

R Programming



Sana Rasheed

AL NAFI,

A company with a focus on education, wellbeing and renewable energy.



Course Outline

In this course you will learn:

- Section 1: Overview of R Programming
- Section 2: Features of R Programming Language
- Section 3: Installation & Configuration of R, RStudio & Notebook for R
- Section 4: Version Control and Github (**Important**)
- Section 5: Data Types and Basic Operations I
- Section 6: Getting Help in R
- Section 7: Data Types and Basic Operations II
- Section 8: Control Structures
- Section 9: Functions
- Section 10: Coding Standards
- Section 11: Vectorized Operations
- Section 12: Date and Time in R
- Section 13: Loop Functions
- Section 14: Data Split





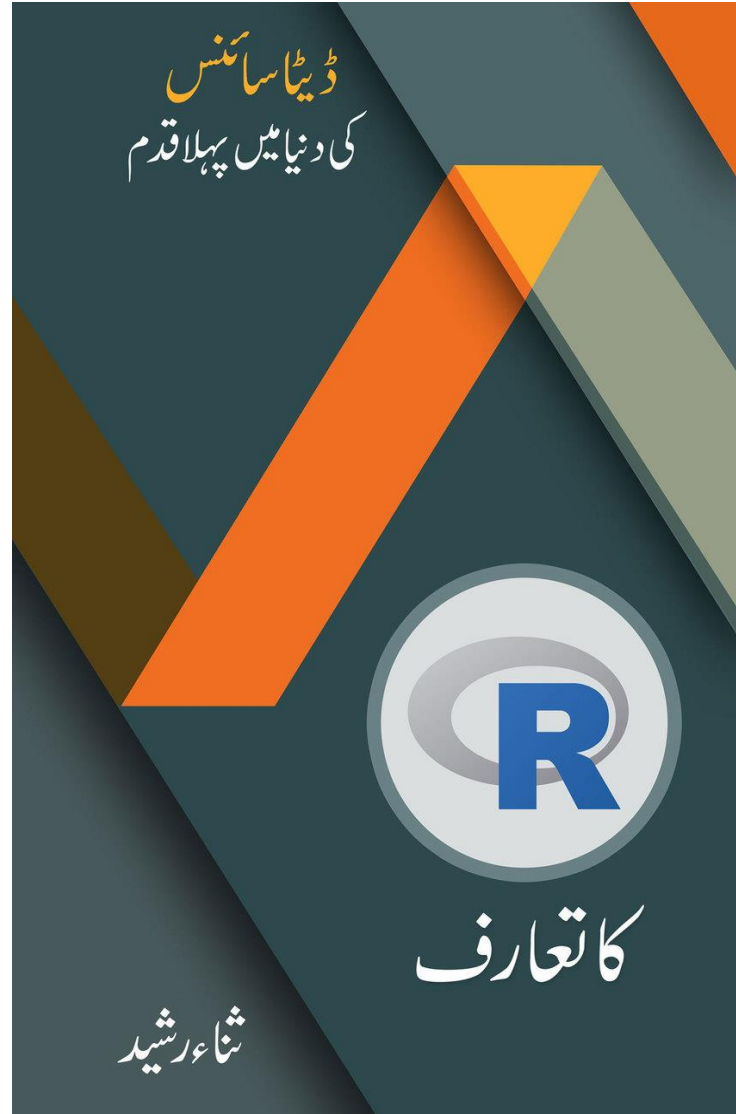
Overview of the R Language

Section 1, 2



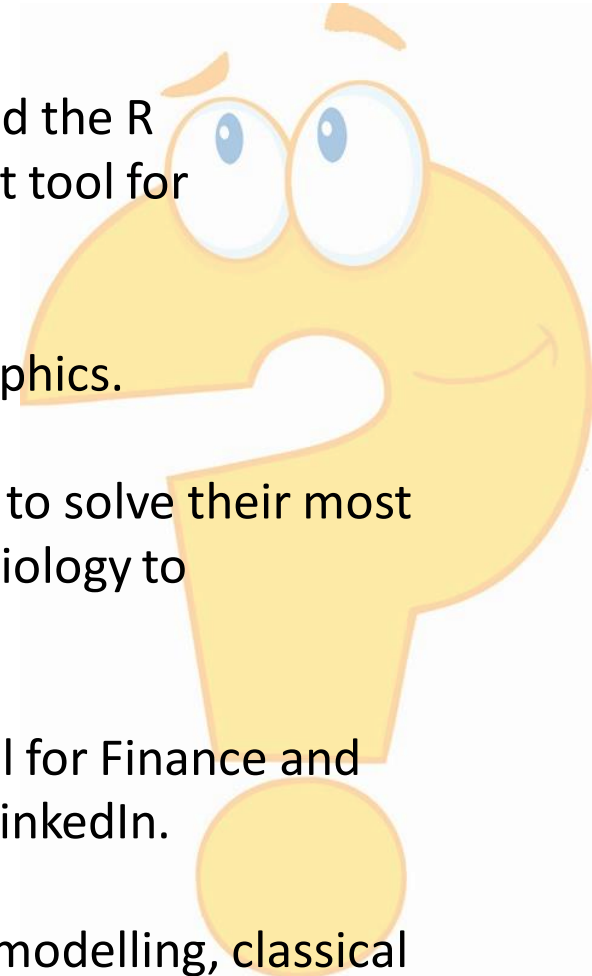
Book to Read

<https://gufhtugu.com/r/>



What is R?

- During the last decade, both academia and industry has lifted the R programming language to become the single most important tool for computational statistics, visualization and data science.
- Language and environment for statistical computing and graphics.
- Worldwide, millions of statisticians and data scientists use R to solve their most challenging problems in fields ranging from computational biology to quantitative marketing.
- Most popular language for data science and an essential tool for Finance and analytics-driven companies such as Google, Facebook, and LinkedIn.
- R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible.



Features of R

- Runs on almost any standard computing platform/OS (even on the PlayStation 3)
- Frequent releases (annual + bugfix releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user -> programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow
- And... **It's free!**



Free Software

With *free software*, you are granted

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.



The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes:

- An effective data handling and storage facility.
- A suite of operators for calculations on arrays, in particular matrices.
- A large, coherent, integrated collection of intermediate tools for data analysis.
- Graphical facilities for data analysis and display either on-screen or on hardcopy.
- A well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term “environment” is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.



Design of the R System

The R system is divided into 2 conceptual parts:

1. The “base” R system that you download from CRAN
2. Everything else.

R functionality is divided into a number of ***packages***:

1. The “base” R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.
2. The other packages contained in the “base” system include **utils**, **stats**, **datasets**, **graphics**, **grDevices**, **grid**, **methods**, **tools**, **parallel**, **compiler**, **splines**, **tcltk**, **stats4**.
3. There are also “Recommend” packages: **boot**, **class**, **cluster**, **codetools**, **foreign**, **KernSmooth**, **lattice**, **mgcv**, **nlme**, **rpart**, **survival**, **MASS**, **spatial**, **nnet**, **Matrix**.



Design of the R System

And there are many other packages available:

- There are about 4000 packages on CRAN that have been developed by users and programmers around the world.



Some Useful Resources on R

A longer list of books is available at

- <http://www.r-project.org/doc/bib/R-books.html>

For a quick review of code and syntax, check out cheat sheets

- <https://rstudio.com/resources/cheatsheets/>





Installation Guide

Section 3



R Installation Guide

Install the R language distribution from <https://cran.r-project.org/>. Choose your respective OS and download. Launch the installer and follow the instruction.



[CRAN](#)
[Mirrors](#)
[What's new?](#)
[Task Views](#)
[Search](#)

[About R](#)
[R Homepage](#)
[The R Journal](#)

[Software](#)
[R Sources](#)
[R Binaries](#)
[Packages](#)
[Other](#)

[Documentation](#)
[Manuals](#)
[FAQs](#)
[Contributed](#)

The Comprehensive R Archive Network

Download and Install R

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

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

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

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2020-02-29, Holding the Windsock) [R-3.6.3.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#)

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the [R project homepage](#) for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN [mirror](#) nearest to you to minimize network load.



RStudio Installation Guide

For IDEs, you have 2 options. We install Rstudio or use Jupyter Notebooks.

Install RStudio from <https://rstudio.com/products/rstudio/download/>



Choose Your Version

RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, and a variety of robust tools for plotting, viewing history, debugging and managing your workspace.

[LEARN MORE ABOUT RSTUDIO FEATURES](#)



RStudio's new solution for every professional data science team. RStudio Team includes RStudio Server Pro, RStudio Connect and RStudio Package Manager.

[LEARN MORE](#)

RStudio Desktop

Open Source License

Free

RStudio Desktop

Commercial License

\$995 /year

RStudio Server

Open Source License

Free

RStudio Server Pro

Commercial License


\$4,975 /year



RStudio Installation Guide


For Rstudio, open Anaconda Navigator and in the Home tab, you will find Rstudio, click on Install. This will take some time.


