

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 it is 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.

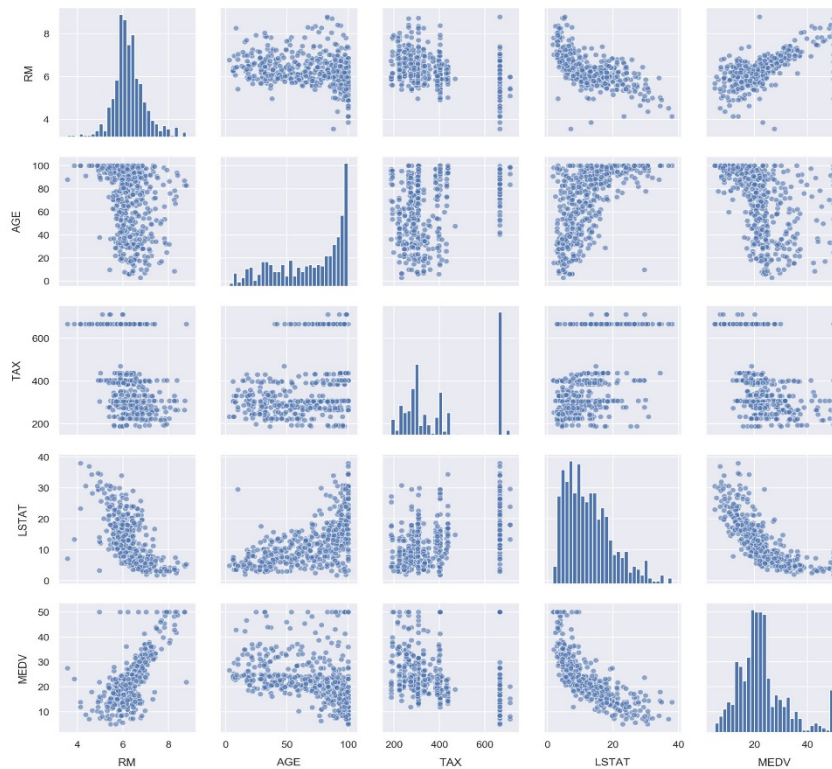
TABLE 1.1. *Average percentage of words or characters in an email message equal to the indicated word or character. We have chosen the words and characters showing the largest difference between **spam** and **email**.*

	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.

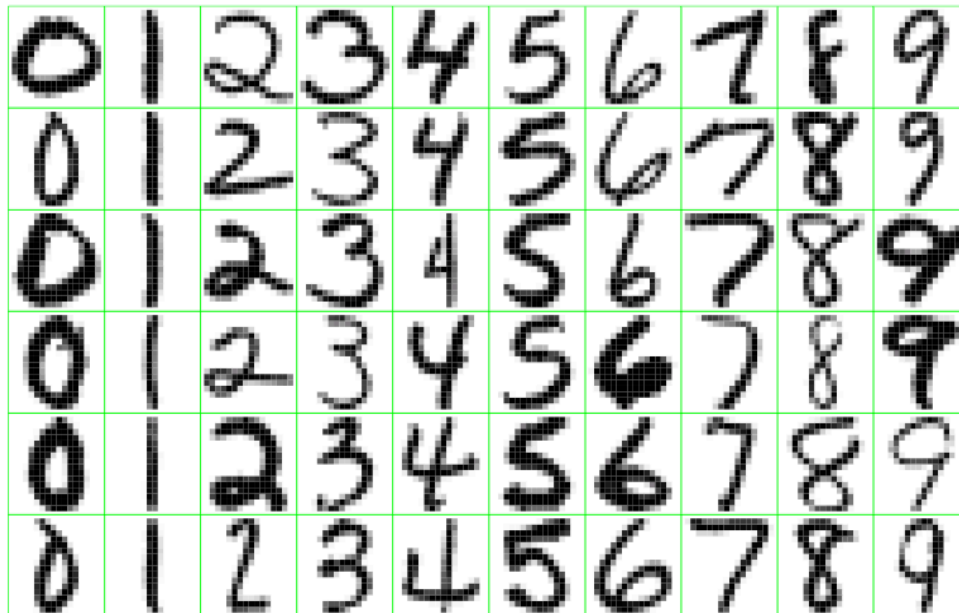


FIGURE 1.2. *Examples of handwritten digits from U.S. postal envelopes.*

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 [autocompletion](#), [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).

