# Data Management and Digital Archaeology

Andreas Angourakis Tim Klingenberg

29 October, 2024

# Table of contents

# Course overview

040468 Datenmanagement und digitale Archäologie (ÜB, unterrichtet in Englisch) Übungen (3 CP)

**Time slot** / ***Zeitfenster***: Fr 14-16 Uhr c.t.

**Place** / ***Ort***: Raum 2

**Course instructors** / ***Kursleiter***: Andreas Angourakis / Tim Klingenberg

**Support** / ***Unterstützung***: Thomas Rose

## Course summary/*Kurszusammenfassung*:

The "Data Management and Digital Archaeology" MA course offers a comprehensive overview of data management and digital tools essential for archaeological research under the open science paradigm. It begins with foundational sessions on data management and the use of Git and GitHub for version control, helping students organize, maintain, and share their research data and analyses.

The course then delves into programming with R, covering basic R programming skills and best practices, followed by advanced topics like count data analysis, seriation techniques, and compositional data analysis tailored for archaeology. We then explore essential concepts in database management, offering two sessions to provide introductory and more advanced knowledge necessary for handling archaeological databases. The course concludes with a brief, but practical introduction to Geographic Information Systems (GIS), introducing students to GIS software and techniques used to analyse spatial data in archaeological research.

By the end, students will be equipped with enough digital skills to effectively manage, analyse, and interpret archaeological data under an open science framework while also being prepared to continue their own learning journey in this ever-evolving field.

## Course schedule/*Kursplan*:

| | Date | Topic |
|---|---|---|
| **Getting started** | | |
| 1 | 2024-10-18 | Data and research data management |
| 2 | 2024-10-25 | Git and GitHub |
| **Programming in R** | | |
| 3 | 2024-11-08 | Introduction to R |
| 4 | 2024-11-15 | Best practices in programming |
| **Data Science in R** | | |
| 5 | 2024-11-22 | Data Science Workflow |
| 6 | 2024-11-29 | Count data and seriation |
| 7 | 2024-12-06 | Compositional data |
| **Databases** | | |
| 8 | 2024-12-13 | Databases (I) |
| 9 | 2024-12-20 | Databases (II) |
| **GIS** | | |
| 10 | 2025-01-10 | GIS (I) |
| 11 | 2025-01-17 | GIS (II) |

## Evaluation / *Kursbewertung*

(Attendance and final examination)

## Acknowledgements

The conception of the course structure, as well as the short summaries, exercises, and images shown in each chapter, greatly benefited from Large Language Models used as companion writer and programmer. As such, we own greatly to the current richness of reference information freely available on Internet.

The models and services used are:

- ChatGPT (GPT-4o) by OpenAI for brainstorming, text and code writing suggestions, collection and articulation of references.
- WebChatGPT, a free browser extension that enhances ChatGPT by providing Internet access directly within the chat interface, used to aid Internet search.
- Leonardo.ai (user tokens) for generating purely aesthetic visual assets.

# Part I

# Getting started

# 1 Data and research data management

## 1.1 Introduction to Research Data

### 1.1.1 What is data? What is it for?

Data refers to factual information, often quantitative or qualitative, used as a basis for reasoning, discussion, or calculation. It is the foundational analysis element and supports decision-making across diverse fields. In research, data helps to generate new insights, verify hypotheses, and contribute to the overall body of knowledge in a specific area.

### 1.1.2 Archaeological Data: Particularities

In archaeology, data encompasses various types, such as field notes, artefacts, images, geospatial coordinates, etc. By systematically collecting, organizing, and analysing this data, researchers can reconstruct past human behaviours, understand environmental contexts, and explore cultural practices.

Archaeological data is unique due to its diverse formats and the complexity of its collection. It often includes material remains like pottery, bones, tools, and structures, as well as environmental data like pollen samples and soil types. Since archaeological data often comes from excavation sites, it is usually non-renewable; once excavated, a site cannot be restored to its original state.

Archaeologists rely heavily on careful documentation to preserve as much information as possible for future study. Furthermore, the context in which artefacts are found is crucial, as it helps interpret their use, significance, and the broader cultural setting.

### 1.1.3 Open Science

Open Science promotes transparency and collaboration in research by making methodologies, data, and findings accessible to the public and other researchers. In archaeology, this can involve sharing excavation reports, datasets, and analysis methods to facilitate broader understanding and scrutiny of findings.

### 1.1.4 FAIR Principles

FAIR stands for Findable, Accessible, Interoperable, and Reusable ("What Is FAIR?" n.d.) and applies to data, *metadata*, and supporting infrastructure to ensure data is systematically organized and readily available for ongoing use and analysis.

> **i** What is *metadata*?
>
> Metadata refers to data that provides information about other data rather than the content of the data itself. It describes various attributes such as the data's structure, format, location, and context ("Metadata" 2024; Editor n.d.). Examples include creation dates, file sizes, authorship details, and keywords, which help organize, understand, and retrieve the associated data ("What Is Metadata and How Does It Work?" n.d.).
>
> *Example:*
>
> - **Data**: identification number, site of origin, and dates of items of a museum exhibition.
> - **Metadata**: title and description of this data, author(s), date of last update, criteria for fixing site of origin, method(s) for establishing dates.
>
> Notice that, in this case, we could also choose to specify the information about the dating method in the dataset, as long as we want or can specify this information for each item independently.

A short breakdown of the concepts behind FAIR:

- **Findable**: Data should be easy to locate for humans and machines. This includes having:
  - A unique and persistent identifier.
  - Descriptive metadata.
  - Metadata containing data identifiers.
  - Registration in searchable resources.
- **Accessible**: Data should be easy to access once found. This includes:
  - Retrieval via a standardized, open protocol.
  - Support for authentication and authorization when necessary.
  - Availability of metadata even if data itself is no longer available.
- **Interoperable**: Data should integrate with other data and tools. This requires:
  - Using a standardized, accessible language for data representation.