 ANACONDA NAVIGATOR


 Upgrade Now

Sign in to Anaconda Cloud

 Home

 Environments

 Learning

 Community

Documentation

Developer Blog



Applications on

base (root)

Channels

Refresh



console_shortcut

0.1.1

Console shortcut creator For Windows (using menuinst)

Launch



JupyterLab

1.1.4

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Launch



Notebook

6.0.1

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Launch



powershell_shortcut

0.0.1

Launch



Qt Console

4.5.5

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Launch



Spyder

3.3.6

Scientific PYTHON Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Launch



VS Code

1.44.2

Streamlined code editor with support for development operations like debugging, task running and version control.

Launch



Glueviz

0.15.2

Multidimensional data visualization across files. Explore relationships within and among related datasets.

Install



Orange 3

3.23.1

Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

Install



RStudio

1.1.456

A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Install



Jupyter Notebook for R - Installation Guide

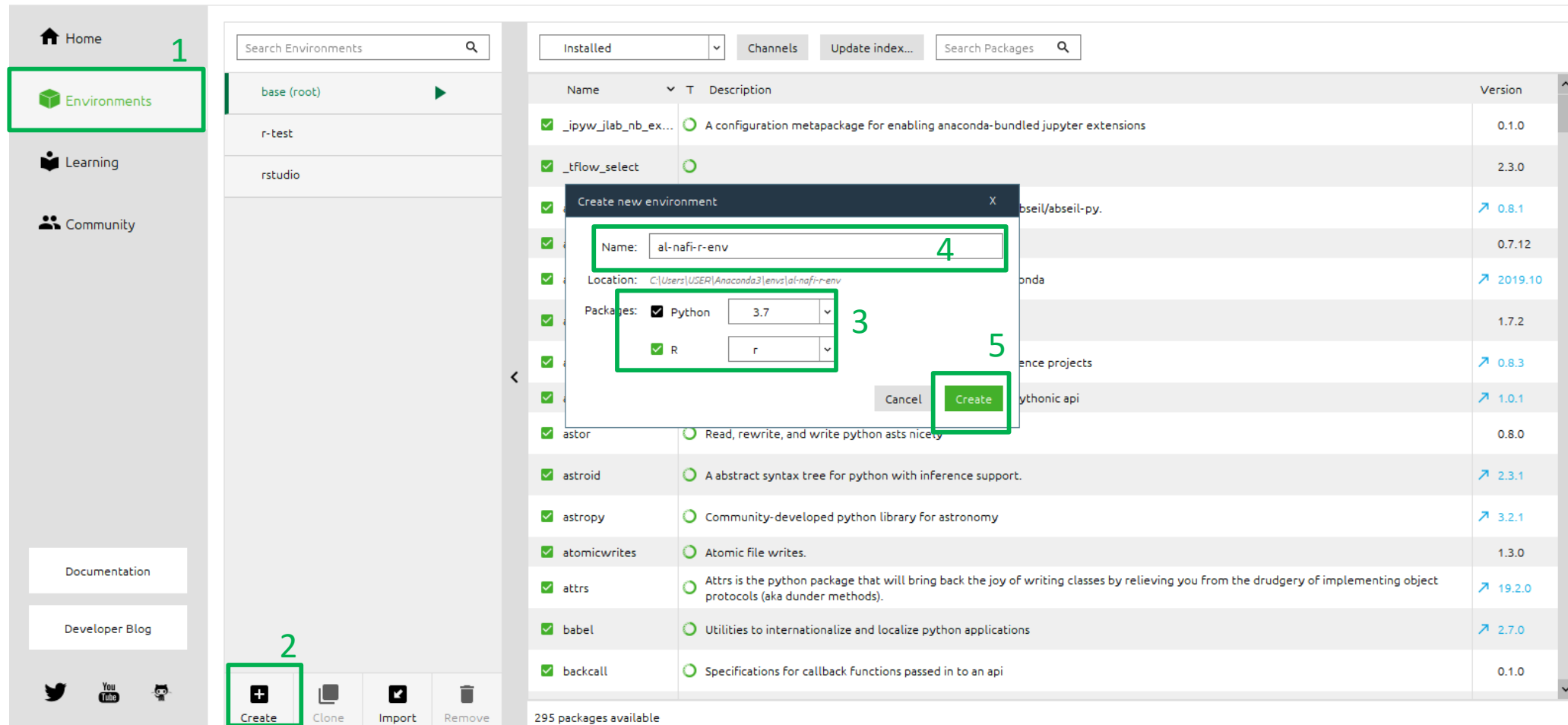
For Jupyter Notebook option

1. Select Environments option from the left panel.
2. Click on the create button and you will be prompted with the following dialogue box.
3. Select Python 3.7 and check R checkbox and select r from the dropdown corresponding to it.
4. Enter environment name
5. Click on Create. This will take some time.

 ANACONDA NAVIGATOR



[Sign in to Anaconda Cloud](#)



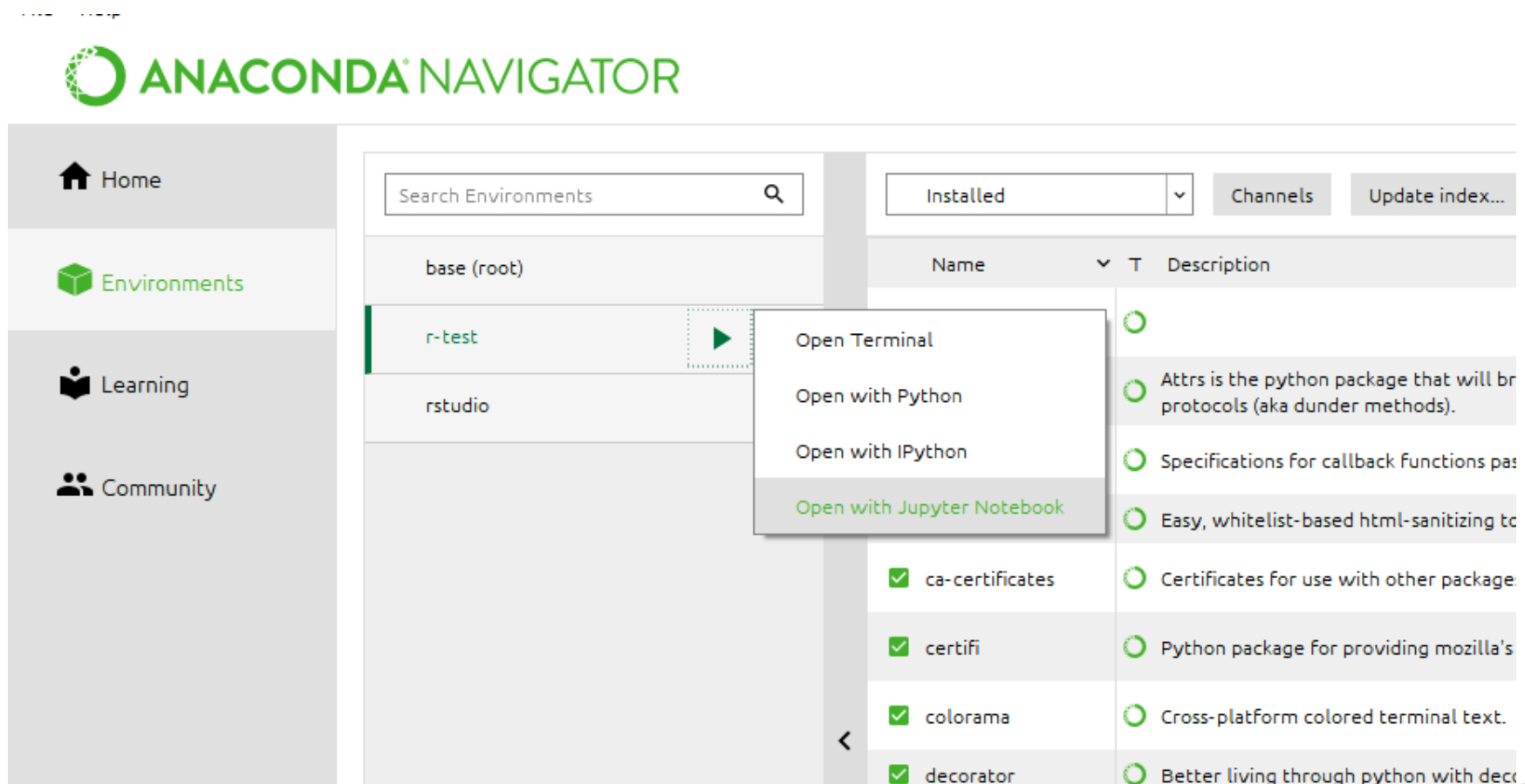
The screenshot shows the Anaconda Navigator application. On the left sidebar, the 'Environments' option is highlighted with a green box and the number 1. Below it, the 'Create' button is highlighted with a green box and the number 2. The main panel displays a list of installed environments: 'base (root)', 'r-test', and 'rstudio'. A 'Create new environment' dialog box is open in the center, with the following details:

- Name: al-nafi-r-env (highlighted with a green box and the number 4)
- Location: C:\Users\USER\Anaconda3\envs\al-nafi-r-env
- Packages:
 - ☒ Python 3.7 (highlighted with a green box and the number 3)
 - ☒ R r (highlighted with a green box and the number 5)

The 'Create' button in the dialog box is also highlighted with a green box and the number 5. The background shows a list of installed packages with columns for Name, Description, and Version.

