R objects, by enabling users to document them “in place,” directly in the scripts that they are defined. To illustrate, recall the `circle.R` in the `R/` directory of `areacalc`’s directory structure. If we open it we see

```
circle <- function(radius) {  
  shape <- list(radius = radius)  
  class(shape) <- "circle"  
  shape  
}  
  
#' @export  
area.circle <- function(object) {  
  pi * object$radius^2  
}
```

Both `circle()` and `area.circle()` are currently not documented; that is if you type `?circle` or `?area.circle` no guidance on how these functions are used comes up. In order to add some documentation we edit the file to look like

```
#' Define a circle through its radius  
#'
```

```

#' This returns an object of class `"circle"` which specifies a circle
through its radius.
#'
#' @param radius a [`numeric`] of length one.
#' @return A list inheriting from `"circle"` with a single element
`"radius`".
#' @seealso [area.circle()]
#' @examples
#' ci <- circle(2.2)
#' ci
#' @export
circle <- function(radius) {
  shape <- list(radius = radius)
  class(shape) <- "circle"
  shape
}

#' Compute the area of a circle generated using [circle()]
#'
#' This computes the area of a circle as defined by the [circle()].
#'
#' @param object an object of class `"circle"`, as generated for example
using the function [circle()].
#' @seealso [circle()]
#' @examples
#' ci <- circle(3)
#' area(ci)
#' @export
area.circle <- function(object) {
  pi * object$radius^2
}

```

As you notice **roxygen2** comments start with `#'`. In this file we have added documentation for two functions, with the documentation appearing *before* the definition of each function. The first line of the documentation is the title of the help file. The following line is a short description of what the function does. `@param` followed by the argument name is used to document each of the function arguments; for functions with multiple arguments we use multiple lines starting with `@param` followed by the argument name. `@seealso` provides links to other related objects, and after `@examples` we have a code chunk that illustrates what the function does. `@export` is a statement we have already seen last week, and it is used to tell `document()` that we want this function to be visible to the users of the package. Everything else is just plain Markdown markup. A detailed account of the conventions used and how Markdown is translated in `.Rd` markup can be found in [roxygen2's pages](#).

Now,

```
document("~/Repositories/areacalc")
```

will parse all scripts, and produce appropriate `.Rd` files under `man/`. If you install **areacalc** and type `?circle` you will now see the help page for the function `circle()`.

The basic steps for creating documentation for objects in an R package are:

- Add **roxygen2** comments to your script files
- Run `document()` to convert these **roxygen2** comments to `.Rd` files

- Preview documentation using `? or help()` on the function names
- Repeat this process until you are satisfied with the documentation

As an exercise, try documenting all methods and generics in the **areacalc** package. Make sure you consult [roxygen2's pages](#) or search in the web if anything does not work for you.

## Documenting Python Code

Documenting Python code and projects is typically done using Docstrings. We are not going to get into the details, but if you are interested a very clear tutorial is provided at [realpython.com's pages](#). [Sphinx](#) is another popular tool (with very detailed documentation!) that makes it easy to create intelligent documentation for Python (and not only) projects.

## Useful Links and Resources

- [Writing R documentation files from the Writing R extensions resource](#).
- [roxygen2's pages](#): Guides and tutorials on how to use **roxygen** for documentations of R objects.
- [realpython.com's pages](#): A tutorial on how do document Python code.
- [Sphinx](#): A popular tool for generating intelligent and good-looking documentation for Python code and more.

# Test-Driven Software Development

## Software Testing

Most probably, you have already been testing things out while writing code, by inspecting the output of your programs and, if it is not as expected, going back to the code, inspecting the output again, and so on, until you are happy with the result.

As can be inferred from the various software development models we introduced last week, testing is not only a vital part of software development process, but the various SDLC models (see “Introduction to Software Development” section) also define when and how software testing is done. Testing your code ensures that it does what you want it to do.

[Wikipedia’s page on software testing](#) provides a good overview of the many software testing approaches (e.g. white-box testing, black-box testing, grey-box testing) and software testing levels (unit testing, integration testing, system testing, acceptance testing) that have been devised.

In this page we will focus on *unit testing* because of its usefulness and importance when developing software for data science projects.