Editor	Platform	Open-Source
Visual Studio Code	Windows, macOS, Linux	Yes
Notepad++	Windows	Yes
Vim	Windows, macOS, Linux	Yes
Sublime Text	Windows, macOS, Linux	No, Shareware
Atom	Windows, macOS, Linux	Yes
Jupyter	Windows, macOS, Linux	Yes
Emacs	Windows, macOS, Linux	Yes
TextMate	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

IDE	Platform	Open-Source	Support for R	Support for Python
Spyder	Windows, macOS, Linux	Yes	No	Yes
RStudio	Windows, macOS, Linux	Yes	Yes	Yes (in recent versions)
Eclipse	Windows, macOS, Linux	Yes	Yes (via plugin)	Yes (via plugin)
Microsoft Visual Studio	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](https://cran.r-project.org/) (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](https://cran.r-project.org/)
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 minimalists choice
- Through the default R GUI (on Window 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] 8.00000000729
```

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`.”

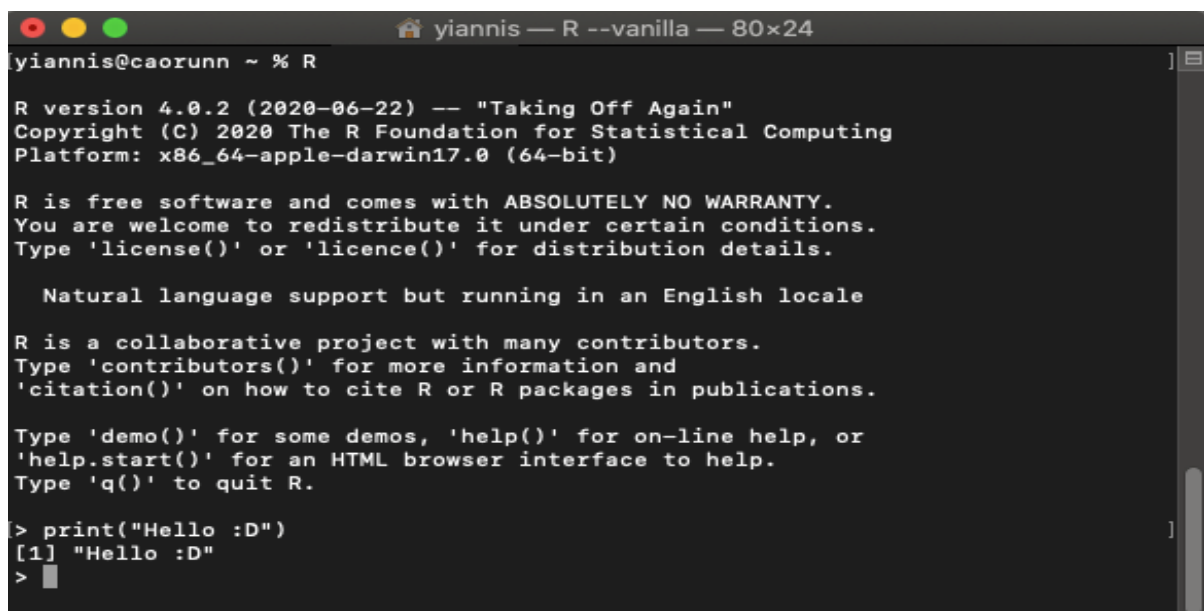
Through the Terminal

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

A screenshot of a macOS Terminal window. The title bar shows the window name "yiannis — R --vanilla — 80x24". The prompt is "yiannis@caorunn ~ % R". The output shows the R version 4.0.2 (2020-06-22) and copyright information. It also displays the R license and some helpful messages. The user has entered the command `print("Hello :D")` and the output is `[1] "Hello :D"`.