Jupyter Notebook for R - Installation Guide

Once you are done, select the same environment that just got created and click on the play button. From the dropdown, select Open with Jupyter Notebook. This will launch Jupyter Notebook in your browser.



Installation Guide

Open the Jupyter Notebook tab in your browser. Navigate to your required directory or create a new folder, where you want to store your R files, and then click on New. You will now see an option for R. Click on this and it will launch a notebook powered by R ready for you to use it.

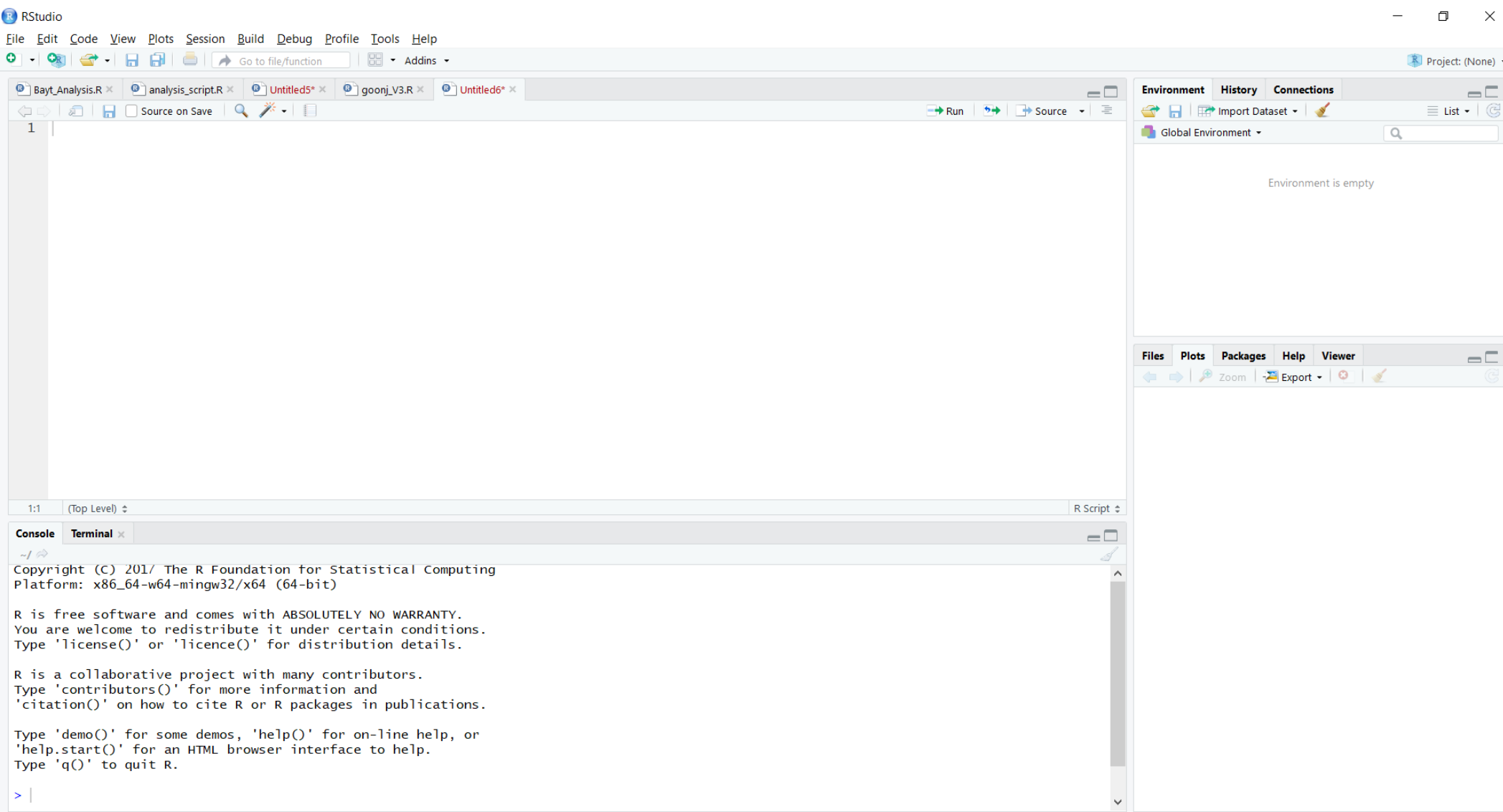


The screenshot shows the Jupyter Notebook web interface. At the top, there's a header with the Jupyter logo and 'Quit' and 'Logout' buttons. Below the header, there are tabs for 'Files', 'Running', and 'Clusters'. The 'Files' tab is active, showing a file browser. A message says 'Select items to perform actions on them.' Above the file list are 'Upload', 'New', and a refresh icon. The 'New' dropdown menu is open, showing options for 'Notebook' (Python 3, R), 'Other' (Text File, Folder, Terminal), and a 'Create a new notebook' button. The file list shows various system folders like '3D Objects', 'Anaconda3', 'Contacts', 'Desktop', 'Documents', 'Downloads', 'Favorites', 'Links', 'Music', 'OpenVPN', 'Pictures', 'Saved Games', 'Searches', 'Videos', and a file 'hw1.ipynb'.

Name	Modified	Size
0		
3D Objects		
Anaconda3		
Contacts		
Desktop		
Documents		
Downloads		
Favorites		
Links		
Music		
OpenVPN		
Pictures		
Saved Games		
Searches		
Videos		
hw1.ipynb	24 days ago	34 kB



Overview of RStudio Interface





Version Control

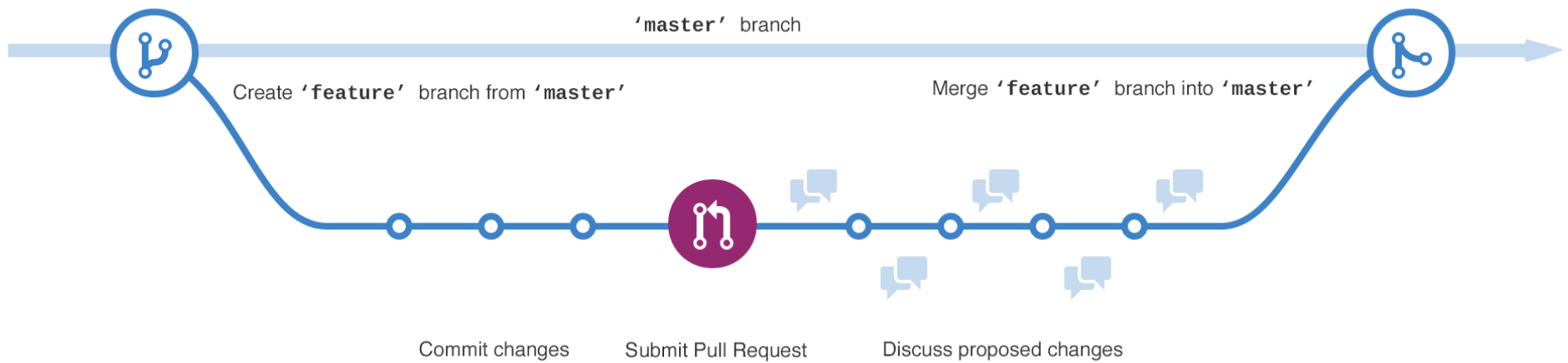
Section 4

Version Control

- Version control is a system that records changes that are made to a file or a set of files over time.
- As you make edits, the version control system takes snapshots of your files and the changes, and then saves those snapshots so you can refer or revert back to previous versions later if need be
- Have you ever used the “Track changes” feature in Microsoft Word?
- **Major benefit of version control**: It keeps a record of all changes made to the files. This can be of great help when you are collaborating with many people on the same files - the version control software keeps track of who, when, and why those specific changes were made. It’s like “Track changes” to the extreme!