## Unit Testing

According to [Wikipedia’s page on software testing](#)

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors

Unit testing refers to tests of the components of the software (e.g. functions and classes) to verify that those and the software itself work as expected. It is typically done using inputs and testing against an expected output. Remarkably, the most experienced programmers start by writing out the test before even writing the code itself!

For example, suppose you intend to add a `rectangle()` functions in **areacalc**. An experienced programmer would first write some unit tests for that function, perhaps the most basic of those being

```
isTRUE(rectangle(length = 2, width = 3)$length == 2)
Error in rectangle(length = 2, width = 3): could not find function
"rectangle"
isTRUE(rectangle(length = 2, width = 3)$width == 3)
Error in rectangle(length = 2, width = 3): could not find function
"rectangle"
isTRUE(class(rectangle(length = 2, width = 3)) == "rectangle")
```

```
Error in rectangle(length = 2, width = 3): could not find function
"rectangle"
```

Of course, all tests above *fail* with errors, simply because the code for `rectangle()` has not been written yet! This may seem like a waste of time, but it pays off very quickly.

Let's think for a moment what have we achieved by writing the tests: i) We have determined that for a well-specified rectangle ones needs to specify a length and a width, ii) We thought about the interface of the `rectangle()` function, i.e. it should take as input the arguments `length` and `width`, iii) We decided that the result should be a list with elements `length` and `width` and of class "rectangle," iv) We set ourselves the goal to write code for passing the tests. So, oftentimes by writing the tests you have almost written the code! In this case,

```
rectangle <- function(length, width) {
  out <- list(length = length, width = width)
  class(out) <- "rectangle"
  out
}
```

and then, all unit tests above pass

```
isTRUE(rectangle(length = 2, width = 3)$length == 2)
[1] TRUE
isTRUE(rectangle(length = 2, width = 3)$width == 3)
[1] TRUE
isTRUE(class(rectangle(length = 2, width = 3)) == "rectangle")
[1] TRUE
```

Imagine doing the above throughout development and for as many components of your code as possible, testing results, corner cases, exceptions, and so on. You are not only more confident that your code does exactly what you want it to do (validation), but you have also eliminated many potential defects in your code. You also have a list of ready tests to run whenever you change anything to see if anything breaks, hence improving maintainability and making code refactoring easier.

It is inevitable that your tests will eventually fail after changing your code! Once they fail, though, you already know what failed and, if your unit tests are granular enough, you also know why. Then, you can use the techniques (debugging and error handling) and the strategies (e.g. defensive programming) we discussed on the "Exceptions, Error Handling and Debugging in R" section to both fix the issue and improve your code.

## Testing Frameworks in R

There is nothing wrong with doing unit testing by incrementally adding tests into a script, which you then run whenever you make changes or additions to the software.

However, it is best to use one of the specialized dedicated R packages, like [testthat](#) and [tinytest](#), which allow you to unit test in a more effective and structured manner, and also integrate the unit tests within your R package. For example, with [tinytest](#) we can write the above unit tests, and a few more for `areacalc`, as

```

library("devtools")
Loading required package: usethis
load_all("~/Repositories/areacalc")
Loading areacalc
library("tinytest")
re <- rectangle(length = 2, width = 3)
expect_true(is.list(re))
---- PASSED : <-->
call| expect_true(is.list(re))
expect_identical(re$length, 2)
---- PASSED : <-->
call| expect_identical(re$length, 2)
expect_identical(re$width, 3)
---- PASSED : <-->
call| expect_identical(re$width, 3)
expect_identical(class(re), "rectangle")
---- PASSED : <-->
call| expect_identical(class(re), "rectangle")
expect_error(circle(diameter = 3))
---- PASSED : <-->
call| expect_error(circle(diameter = 3))
expect_equal(area(circle(1)), pi)
---- PASSED : <-->
call| expect_equal(area(circle(1)), pi)

```

Both **testthat** and **tinytest** have detailed vignettes and tutorials on how they can be used and be integrated within an R package; just follow the links on their CRAN pages. As a note of caution, it is not recommended to mix **tinytest** and **testthat** unit testing, or to have them both loaded at the same R session.

## Testing Frameworks in Python

Python, like R, provides unit testing frameworks, like [unittest](#) and [pytest](#). [The unit testing tutorial by Anthony Shaw](#) is an accessible tutorial about unit testing in Python.

## Useful Links and Resources

- [Wikipedia's page on software testing](#)
- [“Getting started with testing in Python” by Anthony Shaw](#)