```
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

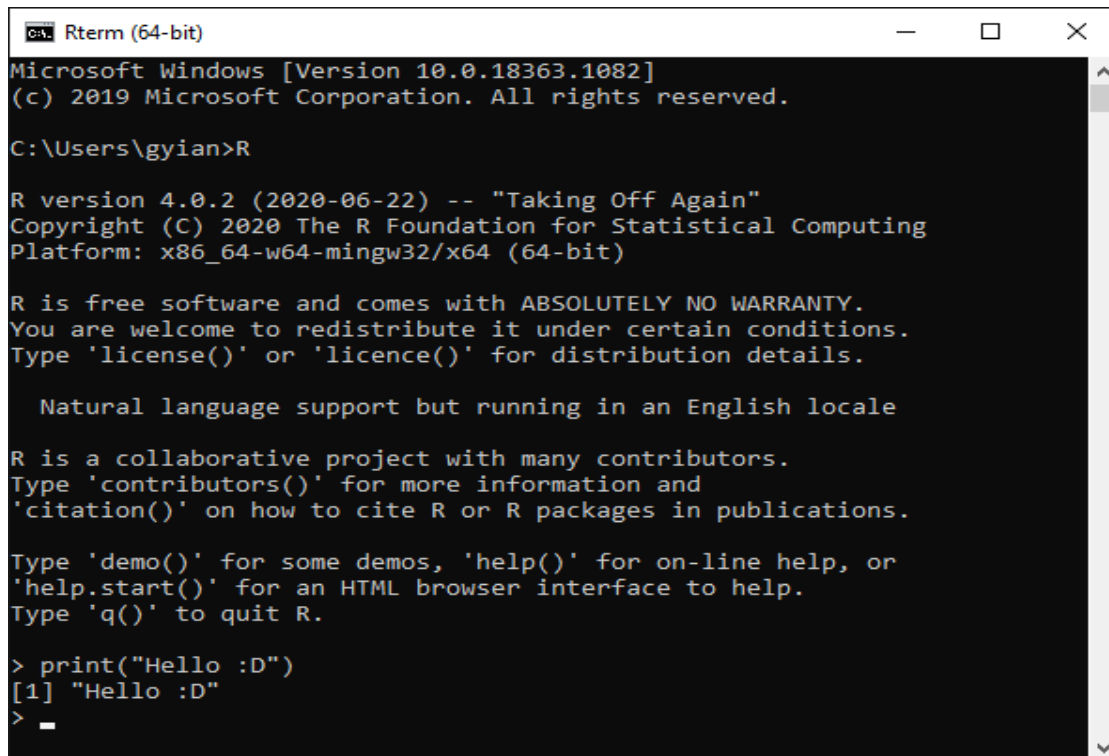
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

A screenshot of a Windows Command Prompt window titled "Rterm (64-bit)". The window shows the output of running the R command. It displays the Microsoft Windows version (10.0.18363.1082), copyright information (© 2019 Microsoft Corporation), and the current directory (C:\Users\gyian). The R version (4.0.2) and platform (x86_64-w64-mingw32/x64) are