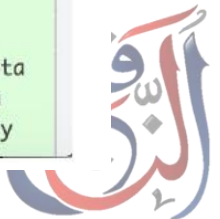
Version Control



Version Control

The screenshot shows the RStudio 'Review Changes' interface. At the top, there are tabs for 'Changes' and 'History', and a dropdown menu showing 'master' and '(all commits)'. A search bar and a 'Pull' button are also visible. Below this, a list of commits is displayed, each with a red circle icon, a description, the author, the date, and the SHA hash. The selected commit is 'Initial commit of lecture 18, Big data (no script)' by Jane Everyday Doe, dated 2018-05-02, with SHA hash ec9f5a78. Below the list, there are navigation buttons and a summary of the selected commit, including the SHA, author, date, subject, and parent. The file 'manuscript/18_Big_data.md' is highlighted, and its content is shown in a text editor. The content of the file is as follows:

```
@@ -0,0 +1,58 @@
1 # Big Data
2
3 A term you may have heard of before this course is "Big Data" - there have always been large datasets, but it seems like
  lately, this has become a buzzword in data science. But what does it mean?
4
5 ### What is big data?
6
7 We talked a little about big data in the very first lecture of this course. As the name suggests, big data are very large data
  sets. We previously discussed three qualities that are commonly attributed to big data sets: Volume, Velocity, Variety. From
  these three adjectives, we can see that big data involves large data sets of diverse data types that are being generated very
```



Version Control Software



Git:

- Git is a free and open source version control system. It was developed in 2005 and has since become the most commonly used version control system around!
- It allows you to view and manage changes in your repository. You will be required to install it: <https://git-scm.com/download/>

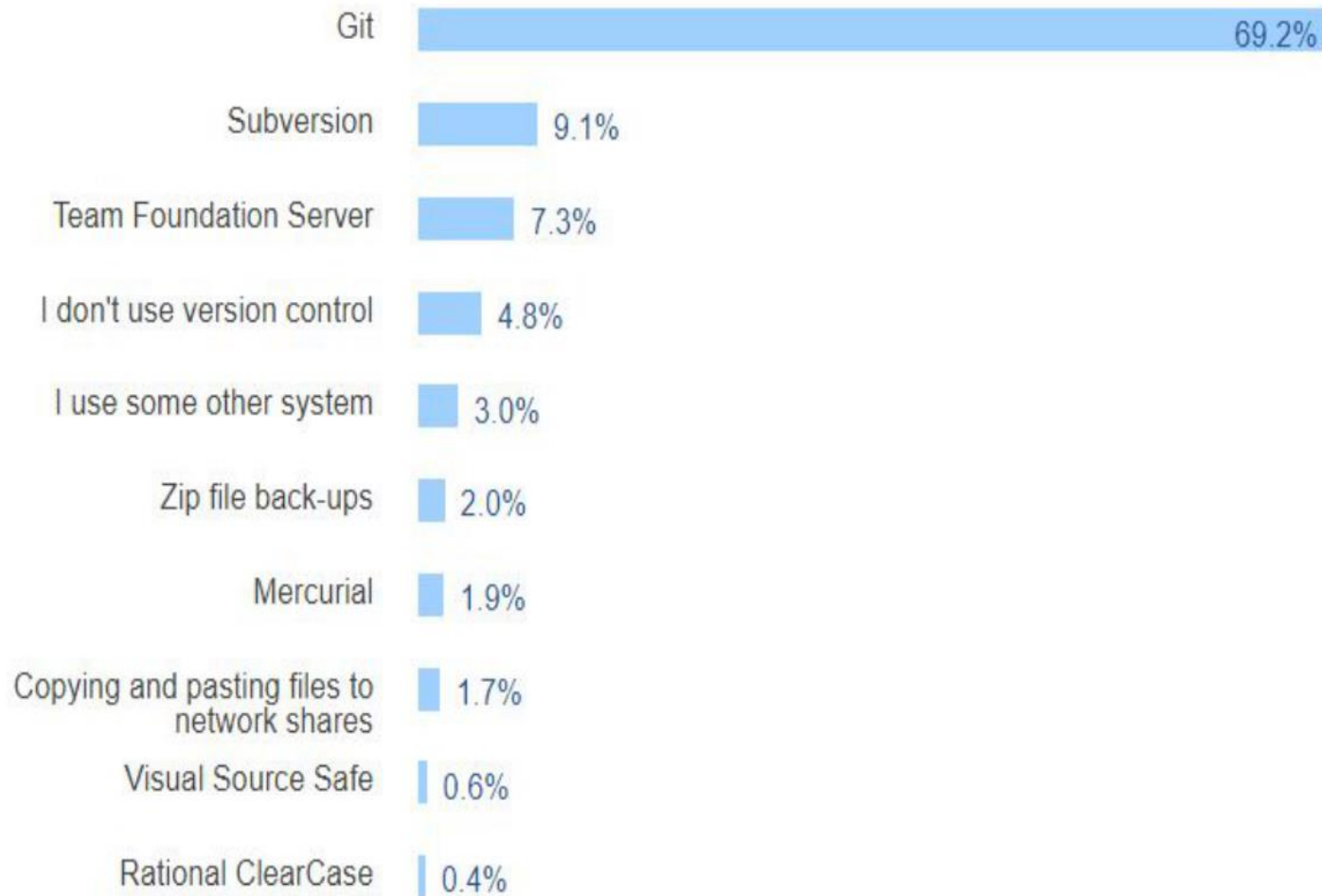


GitHub:

- GitHub is an online interface for Git. Git is software used locally on your computer to record changes. GitHub is a host for your files and the records of the changes made.
- Think of it as being similar to **DropBox** - the files are on your computer, but they are also hosted online and are accessible from any computer. GitHub has the added benefit of interfacing with Git to keep track of all of your file versions and changes.
- You need to create an account on GitHub: <https://github.com/>



Popular Version Control Software



Version Control Vocabulary

Clone: Making a copy of an existing Git repository. If you have just been brought on to a project that has been tracked with version control, you would *clone* the repository to get access to and create a local version of all of the repository's files *and all of the tracked changes*.

Fork: A personal copy of a repository that you have taken from another person. If somebody is working on a cool project and you want to play around with it, you can fork their repository and then when you make changes, the edits are logged on *your* repository, not theirs.



Version Control Vocabulary

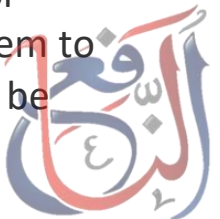
Repository: Equivalent to the project's folder/directory - all of your version controlled files (and the recorded changes) are located in a repository. This is often shortened to **repo**. Repositories are what are hosted on GitHub and through this interface you can either keep your repositories private and share them with select collaborators, or you can make them public - anybody can see your files and their history.

Commit: To commit is to save your edits and the changes made. A commit is like a snapshot of your files: Git compares the previous version of all of your files in the repo to the current version and identifies those that have changed since then. Those that have not changed, it maintains that previously stored file, untouched. Those that have changed, it compares the files, logs the changes and uploads the new version of your file. We'll touch on this in the next section, but when you commit a file, typically you accompany that file change with a little note about what you changed and why.

When we talk about version control systems, commits are at the heart of them. If you find a mistake, you revert your files to a previous *commit*. If you want to see what has changed in a file over time, you compare the *commits* and look at the messages to see why and who.

Push: Updating the repository with your edits. Since Git involves making changes locally, you need to be able to share your changes with the common, online repository. Pushing is sending those committed changes to that repository, so now everybody has access to your edits.

Pull: Updating your local version of the repository to the current version, since others may have edited in the meanwhile. Because the shared repository is hosted online and any of your collaborators (or even yourself on a different computer!) could have made changes to the files and then pushed them to the shared repository, you are behind the times! The files you have locally on *your* computer may be outdated, so you pull to check if you are up to date with the main repository.



Important to Remember

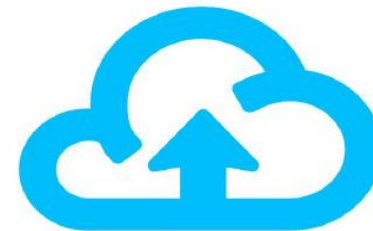
Repository, AKA Repo



Commit



Push



Pull



Version Control Vocabulary

Staging: The act of preparing a file for a commit. For example, if since your last commit you have edited three files for completely different reasons, you don't want to commit all of the changes in one go; your message on why you are making the commit and what has changed will be complicated since three files have been changed for different reasons. So instead, you can stage just one of the files and prepare it for committing. Once you've committed that file, you can stage the second file and commit it. And so on. Staging allows you to separate out file changes into separate commits. Very helpful!

To summarize these commonly used terms so far and to test whether you've got the hang of this, files are hosted in a **repository** that is shared online with collaborators. You **pull** the repository's contents so that you have a local copy of the files that you can edit. Once you are happy with your changes to a file, you **stage** the file and then **commit** it. You **push** this commit to the shared repository. This uploads your new file and all of the changes and is accompanied by a message explaining what changed, why and by whom.