  - Using FAIR-aligned vocabularies.
  - Including references to related data.

- **Reusable**: Data should be well-described for reuse and replication in various contexts. This involves:

  - Detailed, relevant metadata.
  - Clear data usage licensing.
  - Documentation of data provenance.
  - Conformance to community standards.

These principles aim to enhance the usefulness of digital assets by ensuring they can be easily located, understood, and utilized by others (Lamprecht et al. 2020; Wilkinson et al. 2016). Applying FAIR principles in archaeology involves creating well-documented, datasets that other researchers can readily access openly or through transparent authentication procedures, in case of sensitive data (Hiebel et al. 2021; Lien-Talks 2024; Nicholson et al. 2023).

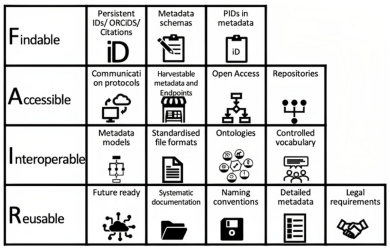


Figure 1.1: A summary breakdown of the FAIR data principles (reproduction from Lien-Talks (2024), Figure 1)

### 1.1.5 CARE Principles

The CARE Principles for Indigenous Data Governance were defined more recently (Carroll et al. 2020) and emphasize ethical and responsible use of data:

- **Collective Benefit**: Data should benefit Indigenous communities collectively.

- **Authority to Control**: Indigenous communities should have control over how their data is collected, used, and shared.

- **Responsibility**: Data practices must uphold the well-being and interests of Indigenous peoples.

- **Ethics**: Data use must be guided by ethical considerations, respecting cultural protocols and privacy.

These principles aim to protect Indigenous rights to self-determination and ensure data contributes positively to community development and innovation. Researchers involved in archaeology should remain sensitive to the concerns raised by CARE, especially when conceptualising a new research project and later, when publishing data and research outputs.

Applying the CARE principles in archaeological research will strongly depend on whether an indigenous group (self-identified and/or legally recognised) is directly involved as researchers or research enablers (e.g., creators of material culture, subjects to surveys and interviews, consulting experts, etc). In countries with a recent colonial past, this criteria tends to be clear for all parts involved. However, the concept of indigenous is hardly consensual when speaking about cultural groups not self-identified in oposition to a colonising cultural group. Initiatives, such as the Traditional Knowledge Labels by Local Contexts, continue to be developed to help to clarify this and other points.

> **See also**
>
> - "CARE Principles for Indigenous Data Governance" (2024)
> - "CARE Principles for Indigenous Data Governance" (n.d.)
> - "CARE Principles for Indigenous Data Governance | ARDC" (2022)

### 1.1.6 Reproducibility

Reproducibility refers to the ability to replicate a study's results using the original author's assets or following their methodology. It ensures that findings can be independently verified under similar conditions, reinforcing the reliability of scientific research (National Academies of Sciences et al. 2019).

In scientific research, reproducibility is crucial for confirming the validity of experimental findings and building upon existing knowledge. It enhances transparency and trustworthiness in scientific practices, promoting better peer review and collaboration "GRN · German Reproducibility Network" (n.d.).

Most archaeological research does not necessarily involve controlled experiments, and archaeological surveys and excavations are destructive and thus unrepeatable. However, the reproducibility of data collection, processing, and analysis is not a trivial concern.

Making research reproducible must be considered a spectrum of practices in which researchers should strive to improve despite the challenges (Marwick 2017).

### 1.1.7 Open Source

Open Source refers to software and tools that are freely available for anyone to use, modify, and distribute ("Open Source" 2024; "The Open Source Definition" n.d.).

Key criteria include:

- **Access to Source Code**: Users can access the software's source code.
- **Free Redistribution**: The software can be freely redistributed.
- **Modifications and Derived Works**: Users can modify the software and create derived works based on it.
- **Integrity of the Author's Source Code**: License terms ensure modifications do not restrict access to the original code.
- **No Discrimination Against Persons or Groups**: The license must not discriminate against any person or group.
- **No Discrimination Against Fields of Endeavor**: The license must apply to all fields of use.
- **Distribution of License**: The rights attached to the software must apply to all who receive it without needing to acquire additional licenses.

For archaeologists, open-source tools can offer affordable solutions for data analysis, visualization, and management, promoting a more inclusive research environment. Additionally, there is a growing community of archaeological software engineers who identify entirely with open-source principles (Batist and Roe 2024; *Open Source Archaeology: Ethics and Practice* 2015).

### 1.1.8 The lay of the land: Organizations, tools, data repositories and where to find them

Several organizations and platforms support Open Science and the use of FAIR and Open Source principles, providing archaeologists with access to valuable tools and datasets.

Here is a list of those that will be most useful during this course:

- Organizations:

- Wikimedia Foundation: A non-profit organization focused on promoting the growth, development, and dissemination of free, multilingual content. It supports projects like Wikipedia, which provide free access to knowledge globally.

- OpenAIRE: OpenAIRE AMKE is a non-profit organization with a mission to promote open scholarship and improve discoverability, accessibility, shareability, reusability, reproducibility, and monitoring of data-driven research results, globally (Iatropoulou n.d.).
- Go FAIR Initiative: Provides guidelines on implementing FAIR principles, with resources tailored for researchers in various fields, including archaeology ("FAIR Principles" n.d.).
- ARIADNE: A research infrastructure that integrates existing archaeological datasets across Europe ("Ariadne Research Infrastructure" n.d.).

- CIDOC-CRM: a theoretical and practical tool for information integration in the field of cultural heritage ("Home | CIDOC CRM" n.d.).