also shown. The R startup message includes a disclaimer about warranty and a welcome to redistribute it under certain conditions. It also mentions natural language support running in an English locale. The R prompt (>) is shown, and the command print("Hello :D") is entered, resulting in the output [1] "Hello :D".

```
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`:

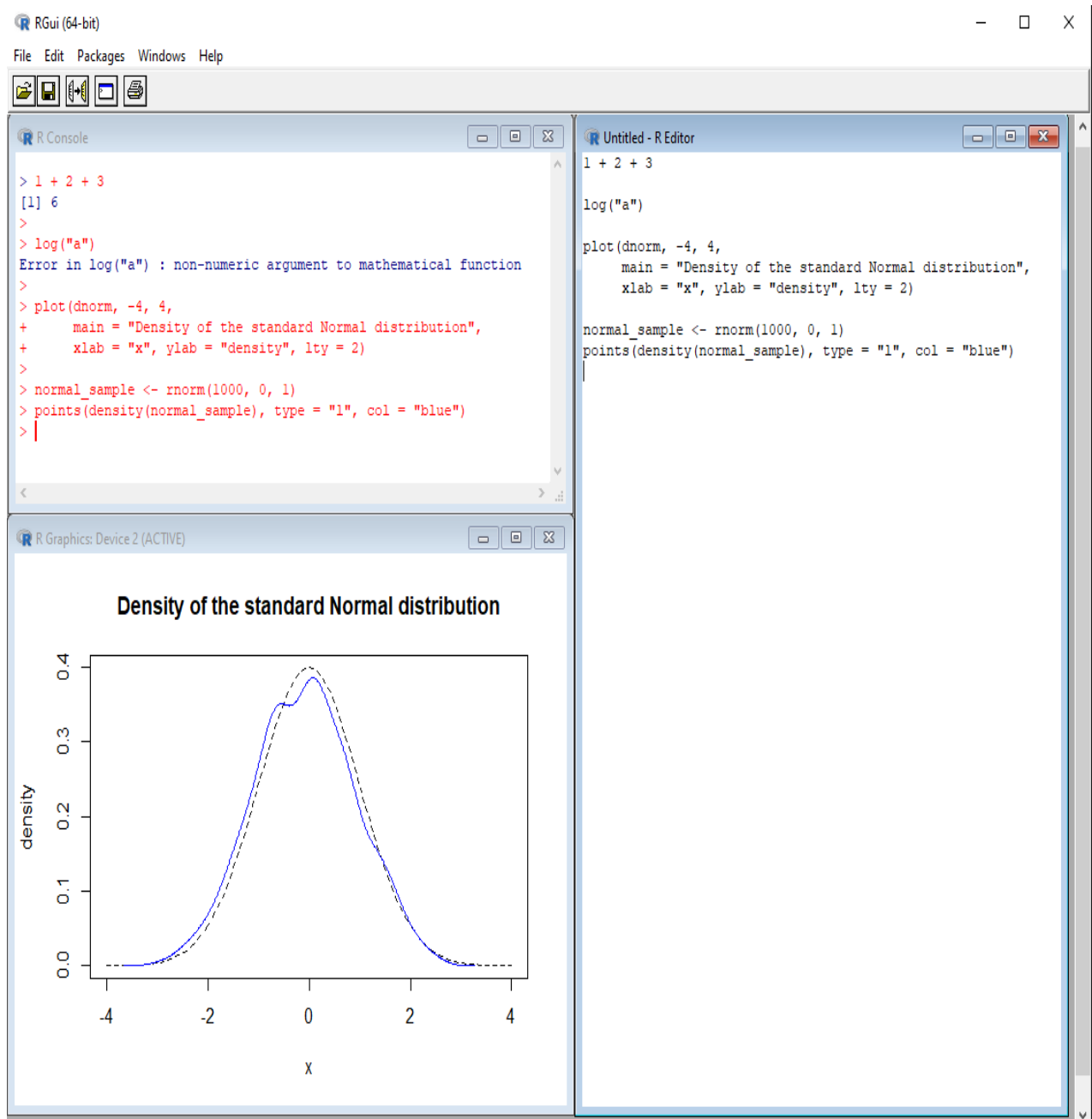
```
> a <- log(2) + 3
> a
[1] 3.693147
```

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