Version Control Vocabulary

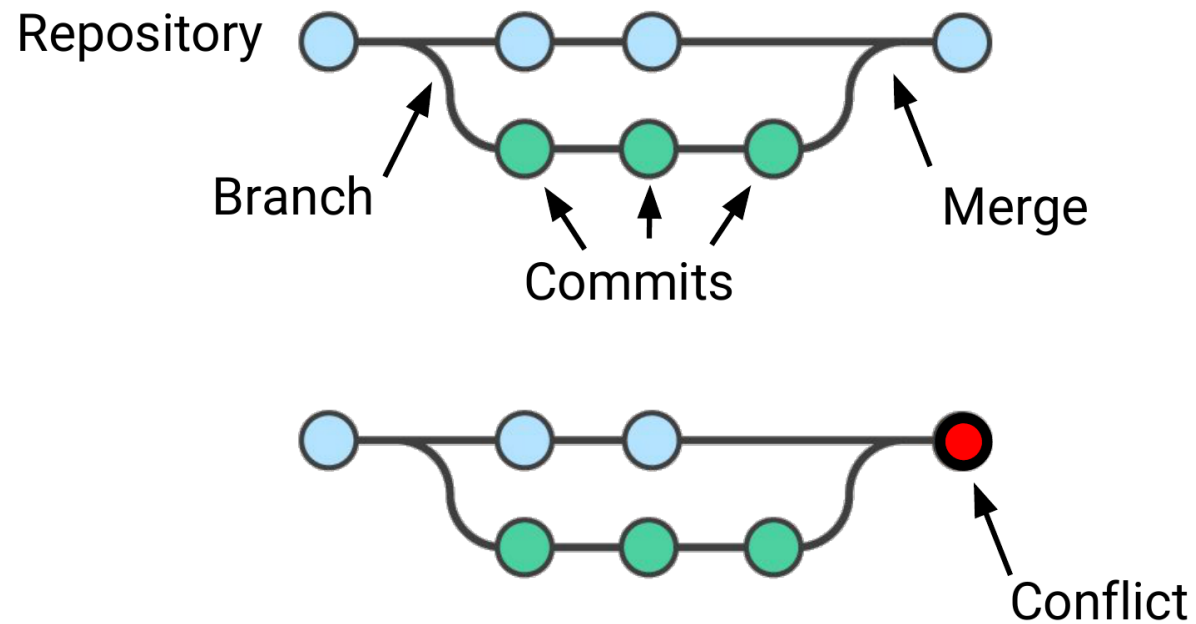
Branch: When the same file has two simultaneous copies. When you are working locally and editing a file, you have created a branch where your edits are not shared with the main repository (yet) - so there are two versions of the file: the version that everybody has access to on the repository and your local edited version of the file. Until you push your changes and merge them back into the main repository, you are working on a branch. Following a branch point, the version history splits into two and tracks the independent changes made to both the original file in the repository that others may be editing, and tracking your changes on your branch, and then merges the files together.

Merge: Independent edits of the same file are incorporated into a single, unified file. Independent edits are identified by Git and are brought together into a single file, with both sets of edits incorporated. But, you can see a potential problem here - if both people made an edit to the same sentence that precludes one of the edits from being possible, we have a problem! Git recognizes this disparity (**conflict**) and asks for user assistance in picking which edit to keep.

Conflict: When multiple people make changes to the same file and Git is unable to merge the edits. You are presented with the option to manually try and merge the edits or to keep one edit over the other.



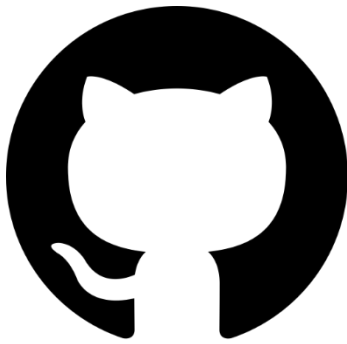
Version Control Vocabulary



Adapted from: <https://www.atlassian.com/git/tutorials/using-branches/git-merge>

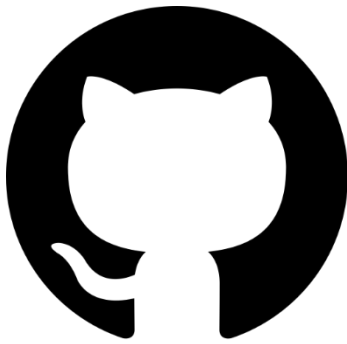
Connecting GitHub with RStudio via Git

- Follow instructions provided in “Github and Git” and “Linking Git” documents
- By the end of this exercise, you must have:
 - Github account,
 - 1 Repository,
 - Repository should be linked with RStudio
 - Have at least 2 commits in repository



Connecting GitHub with RStudio via Git

Congrats! You have successfully created your first repo and connected it to your RStudio!





Practice Exercises





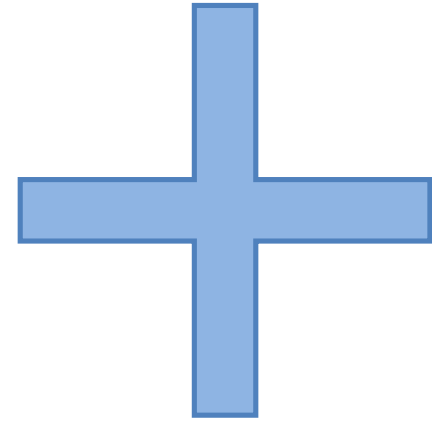
Data Types and Basic Operations (Part I)

Section 5



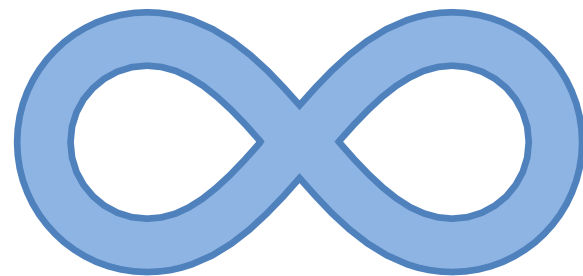
Objects

- R has five basic or “atomic” classes of objects:
 1. character
 2. numeric (real numbers)
 3. integer
 4. complex
 5. logical (True/False)
- The most basic object is a vector
- A vector can only contain objects of the same class
- BUT: The one exception is a ***list***, which is represented as a vector but can contain objects of different classes (indeed, that’s usually why we use them)
- Empty vectors can be created with the `vector()` function.



Numbers

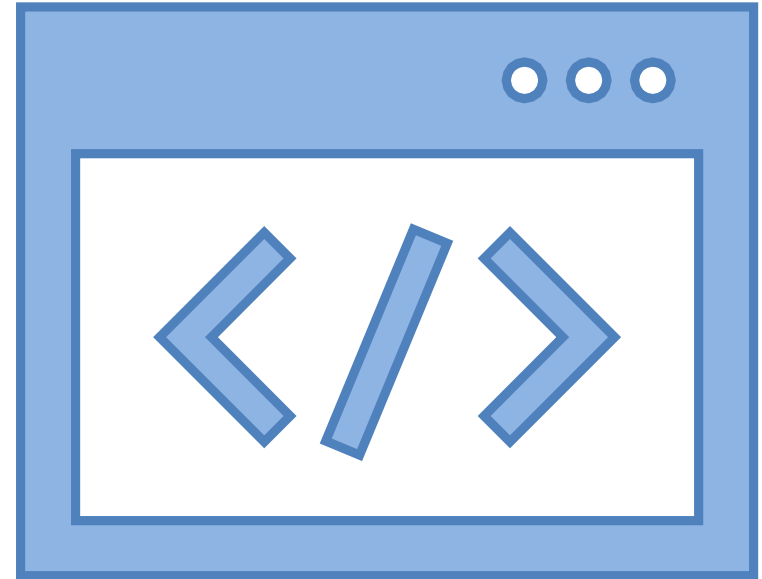
- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the `L` suffix
- Ex: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g. `1/0`; `Inf` can be used in ordinary calculations; e.g. `1/Inf` is `0`.
- The value `NaN` represents an undefined value (“not a number”); e.g. `0/0`; `NaN` can also be thought of as a missing value (more on that later).



Attributes

R objects can have attributes:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/metadata



Attributes of an object can be accessed using the `attributes()` function.

Entering Input

At the R prompt we type expressions. `<-` symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
> x <- ## Incomplete expression
```

The `#` character indicates a comment. Anything to the right of the `#` (including the `#` itself) is ignored.



Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5      ## nothing printed
> x           ## auto-printing occurs
[1] 5
> print(x)    ## explicit printing
[1] 5
```

The `[1]` indicates that `x` is a vector and `5` is the first element.



Printing

```
> x <- 1:20  
> x  
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  
[16] 16 17 18 19 20
```

The : operator is used to create integer sequences.



Creating Vectors

The `c ()` function can be used to create vectors of objects.

```
> x <- c(0.5, 0.6)      ## numeric
> x <- c(TRUE, FALSE)   ## logical
> x <- c(T, F)          ## logical
> x <- c("a", "b", "c") ## character
> x <- 9:29              ## integer
> x <- c(1+0i, 2+4i)     ## complex
```

Using the `vector ()` function

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```



Mixing Objects

What about the following?

```
> y <- c(1.7, "a")      ## character  
> y <- c(TRUE, 2)       ## numeric  
> y <- c("a", TRUE)     ## character
```

When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.



Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1]
"integer"
>
as.numeric(x)
[1] 0 1 2 3 4
5 6
> as.logical(x)
[1] FALSE      TRUE TRUE TRUE TRUE TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Explicit Coercion

Nonsensical coercion results in NAs.

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```



Matrices

Matrices are vectors with a ***dimension*** attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
```

```
> m
```

```
      [,1] [,2] [,3]  
[1,]   NA   NA   NA  
[2,]   NA   NA   NA
```

```
> dim(m)
```

```
[1] 2 3
```

```
> attributes(m)
```

```
$dim [1]
```

```
2 3
```



Matrices (Contd.)

Matrices are constructed **column-wise**, so entries can be thought of starting in the “upper left” corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```



Matrices (Contd.)

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10

> dim(m) <- c(2, 5)

> m
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```



cbind-ing and rbind-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
      x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
      [,1] [,2] [,3]
x         1    2    3
y        10   11   12
```



Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x

[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```



Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a ***label***.