- The Turing Way: collaborative writing of an online handbook (Community 2022).
- CAA: Computer Applications and Quantitative Methods in Archaeology (CAA) is an international organisation bringing together archaeologists, mathematicians, and computer scientists. CAA counts with National Chapters, including one for Germany.
- CAA Special Interest Group on Scientific Scripting Languages in Archaeology: group within the CAA that focuses on the application of Scripting Languages in archaeological research ("Home – CAA/SSLA" n.d.).
- Coalition for Archaeological Synthesis: an alliance of Partner organizations and individual Associates who are committed to fostering synthesis in archaeology to expand knowledge and benefit society ("Coalition for Archaeological Synthesis" 2024).

- NFDI4Objects - Research Data Infrastructure for the Material Remains of Human History: NFDI4Objects is aimed at researchers, practitioners, and students whose interests focus on the material heritage of around three million years of human and environmental history and address the challenges of modern research data infrastructures (NFDI4Objects 2024).

- Data Repositories and Data Management Tools:

  - ORCID: ORCID, which stands for Open Researcher and Contributor ID, is a unique alphanumeric code that identifies researchers and contributors in scholarly activities.

  - Zenodo: A repository where researchers can deposit datasets, software, and publications.

  - Open Science Foundation: OSF is a free, open platform to support your research and enable collaboration.
  - Archaeology Data Service (ADS): An archive for archaeological data from the UK, which provides access to various datasets, including excavation reports and geospatial data.
  - the Digital Archaeological Record (tDAR): An archive for archaeological data hosted in the USA, which provides access to various datasets, including excavation reports and geospatial data.
  - Open Context: A repository offering access to archaeological data from various global sources, adhering to FAIR principles [noauthor_open_nodate-1].
  - Hypothes.is: an open-source platform designed to facilitate collaborative annotation of web content. It is used as a browser extension or plug-in.

- Open Source Tools:

  - Markdown: Markdown is a lightweight markup language designed for creating formatted text using a plain-text editor ("Markdown Guide" n.d.).

  - R and Python: Programming languages with extensive libraries for data analysis, which are widely used in archaeology for statistical analysis and modeling "Welcome to Python.org" (2024). See more about these and other programming languages below.

  - RStudio: RStudio IDE is an integrated development environment for R, a programming language for statistical computing and graphics.
  - Quarto: An open-source scientific and technical publishing system.
  - Git: Git is a distributed version control system designed to track changes in source code during software development.
  - GitHub: GitHub is a developer platform that allows developers to create, store, manage, and share their code using Git software and a series of additional automated services.
  - GitLab: GitLab is a developer platform that allows developers to create, store, manage, and share their code using Git software and a series of additional automated services.
  - Zotero: Zotero is a free, easy-to-use tool to help collect, organize, annotate, cite, and share metadata on references.
  - Zotero Style Repository: An open database for citations styles, which can be used in various reference managers and other tools for the formatting of references.
  - QGIS: A free, open-source GIS tool useful for mapping and spatial analysis in archaeology ("Spatial Without Compromise · QGIS Web Site" n.d.).

  - Arches: Arches is an open-source software platform freely available for cultural

heritage organizations to deploy independently to help them manage their cultural heritage data ("Arches Project Open Source Data Management Platform" 2015).
  - OpenAtlas: an open-source database software developed primarily to acquire, edit, and manage research data from various fields of humanities like history, archaeology, and cultural heritage as well as related scientific data ("OpenAtlas" n.d.).

  - open-archaeo: An extensive list of open source archaeological software and resources ("Open-Archaeo" n.d.).

By utilizing these resources, archaeologists can ensure that their research aligns with Open Science, FAIR, and Open Source principles, ultimately enhancing the transparency, accessibility, and longevity of their work.

## 1.2 Introduction to Programming for Research

### 1.2.1 What is Programming?

Programming is designing and implementing instructions a computer can follow to perform specific tasks. It involves writing code in various programming languages that communicate with the computer's hardware and software to solve problems, process data, and automate repetitive tasks. At its core, programming is about creating a sequence of steps, known as algorithms, which help achieve a desired outcome ("What Is Programming? And How To Get Started" 2024).

### 1.2.2 Importance of Learning Programming

For researchers, learning programming offers several significant advantages:

- *Efficiency and Automation*: Programming can help automate data collection, processing, and analysis, saving time and reducing human error. It also enables researchers to handle large datasets and complex calculations with ease.
- *Reproducibility*: Writing scripts to perform analysis allows other researchers to replicate experiments, thus ensuring results can be verified and reproduced, a fundamental aspect of scientific research.
- *Access to Powerful Tools*: With programming skills, researchers can access a wide range of tools for data visualization, statistical analysis, machine learning, and simulation. These tools can enhance the scope and quality of research projects.

### 1.2.3 Overview of Common Programming Languages

Several programming languages are popular in research due to their specific features and libraries("Introduction to Programming Languages" 2018):

- *Python*: Known for its readability and versatility, Python is widely used for data analysis, machine learning, and automation. It has extensive libraries such as *NumPy*, *pandas*, and *matplotlib*, particularly useful for data-intensive research.
- *R*: A language developed explicitly for statistical computing and graphics, R is preferred in data science, bioinformatics, and fields requiring extensive statistical analysis. The Comprehensive R Archive Network (CRAN) offers thousands of packages that can handle various analytical tasks.
- *MATLAB*: Commonly used in engineering and scientific research, MATLAB excels at numerical computing, simulation, and algorithm development. It is prevalent in fields like physics, engineering, and finance. In contrast to R and Python, MATLAB is not open source. A MATLAB license can be obtained through the campus license of the RUB.
- *JavaScript*: A web development language also used in research to develop interactive data visualizations and web-based applications.