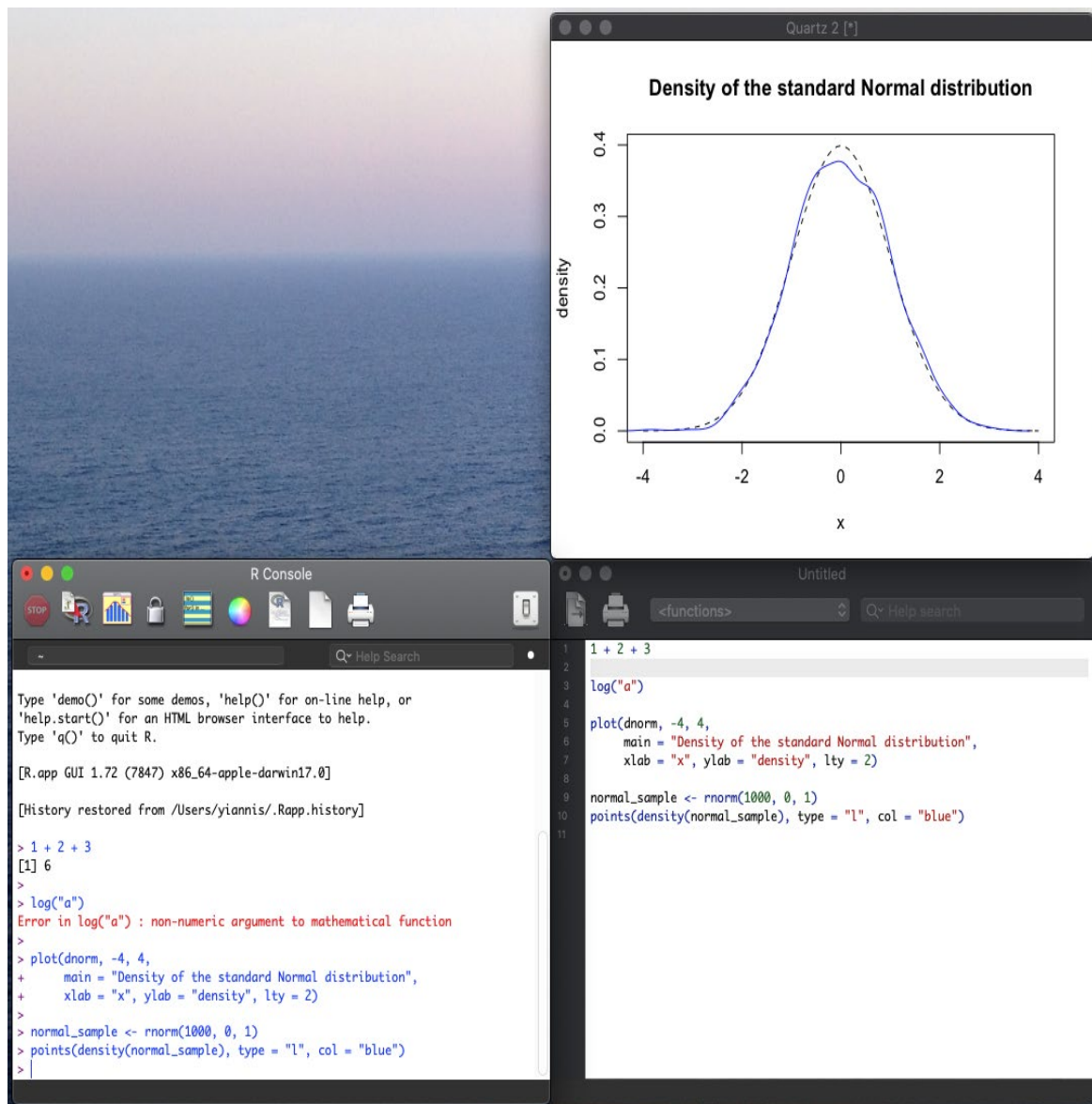
On Windows and macOS machines the R installer installs by default a custom graphical user interface (GUI) for R.

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](#).

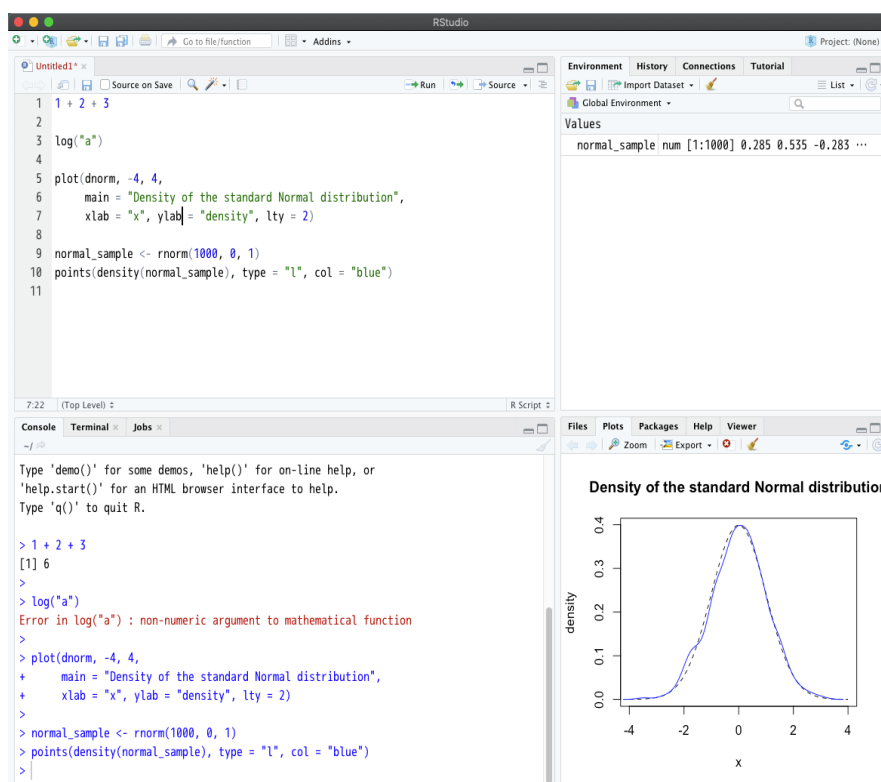
RStudio can be downloaded from [here](#). It comes in two flavours: one is RStudio Desktop, which is a stand-alone computer applications and the other is RStudio Server which runs on a remote server and allows accessing RStudio using a web