- Factors are treated specially by modelling functions like `lm()` and `glm()`.
- Using factors with labels is ***better*** than using integers because factors are self-describing; having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.



Factors (Contd.)

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes
no Levels: no yes

> table(x)
x
no yes
  2   3

> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no" "yes"
```



Factors (Contd.)

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

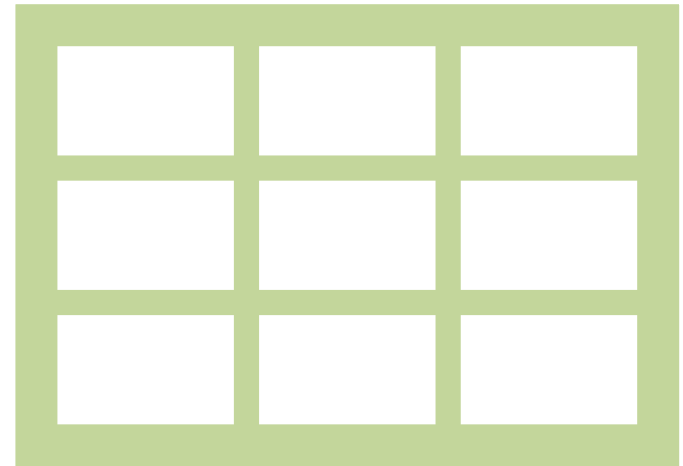
```
> x <- factor(c("yes", "yes", "no", "yes",  
               "no"), levels = c("yes", "no"))  
  
> x  
[1] yes yes no yes no  
Levels: yes no
```



Missing Values

Missing values are denoted by NA or NaN for undefined mathematical operations.

- `is.na()` is used to test objects if they are NA
- `is.nan()` is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true



Missing Values (Contd.)

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE

> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```



Data Frames

Data frames are used to store tabular data

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



Data Frames (Contd.)

```
> x <- data.frame(day = 1:4, rain = c(T, T, F, F))
> x
  day  rain
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE

> nrow(x)
[1] 4

> ncol(x)
[1] 2
```



Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x) NULL
> names(x) <- c("Col_A", " Col_B", " Col_C")
> x
Col_A  Col_B  Col_C
  1      2      3
> names(x)
[1] "Col_A"  " Col_B" " Col_C"
```



Names (Contd.)

Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1]
1

$b
[1]
2

$c
[1]
3
```



Names (Contd.)

And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```



Summary

Data Types

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frame
- names





Getting Help in R

Section 6





Simplest way to get help

- To click on the Help button on the toolbar of the Rgui window.
- Alternatively, if you are connected to the internet, you can type CRAN in Google and search for the help you need at CRAN.
- If you know the name of the function you want help with, you just type a question mark ? at the command line prompt followed by the name of the function. So to get help on `read.table`, just type:

```
?read.table
```

- Sometimes you cannot remember the precise name of the function, but you know the subject on which you want help (e.g. data input in this case). Use the `help.search` function (without a question mark) with your query in double quotes like this:

```
help.search("data input")
```





Getting Help...

Other useful functions are

- `find`

The `find` function tells you what package something is in:

```
> find(lowess)
[1] "package:stats"
```

- `apropos`

`apropos` returns a character vector giving the names of all objects in the search list that match your (potentially partial) enquiry:

```
> apropos(lm)

[1] ". __C__anova.glm" ". __C__anova.glm.null" ". __C__glm"
[4] ". __C__glm.null" ". __C__lm" ". __C__mlm"
[7] "anova.glm" "anova.glm.list" "anova.lm"
[10] "anova.lm.list" "anova.mlm" "anova.list.lm"
[13] "contr.helmert" "glm" "glm.control"
[16] "glm.fit" "glm.fit.null" "hatvalues.lm"
[19] "KalmanForecast" "KalmanLike" "KalmanRun"
[22] "KalmanSmooth" "lm" "lm.fit"
[25] "lm.fit.null" "lm.influence" "lm.wfit"
[28] "lm.wfit.null" "model.frame.glm" "model.frame.lm"
[31] "model.matrix.lm" "nlm" "nlminb"
[34] "plot.lm" "plot.mlm" "predict.glm"
```



Example: Error Messages

```
> library(datasets)
> data(airquality)
> cor(airquality)
Error in cor(airquality) : missing observations in cov/cor
```



Google is Your Friend...



missing observations in cov/cor



Web

Images

News

Videos

Maps

More

Filters Settings

Error in cor.default(x1, x2) : missing observations in cov/cor

<https://r.789695.n4.nabble.com/Error-in-cor-default-x1-x2-missing-observations-in-cov...>

What happens when you type "cor" at the R prompt? Perhaps your calling of the cor function is not calling the cor function in the stats package? Ken Spriggs wrote:

r - Dealing with missing values for correlations ...

<https://stackoverflow.com/questions/7445639>

Consider this example, merely an extension of the author's demo: if A and B agree on all observations, but A has 99 observations and B only has 97, is it really absurd that pairwise-cor gives a correlation of 1, and would you conclude with the author that a correlation of NA is more reasonable? – David Klotz Mar 23 '18 at 2:00

cor function | R Documentation

<https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/cor>

Correlation, Variance and Covariance (Matrices) var, cov and cor compute the variance of x and the covariance or correlation of x and y if these are vectors. If x and y are matrices then the covariances (or correlations) between the columns of x and the columns of y are computed.. cov2cor scales a covariance matrix into the corresponding correlation matrix efficiently.

R: Correlation, Variance and Covariance (Matrices)

<https://stat.ethz.ch/R-manual/R-devel/library/stats/html/cor.html>

Correlation, Variance and Covariance (Matrices) Description. var, cov and cor compute the variance of x and the covariance or correlation of x and y if these are vectors. If x and y are matrices then the covariances (or correlations) between the columns of x and the columns of y are computed.. cov2cor scales a covariance matrix into the corresponding correlation matrix efficiently.

Cor function | R Documentation

<https://www.rdocumentation.org/packages/DescTools/versions/0.99.34/topics/Cor>

Arguments x. a numeric vector, matrix or data frame. y. NULL (default) or a vector, matrix or data frame with compatible dimensions to x. The default is equivalent to y = x (but more efficient). use. an optional character string





Data Types and Basic Operations (Part II)

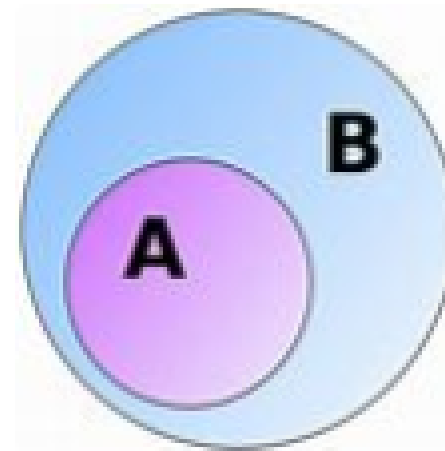
Section 7



Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- `[]` always returns an object of the same class as the original; can be used to select more than one element (there is one exception).
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- `$` is used to extract elements of a list or data frame by name; semantics are similar to that of `[[`.



Subsetting (Contd.)

```
> x <- c("a", "b", "c", "c", "d", "a")
```

```
> x[1]
```

```
[1] "a"
```

```
> x[2]
```

```
[1] "b"
```

```
> x[1:4]
```

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

```
> x[x > "a"]
```

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

```
> u <- x > "a"
```

```
> u
```

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

```
> x[u]
```

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



Subsetting a Matrix

Matrices can be subsetting in the usual way with (i, j) type indices.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```



Subsetting a Matrix (Contd.)

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1,2]
[1] 3
> x[1, 2, drop=FALSE]
      [,1]
[1,]      3
```



Subsetting a Matrix (Contd.)

Similarly, sub-setting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> x[1, , drop=FALSE]
      [,1] [,2] [,3]
[1,]    1    3    5
>
```



Subsetting Lists

```
> x <- list(weekday = 1:4, rain_prob = 0.6)
```

```
> x[1]
```

```
$ weekday
```

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

```
> x[[1]]
```

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

```
> x$rain_prob
```

```
[1] 0.6
```

```
> x[["rain_prob"]]
```

```
[1] 0.6
```

```
> x["rain_prob"]
```

```
$rain_prob [1] 0.6
```



Subsetting Lists (Contd.)

```
> x <- list(weekday = 1:4, rain_prob = 0.6, item = "umbrella")
> x[c(1, 3)]
$weekday
[1] 1 2 3 4

$item
[1] "umbrella"
```



Subsetting Lists (Contd.)

The `[]` operator can be used with **computed** indices; `$` can only be used with literal names.