### 1.2.4 Concept of Research Software: Tools and Scripts

Research software comprises tools, libraries, and scripts that aid in conducting and managing research activities. It can range from simple data cleaning and preprocessing scripts to complex software packages for statistical analysis and simulation. Although any software used in research could be considered research software, advanced training focuses on programming or scripting skills, not graphical user interface (GUI) operations (e.g., creating a plot in R or Python rather than in Microsoft Excel).

- *Tools*: Research software tools like Jupyter Notebooks, RStudio, PyCharm, Visual Studio Code, etc., offer integrated development environments (IDEs) tailored to a specific range of languages, enhancing productivity and facilitating code sharing and version control.
- *Scripts*: Scripts are text files encoding programs written to perform specific tasks. In research, scripts are often used for data cleaning, model training, and result visualization. Scripts can be used even to generate and format an entire dataset (have a peek on how the course schedule has been built with R code). Researchers can share these scripts to ensure that analyses are reproducible and consistent.

Learning programming for research allows scientists to leverage these tools and scripts effectively, making research more efficient, reproducible, and insightful.

### 1.2.5 Markdown: a language in between

As already mentioned, Markdown is a markup language with minimal syntax, designed for creating formatted text using a plain-text editor ("Markdown Guide" n.d.). Markdown is easily readable and editable for us humans, just like text. However, it is technically a programming language, as it holds machine-readable instructions (i.e., program), which, once compiled (or rendered) by the right software, produces the desired output in the form of a formatted text. An advanced text editor, such as Microsoft Word, allows you to do exactly the same (and more), but only through a sequence of interactions with the graphic user interface (GUI).

Writing in Markdown or "writing Markdown code" is straightforward and will only require some getting used to, mainly if you are accustomed to advanced text editors. For example, instead of marking a text fragment in bold font through the usual steps, we would do it by typing ** (two asterisks) before and after the fragment.

To illustrate this, consider the rendering and source code of a markdown text:

"**Live** *as if* you were to ***die tomorrow***. **Learn** *as if* you were to ***live forever***." - Mahatma Gandhi, Source: BrainyQuote

```
"**Live** *as if* you were to **_die tomorrow_**. **Learn** as if you were to **_live forever
Mahatma Gandhi, Source: [BrainyQuote](https://www.brainyquote.com/citation/quotes/mahatma_ga
```

The simplicity of this language might seem unnecessary or even annoying, but it is actually a trait that opens a wide range of potential applications. Today, markdown is indispensable for all practices related to open science and can offer a good entry point for other, more complex programming languages.

> **i** See also
>
> - Obregon (2024)
> - "What Is Markdown? Syntax, Examples, Usage, Best Practices" (2021)
> - "Functional Documentation with Markdown and Version Control" (2017)
> - "Basic Syntax | Markdown Guide" (n.d.)

### 1.2.6 Where to learn (and keep learning)

- Datacamp ("Learn R, Python & Data Science Online" n.d.)
- Udemy ("Online Courses - Learn Anything, On Your Schedule | Udemy" n.d.)

- Coursera ("Coursera | Degrees, Certificates, & Free Online Courses" n.d.)
- Courses and workshops by higher education institutions (e.g„ RUB Research School)

---

Hands-on Practice

- Task 1: register yourself as a user at:
  - ORCID
  - GitHub
  - Zenodo
  - Zotero (optional)
  - Hypothes.is (optional)
- Task 2: installations:
  - Git. Follow general instructions here (de) or a video step-by-step tutorial like this or this (de), among many available online.
  - GitHub Desktop
  - R
  - RStudio
  - QGIS
- Task 3: explore datasets at Open Context
- Task 4: try writing a short document in Markdown at Markdown Live Preview.

# 2 Git and GitHub

## 2.1 Version-control and Git

### 2.1.1 Version control: general concept and its usefulness

Version control is a system that helps track and manage file changes over time. It's widely used in software development, but its applications extend to any field where file management is essential, including document editing, research, and project management. At its core, version control provides a historical record of changes. It allows users to revert to previous versions, identify when and why changes were made, and work collaboratively without the risk of overwriting each other's work.

There are two main types of version control: centralized and distributed. Centralized version control systems, such as Subversion (SVN), store files in a central repository. Users check out files, make changes, and then commit them back to the central repository. While effective, centralized systems can be vulnerable if the central server fails. Distributed version control systems, like Git, address this by allowing every user to have a complete copy of the repository on their local machine. This setup enhances collaboration and provides redundancy, as users can work offline and synchronize changes with others when connected.

Git icon

### 2.1.2 Benefits of Version Control

- *Collaboration*: Version control systems make collaboration easier and more efficient by allowing multiple users to simultaneously work on the same project. With distributed systems like Git, branches can be created for different features or tasks, and changes can later be seamlessly merged into the main project. This enables teams to work independently and minimize conflicts.
- *Historical Tracking*: Version control systems keep a detailed history of all changes made to the files. This allows users to see who made changes, when, and why. If an issue arises, it's possible to revert to a previous state without losing any data, making debugging easier.

- *Backup and Redundancy*: In distributed systems, each user's local copy is a backup of the entire project. This redundancy reduces the risk of data loss due to server failures or other issues and allows users to work offline and sync changes later.
- *Version Management*: Version control systems assign unique identifiers to each change, per commit and file changed, usually called "Git hash" or "commit hash". A Git hash a 40-character hexadecimal string, such as 2d3acf90f35989df8f262dc50beadc4ee3ae1560, derived from the contents of the commit, including its parent commit(s), timestamp, and author details [REF]. These identifiers allow users to switch between different versions of the project easily. It's also possible to create branches for experimental features and merge them with the main project once they're stable, facilitating smoother integration of new features.
- *Enhanced Workflow*: Many version control systems support automated processes such as Continuous Integration (CI) and Continuous Deployment (CD), which streamline development and testing. These systems can automatically test changes before they are merged, ensuring higher code quality and reducing the risk of introducing bugs.