browser. For the purposes of this course, you will only need 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, John 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, John 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 a 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

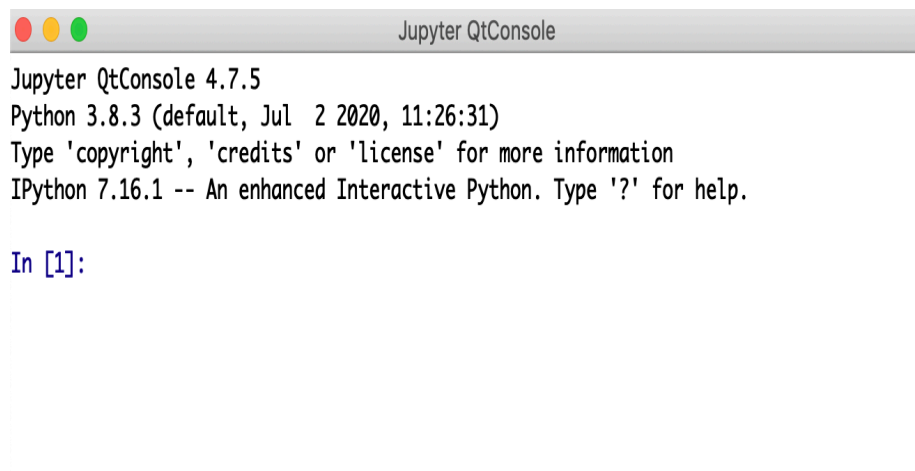
In this set of notes we will work with [QtConsole](#), Jupyter Notebook, and Spyder to perform basic operations in Python.

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