```
> x <- list(weekday = 1:4, rain_prob = 0.6, item = "umbrella")

> name <- "weekday"

> x[[name]]                                # computed index for 'weekday'
[1] 1 2 3 4

> x$name                                    # element 'name' doesn't exist!
NULL

> x$weekday                                # element 'weekday' does exist
[1] 1 2 3 4
```



Subsetting Nested Elements of a List

The `[[` can take an integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))  
> x[[c(1, 3)]]  
[1] 14  
  
> x[[1]][[3]]  
[1] 14  
  
> x[[c(2, 1)]]  
[1] 3.14
```



Removing NA Values

A common task is to remove missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4
5
```



Removing NA Values (Contd.)

What if there are multiple things and you want to take the subset with no missing values?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSE          TRUE FALSE TRUE
> x[good]
[1] 1 2
4 5
> y[good]
[1] "a" "b" "d" "f"
```



Removing NA Values (Contd.)

```
> airquality[1:6, ]
```

	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

```
> good <- complete.cases(airquality)
```

```
> airquality[good, ][1:6, ]
```

	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
7	23	299	8.6	65	5	7





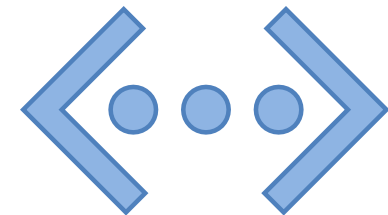
Control Structures

Section 8



Control Structures

- Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are:
 - `if, else`: testing a condition
 - `for`: execute a loop a fixed number of times
 - `while`: execute a loop while a condition is true
 - `repeat`: execute an infinite loop
 - `break`: break the execution of a loop
 - `next`: skip an iteration of a loop
 - `return`: exit a function
- Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.



Control Structures: **if**

```
if(<condition>) {  
    # do something  
} else {  
    # do something else  
}  
  
if(<condition1>) {  
    # do something  
} else if(<condition2>) {  
    # do something different  
} else {  
    # do something different  
}
```



Control Structures: `if` (Contd.)

This is a valid `if/else` structure.

```
x <- 4
y <- 0
if(x > 3) {
    y <- 10
} else {
    y <- 0
}
```

So is this one.

```
y <- if(x > 3) {
    10
} else {
    0
}
```

Control Structures: `if` (Contd.)

Of course, the `else` clause is not necessary.

```
if(<condition1>) {  
  
}  
  
if(<condition2>) {  
  
}
```



Control Structures: **for**

`for` loops take an iterator variable and assign it successive values from a sequence or vector. `for` loops are most used for iterating over the elements of an object (list, vector, etc.).

```
for(i in 1:10) {  
    print(i)  
}
```

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.



Control Structures: `for` (Contd.)

These three loops have the same behavior.

```
x <- c("a", "b", "c", "d")
```

```
for(i in 1:4) {  
  print(x[i])  
}
```

```
for(i in seq_along(x)) {  
  print(x[i])  
}
```

```
for(letter in x) {  
  print(letter)  
}
```

```
for(i in 1:4) print(x[i])
```



Control Structures: Nested for Loops

for loops can be nested.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    print(x[i, j])
  }
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.



Control Structures: **while**

while loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0
while(count < 10) {
  print(count)
  count <- count + 1
}
```

while loops can potentially result in infinite loops if not written properly. Use with care!



Control Structures: **while** (Contd.)

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)

  if(coin == 1) { ## random walk
    z <- z + 1
  } else {
    z <- z - 1
  }
}
```

Conditions are always evaluated from left to right.



Control Structures

Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if they are theoretically correct.
- Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the `*apply` functions are more useful.





Functions

Section 9



Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

```
f <- function(<arguments>) {  
  ## Do something interesting  
}
```

Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.



Function Arguments

Functions have ***named arguments*** which potentially have ***default*** values.

- The ***formal arguments*** are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be ***missing*** or might have default values



Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.



Argument Matching (Contd.)

You can mix positional matching with matching by name. When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights,
         na.action, method = "qr", model =
         TRUE, x = FALSE,
         y = FALSE, qr = TRUE, singular.ok =
         TRUE, contrasts = NULL, offset,
         ...)
```

The following two calls are equivalent.

```
mydata <- data.frame(x=rnorm(100),
                    y=rnorm(100))
lm(y ~ x, mydata, model = FALSE)

lm(data=mydata, y~x, model = FALSE, 1:100)
```



Argument Matching (Contd.)

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list.
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).



Argument Matching (Contd.)

Function arguments can also be ***partially*** matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument.
2. Check for a partial match.
3. Check for a positional match.



Defining a Function

```
f <- function(a, b = 1, c = 2, d = NULL) {  
  
}
```

In addition to not specifying a default value, you can also set an argument value to NULL.



Lazy Evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
> f <- function(a, b)
  { a^2
}

> f(2)
[1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`.



Lazy Evaluation (Contd.)

```
> f <- function(a, b)
  { print(a)
    print(b)
  }
```

```
> f(45)
[1] 45
```

```
Error: argument "b" is missing, with no default
```

Notice that “45” got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.





Coding Standards for R

Section 10



Coding Standards for R

- Always use text/script files or text editor.
- Indent your code.
- Limit the width of your code (80 columns?)
- Limit the length of individual functions



Indenting

- Indenting improves readability.
- Fixing line length (80 columns) prevents lots of nesting and very long functions.
- *Suggested: Indents of 4 spaces at minimum; 8 spaces ideal.*





Vectorized Operations

Section 11



Vectorized Operations

Many operations in R are **vectorized** making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1] 7 9 11 13
> x > 2
[1] FALSE FALSE TRUE TRUE
> x >= 2
[1] FALSE TRUE TRUE TRUE
> y == 8
[1] FALSE FALSE TRUE FALSE
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```



Vectorized Matrix Operations

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y                                ## element-wise multiplication [,1] [,2]
[1,] 10  30
[2,] 20  40
> x / y
      [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y                                ## true matrix multiplication
      [,1] [,2]
[1,]  40   40
[2,]  60   60
```





Dates and Times in R

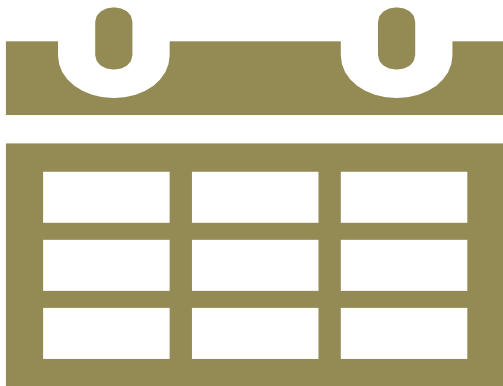
Section 12



Dates and Times in R

R has developed a special representation of dates and times:

- Dates are represented by the `Date` class.
- Times are represented by the `POSIXct` or the `POSIXlt` class.
- Dates are stored internally as the number of days since 1970-01-01.
- Times are stored internally as the number of seconds since 1970-01-01.



Dates in R

Dates are represented by the Date class and can be coerced from a character string using the `as.Date()` function.

```
> as.Date("1970-01-01")  
[1] "1970-01-01"  
  
> as.POSIXct("1970-01-01")  
[1] "1970-01-01 EET"  
  
> as.POSIXlt("1970-01-01")  
[1] "1970-01-01 EET"
```



Times in R



Times are represented using the `POSIXct` or the `POSIXlt` class

- `POSIXct` is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame.
- `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month.

There are a few generic functions that work on dates and times

- `weekdays`: give the day of the week.
- `months`: give the month name.
- `quarters`: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”).



Times in R (Contd.)

Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.

```
> x <- Sys.time()
> x
[1] "2020-04-19 19:13:53 PKT"
> p <- as.POSIXlt(x)
> p
[1] "2020-04-19 19:13:53 PKT"
> names(unclass(p))
[1] "sec"      "min"      "hour"     "mday"     "mon"      "year"     "yday"
[9] "isdst"    "zone"     "gmttoff"
> p$sec
[1] 53.78624
> p$mon
[1] 3
> p$yday
[1] 0
```

To remove temporarily the effects of class, use the function `unclass()`



Times in R (Contd.)

You can also use the `POSIXct` format.

```
> x <- Sys.time()
> x      ## Already in 'POSIXct' format
[1] " 2020-04-19 19:13:53 PKT"

> x$sec
## Error: $ operator is invalid for atomic vectors

p <- as.POSIXlt(x)
p$sec
## [1] 14.37
```



Times in R (Contd.)

Finally, there is the `strptime` function in case your dates are written in a different format

```
datestring <- c("January 10, 2012 10:40",  
"December 9, 2011 9:10")  
  
x <- strptime(datestring, "%B %d, %Y %H:%M")  
x  
## [1] "2012-01-10 10:40:00" "2011-12-09 09:10:00"  
class(x)  
## [1] "POSIXlt" "POSIXt"
```

I can ***never*** remember the formatting strings. Check `?strptime` for details.



Operations on Dates and Times

You can use mathematical operations on dates and times. Well, really just addition (+) and subtraction (-). You can do comparisons too (i.e. ==, <=).