Overall, version control systems are crucial tools in modern project management and development workflows. They enable collaboration, ensure data integrity, and improve productivity by providing a structured approach to managing changes in any type of project.

Get a short introduction to Git by watching the official Git Documentation videos here.

## 2.2 Git terminology

Here are some essential Git terms to know:

- **Repository**: A storage space for your project files and their history. Repositories can be local (on your computer) or remote (on platforms like GitHub).
- **Initialise**: configure a specific local directory (your "working directory") as a local repository by creating all necessary files for Git to work.
- **Add/Stage**: adds a change in the working directory to the staging area, telling Git to include updates to a particular file in the next commit. However, adding or staging doesn't really affect the repository since changes are not actually recorded until they are committed (see below).
- **Commit**: A snapshot of changes in the repository. Each commit has a unique ID, allowing you to track and revert changes as needed [1].

- **Branch**: A separate line of development. The default branch is usually called `main` or `master`. Branches allow you to work on features independently before merging them into the main project [2].
- **Merge**: The process of integrating changes from one branch into another. Typically, this involves merging a feature branch into the main branch.
- **Pull**: A command that fetches changes from a remote repository and merges them into your local branch, ensuring your local work is up-to-date with the remote [2].
- **Push**: Uploads your commits from the local repository to the remote repository, making your changes available to others.

Understanding these terms is crucial for effective Git usage and collaboration in any project.

See also

- "Git - Gitglossary Documentation" (n.d.)
- "Git Definitions and Terminology Cheat Sheet" (n.d.)

CHECK: Git software installation

To verify if Git is installed on your machine, follow these steps:

1. **Open Command Prompt** (Windows 10 or 11)
   - Press `Win + R`, type `cmd`, and hit Enter.
   - Alternatively, you can search for "Command Prompt" in the Start menu and select it.
2. **Check for Git**
   - In the Command Prompt window, type the following command and press Enter:

     ```
     git --version
     ```

   - If Git is installed, you will see the installed version, e.g., `git version 2.34.1`.
   - If Git is not installed, you will receive an error message or see that the command is unrecognized. You can download the installer from git-scm.com and follow the installation instructions.

## 2.3 GitHub

### 2.3.1 What is GitHub?

GitHub is a cloud-based platform that enables developers to store, manage, and collaborate on code repositories. It builds on Git, a version control system, by adding collaborative features like pull requests, issue tracking, and discussions, which make it easier for teams to work together on software projects.

GitHub also offers hosting for open-source projects, allowing anyone to contribute or review code. With integrations for CI/CD, project management tools, and documentation, GitHub is a popular choice for developers worldwide to manage both personal and professional projects.

GitHub icon

CHECK: GitHub user and GitHub Desktop installation

#### Check GitHub Desktop Installation

To verify that GitHub Desktop is installed:

- On **Windows**: Go to the Start menu, search for "GitHub Desktop," and open the app. If it launches successfully, GitHub Desktop is installed.
- On **macOS**: Use Spotlight Search (`Cmd + Space`), type "GitHub Desktop," and press Enter. If the app opens, it is installed.

If you don't have GitHub Desktop, you can download it from desktop.github.com and follow the installation instructions [1][2].

#### Verify GitHub User

To check if you are signed in as a GitHub user:

- Open **GitHub Desktop**.
- Go to **File > Options** (on Windows) or **GitHub Desktop > Preferences** (on macOS).
- Under the **Accounts** tab, you should see your GitHub username and avatar if you are signed in. If not, you can sign in with your GitHub credentials here.

#### Bookmark your GitHub user profile page

In your Internet browser, make sure that your own GitHub user profile page is saved in Bookmarks for easy access later.

### 2.3.2 GitHub terminology

Understanding the core vocabulary associated with GitHub operations can help users make the most of this platform, especially for collaborative or open-source projects. Here are some key concepts to keep in mind:

- **Forking**: Forking creates a personal copy of someone else's GitHub repository in your account. It allows users to experiment with changes without affecting the original repository, and is often used to contribute to open-source projects. After forking, developers can freely modify their own versions and submit a pull request to propose these changes to the original repository if they have improvements or fixes to offer.
- **Cloning**: Cloning involves creating a local copy of a repository on your machine. By cloning a repository, users can work offline and change files that can later be pushed back to the GitHub repository. This process is essential for local development, allowing users to commit changes and manage their workflow effectively with Git commands.
- **Pull Requests**: A pull request (PR) is a way to propose changes in a repository. After modifying a forked or branched version of a repository, a developer can open a pull request, which initiates a review process. This feature is central to collaboration on GitHub, allowing others to review, discuss, and approve proposed changes before they are merged into the main branch.
- **Branching**: A branch is a parallel version of the repository within the same repository structure. By branching, developers can isolate work on different features or fixes without altering the main project files. For example, many projects have a main or master branch for the official release version, while other branches are used for development or testing. Branches are typically merged into the main branch once they are finalized.
- **Commits and Push**: A commit is a snapshot of changes in the repository. Every commit includes a message describing the changes, and each commit builds upon previous ones, creating a history of the repository's development. Pushing is uploading these commits to GitHub from a local repository. After a series of commits on a local branch, a user can push these changes to the corresponding branch on GitHub to keep the remote repository up-to-date.
- **Gists**: A gist is a simple way to share code snippets or single files. Gists can be public or secret, and they are particularly useful for sharing configuration files or code examples. Users can fork and edit Gists, making them a lightweight collaboration and code-sharing tool.
- **Issues and Discussions**: Issues are GitHub's built-in tracking system for bugs, tasks, and feature requests. They allow users to report problems, suggest new features, and engage in conversations related to the project. Discussions provide a more open forum-style setting for broader conversation, enabling users to share ideas, ask questions, and contribute knowledge that might not directly relate to specific code changes.