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

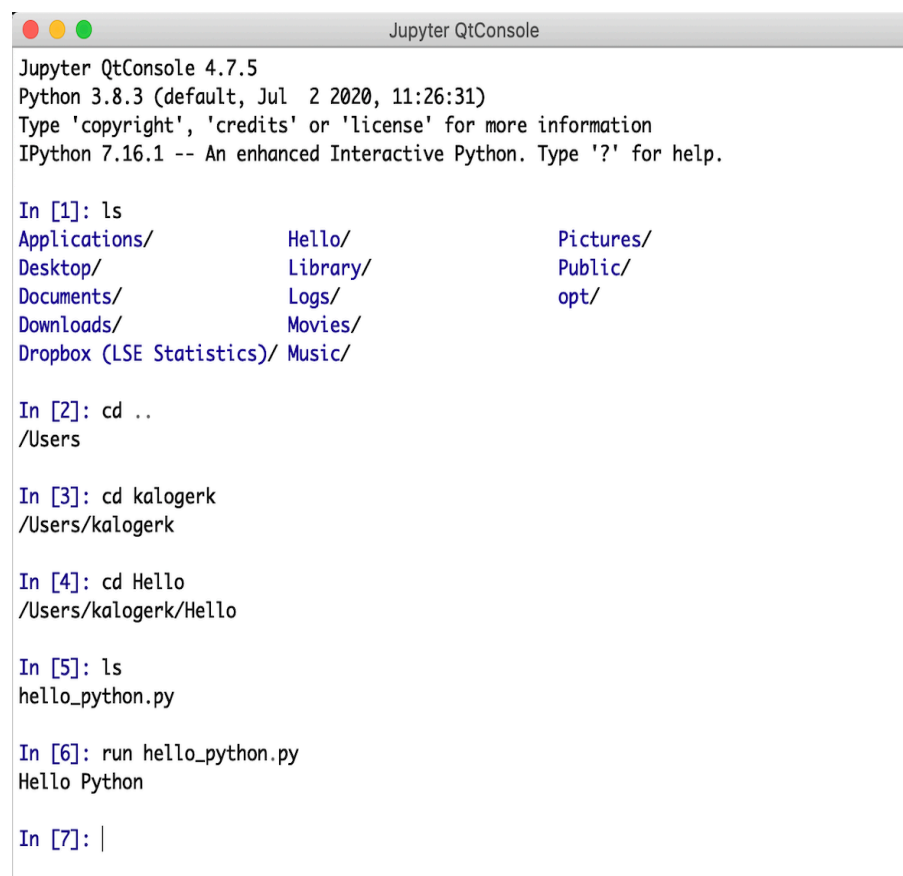
```
run hello_python.py
```

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`:



```
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/
Dropbox (LSE Statistics)/ Music/

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

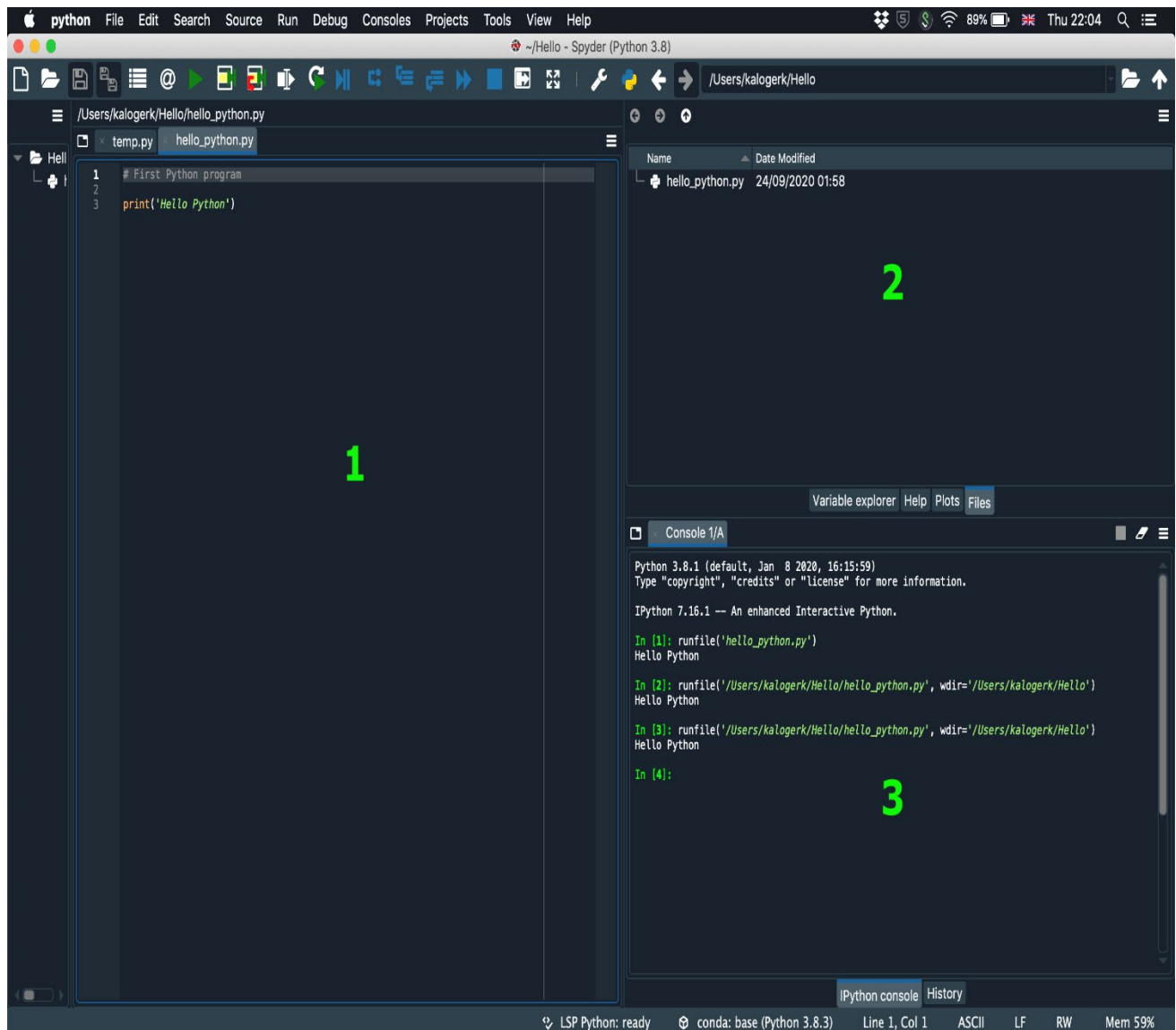
In [7]: |
```

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

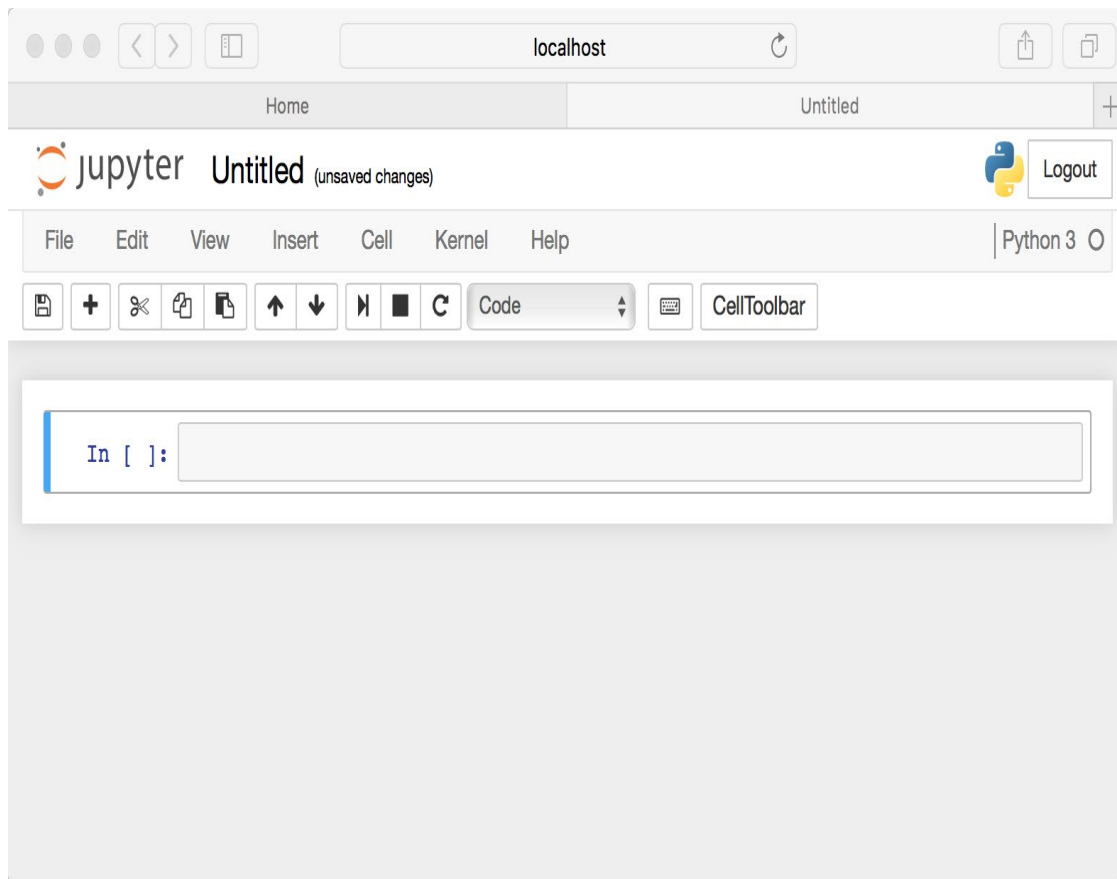
In order to launch open the Anaconda Navigator and click on Jupyter Notebook launch icon, i.e. in the same way as with QtConsole and Spyder.

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

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 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 96a5cf143fcel135a7db11aeb5776ea9bff353fcc
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).

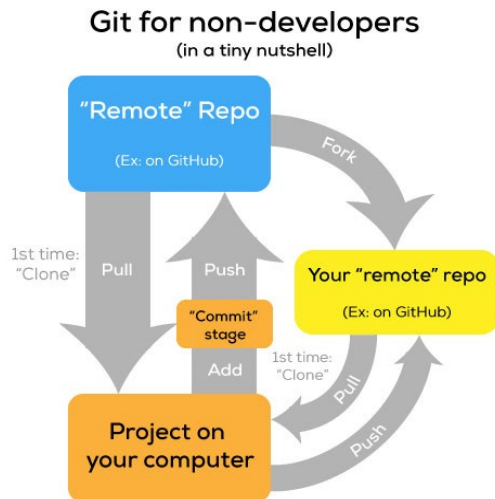
Useful Links and Resources

- [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.