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
x-y
## Warning: Incompatible methods ("-.Date",
## "-.POSIXt") for "-"
## Error: non-numeric argument to binary operator

> class(x)
[1] "Date"
> class(y)
[1] "POSIXlt" "POSIXt"

x <- as.POSIXlt(x)
x-y
## Time difference of 356.3 days
```



Operations on Dates and Times (Contd.)

Even keeps track of leap years, leap seconds, daylight savings, and time zones.

```
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
x-y

## Example of Time difference of 2 days
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y-x
## Time difference of 1 hours
```



Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations.
- Dates use the `Date` class.
- Times use the `POSIXct` and `POSIXlt` class.
- Character strings can be coerced to Date/Time classes using the `strptime` function or `as.Date`, `as.POSIXlt`, or `as.POSIXct`.





Loop Functions

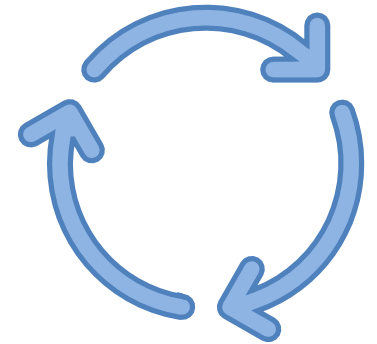
Section 13



Looping on the Command Line

Writing `for`, `while` loops is useful when programming but not particularly easy when working interactively on the command line for data sets. There are some functions which implement looping to make life easier.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of `lapply`
- An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.



lapply

Apply loop on List (lapply)

`lapply` takes three arguments:

- (1) a list `X`;
- (2) a function (or the name of a function) `FUN`;
- (3) other arguments via its `...` argument.

If `X` is not a list, it will be coerced to a list using `as.list`.

```
## function (X, FUN, ...)  
## {  
##   FUN <- match.fun(FUN)  
##   if (!is.vector(X) || is.object(X))  
##     X <- as.list(X)  
##   .Internal(lapply(X, FUN))  
## }  
## <bytecode: 0x7ff7a1951c00>  
## <environment: namespace:base>
```

The actual looping is done internally in C code.



lapply (Contd.)

`lapply` always returns a list, regardless of the class of the input.

```
> x <- list(a = 1:5, b = rnorm(10))  
>
```

```
$a  
[1] 3  
  
$b  
[1] 0.4671
```



lapply (Contd.)

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1),  
            d = rnorm(100, 5))  
> lapply(x, mean)
```

```
## $a  
## [1] 2.5  
##  
## $b  
## [1] 0.5261  
##  
## $c  
## [1] 1.421  
##  
## $d  
## [1] 4.927
```



lapply (Contd.)

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.3402839

[[2]]
[1] 0.5263668 0.5649934

[[3]]
[1] 0.7999538 0.4609495 0.3148338

[[4]]
[1] 0.2077871 0.5926260 0.6416160 0.9814817
```



lapply (Contd.)

```
> x <- 1:4  
> lapply(x, runif, min = 0, max = 10)  
[[1]]  
[1] 3.302142  
  
[[2]]  
[1] 6.848960 7.195282  
  
[[3]]  
[1] 3.5031416 0.8465707 9.7421014  
  
[[4]]  
[1] 1.195114 3.594027 2.930794 2.766946
```



sapply

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned.



sapply (Contd.)

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))  
> lapply(x, mean)  
$a  
[1] 2.5  
  
$b  
[1] 0.06082667  
  
$c  
[1] 1.467083  
  
$d  
[1] 5.074749
```



sapply (Contd.)

```
> sapply(x, mean)
      a      b      c      d
2.50000000 0.06082667 1.46708277 5.07474950

> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```



apply

`apply` is used to evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!



apply (Contd.)

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- X is an array.
- MARGIN is an integer vector indicating which margins should be “retained”.
- FUN is a function to be applied.
- ... is for other arguments to be passed to FUN .



apply (Contd.)

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
```

[1]	0.04868268	0.35743615	-0.09104379
[4]	-0.05381370	-0.16552070	-0.18192493
[7]	0.10285727	0.36519270	0.14898850
[10]	0.26767260		

```
> apply(x, 1, sum)
```

[1]	-1.94843314	2.60601195	1.51772391
[4]	-2.80386816	3.73728682	-1.69371360
[7]	0.02359932	3.91874808	-2.39902859
[10]	0.48685925	-1.77576824	-3.34016277
[13]	4.04101009	0.46515429	1.83687755
[16]	4.36744690	2.21993789	2.60983764
[19]	-1.48607630	3.58709251	

col/row sum and mean

For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

The shortcut functions are ***much*** faster, but you won't notice unless you're using a large matrix.



Other Ways to Apply

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
```

	[,1]	[,2]	[,3]	[,4]
25%	-0.3304284	-0.99812467	-0.9186279	-0.49711686
75%	0.9258157	0.07065724	0.3050407	-0.06585436
	[,5]	[,6]	[,7]	[,8]
25%	-0.05999553	-0.6588380	-0.653250	0.01749997
75%	0.52928743	0.3727449	1.255089	0.72318419
	[,9]	[,10]	[,11]	[,12]
25%	-1.2467955	-0.8378429	-1.0488430	-0.7054902
75%	0.3352377	0.7297176	0.3113434	0.4581150
	[,13]	[,14]	[,15]	[,16]
25%	-0.1895108	-0.5729407	-0.5968578	-0.9517069
75%	0.5326299	0.5064267	0.4933852	0.8868922
	[,17]	[,18]	[,19]	[,20]



tapply

`tapply` is used to apply a function over subsets of a vector. I don't know why it's called `tapply`.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- `X` is a vector.
- `INDEX` is a factor or a list of factors (or else they are coerced to factors).
- `FUN` is a function to be applied.
- `...` contains other arguments to be passed to `FUN`.
- `simplify`, should we simplify the result?



tapply (Contd.)

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
      1      2      3
0.1144464 0.5163468 1.2463678
```



tapply (Contd.)

Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1144464

$`2`
[1] 0.5163468

$`3`
[1] 1.246368
```



tapply (Contd.)

Find group ranges.

```
> tapply(x, f, range)
$`1`
[1] -1.097309  2.694970

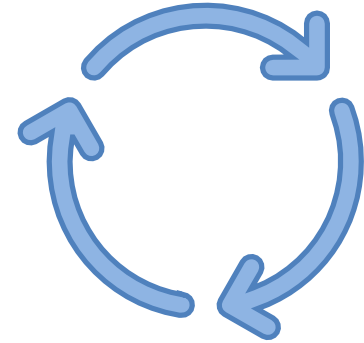
$`2`
[1] 0.09479023 0.79107293

$`3`
[1] 0.4717443 2.5887025
```



Loop - *apply functions

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector





Data Split

Section 14



split

`split` takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
> str(split)
function (x, f, drop = FALSE, ...)
```

- `x` is a vector (or list) or data frame.
- `f` is a factor (or coerced to one) or a list of factors.
- `drop` indicates whether empty factors levels should be dropped.



split (Contd.)

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))  
> f <- gl(3, 10)  
> split(x, f)
```

\$`1`

[1]	-0.8493038	-0.5699717	-0.8385255	-0.8842019
[5]	0.2849881	0.9383361	-1.0973089	2.6949703
[9]	1.5976789	-0.1321970		

\$`2`

[1]	0.09479023	0.79107293	0.45857419	0.74849293
[5]	0.34936491	0.35842084	0.78541705	0.57732081
[9]	0.46817559	0.53183823		

\$`3`

[1]	0.6795651	0.9293171	1.0318103	0.4717443
[5]	2.5887025	1.5975774	1.3246333	1.4372701



split (Contd.)

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.1144464

$`2`
[1] 0.5163468

$`3`
[1] 1.246368
```



Splitting a Data Frame

```
> library(datasets)
> 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



Splitting a Data Frame (Contd.)

```
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$`5`
```

Ozone	Solar.R	Wind
NA	NA	11.62258

```
$`6`
```

Ozone	Solar.R	Wind
NA	190.16667	10.26667

```
$`7`
```

Ozone	Solar.R	Wind
NA	216.483871	8.941935



Splitting a Data Frame (Contd.)

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

	5	6	7	8	9
Ozone	NA	NA	NA	NA	NA
Solar.R	NA	190.16667	216.483871	NA	167.4333
Wind	11.62258	10.26667	8.941935	8.793548	10.1800

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],  
                                na.rm = TRUE))
```

	5	6	7	8	9
Ozone	181.29630	29.44444	59.115385	59.961538	31.44828
Solar.R	23.61538	190.16667	216.483871	171.857143	167.43333
Wind	11.62258	10.26667	8.941935	8.793548	10.18000





Recap



In this course you have learned:

- Overview and Features of R Programming
- Installation & Configuration of R, RStudio & Notebook for R
- Version Control and Github
- Data Types and Basic Operations I and II
- Getting Help in R and Coding Standards
- Control Structures & Functions
- Vectorized Operations & Date and Time
- Loop Functions & Data Split



جزاك الله

To ask questions, Join the Al Nafi Official Group

<https://www.facebook.com/groups/alnafi/>

(This group is only for members to ask questions)