See also

- "Cloning and Forking a Repository — Pythia Foundations" (n.d.)
- "GitHub Glossary" (n.d.)
- "How to Get Familiar with Forking & Cloning GitHub Repos" (2023)
- "Difference Between Fork and Clone in GitHub" (2021)

### 2.3.3 Working with GitHub

GitHub offers various workflows to manage repositories. Here are three common methods:

Local with GitHub Desktop

For those who prefer a graphical user interface (GUI):
**Cloning a Repository**

- Open GitHub Desktop.
- Go to File > Clone Repository.
- Select the repository and click "Clone."

**Creating a New Branch**

- Click on the "Current Branch" dropdown.
- Select "New Branch," name it, and click "Create Branch."

**Making Changes**

- Edit files in your editor.

**Committing Changes**

- Return to GitHub Desktop.
- Stage changed files by ticking the boxes.
- Write a summary of changes and click "Commit to new-branch."

**Pushing Changes**

- Click "Push origin" to upload your changes.

Remote with Web Browser

You can also work directly on GitHub.com:
**Forking a Repository**

- Go to the repository page.
- Click on the Fork button in the top-right corner of the page.

- Choose an owner (user or organisation), a name and description for the new fork repository. The default will always be a exact copy of the original repository. Select whether to copy only the main branch. Click "Create fork".

**Cloning a Repository**

- Go to the repository page.

- Click the green "Code" button and continue the cloning process locally, using console commands (copying link) or with GitHub Desktop.

**Creating a New Branch**

- Click the branch dropdown on the main page.

- Type a new branch name and click "Create branch."

**Making Changes**

- Navigate to the file (and branch) you want to edit.

- Click the pencil icon to edit.

- Make your changes and scroll down to the "Commit changes" section.

**Committing Changes**

- Enter a commit message.

- Choose whether to "commit directly to main" or "Commit to a new branch…".

**Pushing Changes**
(No push is needed as changes are automatically saved to GitHub.)

Local with Console Commands (advanced users)

To work with Git via the command line:
**Navigate to the directory to hold the local copy**

```
cd path/to/local/directory
```

**Cloning a Repository**

```
git clone https://github.com/username/repository.git
```

**Creating a New Branch**

```
git checkout -b new-branch
```

**Making Changes** Edit files in your favorite text editor or IDE.
**Committing Changes**

```
git add .
git commit -m "Describe your changes"
```

**Pushing Changes**

```
git push origin new-branch
```

These workflows enable flexibility in how you manage your projects on GitHub.

### 2.3.4 Markdown (GitHub-flavoured)

When Markdown files (.md) are placed in a GitHub repository, they will be automatically rendered within GitHub web interface by default, while the raw code can still be seen and edited in Markdown.

There are some particularities about how Markdown files will be rendered in GitHub through Internet browsers. Consult GitHub Docs for knowing more about them.

### 2.3.5 How to organise repositories

When structuring your repositories, following some common conventions for organizing files in subdirectories is helpful. This makes projects more readable and more accessible for others to navigate. Here are some commonly used subdirectories:

When software development is a significant part:
* `source/` or `src/`: Contains the main source code for the project.
* `documentation/`, `docs/`, or `doc/`: Stores documentation, such as guides or API references.
* `tests/`: Includes test scripts to ensure code functionality.
* `bin/`: Holds executable scripts or binaries.
* `config/`: Contains configuration files, like YAML or JSON.

When rendering a document or a graphical user interface, such as LaTeX documents, websites, web apps, or video games:
* `assets`: to hold files and subdirectories with closed content and functionality files.
* `assets/images/`, `assets/media/`, etc.: Holds all images or other media files generated externally (not by the repository's source code).
* `assets/styles.css` or `assets/css/`: all CSS code for formatting HTML objects.
* `assets/js/`: JavaScript source code enabling interactive functionalities (it would also apply for other programming languages in similar position). Source code of this kind might also be placed inside the source code folder, if present.

These folder structures are conventions and not strict rules. You can adapt or modify them based on your project's needs.

There are several community-based proposals for standards, including tools that can help automate the creation of a new project directory with conventional files. For example, for a typical Data Science project using Python see Cookiecutter Data Science.

> **See also**
>
> - jimmy (2022)
> - Zestyclose-Low-6403 (2023)
> - danijar (2019)
> - Suhail (2024)
> - Cioara (2018)

### 2.3.6 Conventional files

- `README.md`: Provides an overview of the project, including what it does, how to set it up, and how to contribute. A few sections examples are:
  - General description
  - Authors and/or contributors

  - Acknowledgements
  - Funding
  - Installation or use instructions
  - Contributing

- `LICENSE`: Specifies the terms under which the content of the repository can be used, modified, and distributed. There are many licenses, varying in *permissiveness* and *type of content*. Generally, for projects involving both code and other kinds of content, we recommend CC0-1.0 or MIT. See https://choosealicense.com/ and GitHub Docs).
- `CITATION.cff`: human- and machine-readable citation information for software (and datasets). See example here.
- `.gitignore`: Lists files and directories that Git should ignore, such as build outputs and temporary files.
- `CHANGELOG.md`: This file logs a chronological record of all notable changes made to the project, often following conventions like Conventional Commits.
- `references.bib`: a file containing references in BibTex format, which can be cited within the markdown files of the repository.

### 2.3.7 Version Tags and Releases on GitHub

To manage different versions of your project, GitHub allows you to create **tags** and **releases**:

1. **Create a Tag**:
   - Open your repository on GitHub and navigate to the **Releases** section.
   - Click **Draft a new release**.
   - In the **Tag version** field, type a version number (e.g., `v1.0.0`) to create a new tag (see more in the note below).
   - Specify the target branch or commit for this tag.
2. **Create a Release**:
   - After tagging, enter details such as the release **title** and **description**.
   - Optionally, add **release notes** to summarize changes or new features [1].
   - Click **Publish release** to make it public.

Releases are tied to tags and provide a stable reference for each version, making it easy for users to download specific versions of your project [2].

**About versioning**

If unfamiliar with the logic behind versioning, consult the reference to Semantic Versioning, which can also be found on the right of the "Create a new release" page in GitHub. Their summary states:

> Given a version number MAJOR.MINOR.PATCH, increment the:
> 1. MAJOR version when you make incompatible API changes
> 2. MINOR version when you add functionality in a backward compatible manner
> 3. PATCH version when you make backward compatible bug fixes

However, if your repository is not about creating software products and services, we can do well by simply obeying a few general conventions:

- Add a PATCH version **discretionally** when correcting bugs, typos, tuning aesthetics, etc, or *refactoring* code (explained in Chapter 4).
- Add a MINOR version when expanding code functionality or adding new content (text sections, images)
- Add a new PATCH or MINOR version every time the repository reaches a natural stable point (i.e., there are no changes planned any time soon).
- Make sure that every new MAJOR version is released (GitHub) and published (Zenodo, see below).

**See also**

- "Creating GitHub Releases Automatically on Tags" (2024)
- "Automatically Generated Release Notes" (n.d.)

- Signell (2013)

### 2.3.8 Establishing a GitHub-Zenodo Connection

To link your GitHub repository with Zenodo and enable citation via DOI:

1. **Login to Zenodo**: Go to Zenodo and sign in or create an account.
2. **Authorize GitHub Access**:
   - Click on your profile in Zenodo and select **Linked accounts**.
   - Choose **Connect** next to GitHub.

- You will be redirected to GitHub to authorize Zenodo's access. Approve the request to complete the connection.

3. **Select Repository for DOI Generation**:
    - In Zenodo, navigate to **GitHub** in the **Linked Accounts** section.
    - Enable DOI generation for the desired repository. Zenodo will automatically mint DOIs for any new release you publish.

This connection allows you to generate and manage DOIs for GitHub repositories, enhancing your project's citation and research accessibility.

See also

- "Referencing and Citing Content" (n.d.)
- "Zenodo - Research. Shared." (n.d.)
- "Created New Organization in GitHub and Zenodo Did Not Send a Request for Accessing It · Issue #1596 · Zenodo/Zenodo" (n.d.)
- "Module-5-Open-Research-Software-and-Open-Source/Content_development/Task_2.md at Master · OpenScienceMOOC/Module-5-Open-Research-Software-and-Open-Source" (n.d.)
- "Issue a Doi with Zenodo" (n.d.)

---

Hands-on Practice

- Task 1: Create a profile repository with a README file following GitHub Docs: Quickstart for writing on GitHub.
- Task 2: Making others' repositories your own's
    1. Fork the Course Book repository.
    2. Clone your fork to a local directory.
    3. Modify or add a `.qmd` file in your local directory.
    4. Commit to your local repository (one or more times).

5. Push the changed local repository to your remote repository (i.e. your fork).

6. Create a pull request back to the original repository.

- Task 3: Create a personal project repository

1. Create a GitHub repository under your user. Named it `rub-archwiss_` followed by your surname (no special characters). Initialise with the following properties:

a. Public
b. With a README file
c. with .gitignore (template for R)
d. With a license of your choice.

2. Add all other conventional files mentioned above, even if they remain empty for now.
3. Edit your README file and commit/push your changes.

- Task 4:

  - Set up the GitHub-Zenodo connection.

  - Publish your repository.

  - Update README with the new Zenodo DOI (badge).

# Part II

# Programming in R

# 3 Introduction to R

## 3.1 Preparation

Before learning more about R, make sure that everything is set up properly and that you understand the basics in RStudio GUI.

CHECK: R and RStudio installation

To ensure that both R and RStudio have been installed correctly, follow these steps:

1. Launch RStudio by searching for it in your applications menu.

2. When it opens, you should see an interface with multiple panels, including the Console in the left or bottom left panel.

3. In the RStudio Console, you should see details about the R version currently running, which confirms that both R and RStudio are correctly installed and linked. You can always re-print this information by typing the command `version`.

If any of these steps fail, consider reinstalling R and RStudio, ensuring they are compatible with your operating system.

### 3.1.1 RStudio interface (GUI)

The RStudio interface, or Graphical User Interface (GUI), is designed to help you work efficiently with R. By default, it consists of four main panes or panels, each potentially containing multiple tabs:

1. **Console/Terminal/Background Jobs**: This is where you can type and run R commands directly. It displays output from your code and any error messages.

2. **Environment/History/Connections/Build/Git**: In the top-right, the Environment tab shows all active variables and data loaded into your session, while the History tab keeps a log of previously executed commands.

3. **Files/Plots/Packages/Help/...**: The bottom-right panel has several tabs for navigating your files, viewing plots and other graphical outputs, managing installed packages, and accessing R documentation.

4. **Source or Script Editor**: Located in the top-left, here is where you can write, edit, and save R scripts. You can run selected code from this editor directly in the Console. *This panel will be absent whenever there are no script files open in RStudio.*

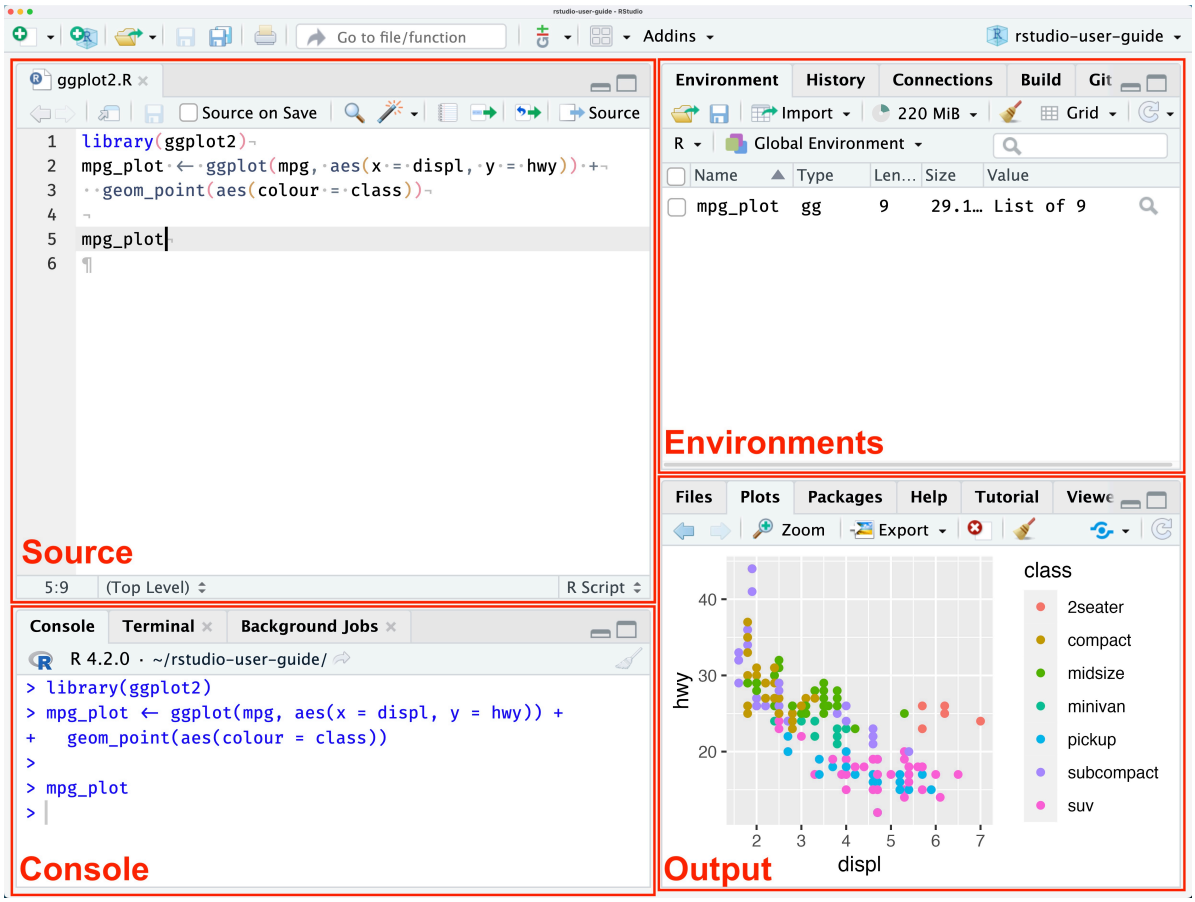


Figure 3.1: RStudio GUI default structure; from "RStudio User Guide - RStudio IDE User Guide" (2024)

Many of the various elements of RStudio GUI are self-explanatory or further explained by pop-up texts and windows. Still, beginners and occasional users can be assured that most elements can be ignored.

### 3.1.2 Global Settings

To customize RStudio, go to Tools > Global Options (or RStudio > Preferences on macOS). Here, you can adjust various settings, including:

- General: Setting the directory path where R has been installed (normally assigned automatically), your default working directory and specify startup options.

- Code: Configuring code formatting, autocompletion, and syntax highlighting settings.
- Appearance: Change the editor theme, font size, and other visual preferences.

- Pane Layout: Changing the default pane structure (not recommended).

### 3.1.3 RStudio Projects

RStudio Projects help organize your work by keeping all related files, scripts, data, and outputs in one place. Each project has its own working directory, which helps to manage dependencies and to maintain reproducibility. Projects are especially useful for keeping different analyses or projects separate from one another.

To create a new project in RStudio, follow these steps:

1. Go to File > New Project.
2. You'll see three options:

- New Directory: Create a project from scratch within a new folder. This is useful when starting a new analysis or project.

- Existing Directory: Convert an existing folder into an RStudio project. Ideal for organizing already-existing files and scripts.

- Version Control: Clone a project from GitHub, GitLab, or other version control systems. This option is helpful when working with collaborative projects or version-tracked repositories.

3. Select the appropriate option based on your needs. For example, if you choose New Directory, you can then select New Project, enter a project name, and specify the location to save it. Alternatively, if you are working with a GitHub repository, you could select Version Control to clone it directly into RStudio, creating a fully synchronized project environment.

4. Click Create Project. RStudio will open a new R session within the project's directory.

### 3.1.4 R Scripts and Rmarkdown notebooks

In RStudio, both R scripts and Rmarkdown notebooks (or rendered notebooks) are used to write and execute R code, but they serve different purposes and have distinct features:

#### Scripts (.R files)

- Plain Text Format: Scripts are simple text files where you can write and save R code. They are best suited for running sequential code and writing reusable functions.
- Execution: You can run code line-by-line or in chunks directly in the Console. Scripts are ideal for production workflows or larger projects where maintaining clear, reproducible code is a priority.
- Comments: You can add comments for documentation, but scripts do not natively support rich formatting like Markdown.

```r
# Calculate the mean of a numeric vector
numbers <- c(1, 2, 3, 4, 5)
mean_value <- mean(numbers)
print(mean_value)
```

```
[1] 3
```

In this script, we define a vector, calculate its mean, and print the result. The focus is on the code itself, without additional formatting or documentation.

#### Notebooks/Rendered notebooks (.Rmd/.qmd files)

- Rich Content: Notebooks allow you to combine code with Markdown for text, making it easy to include formatted text, headings, images, and links alongside your code.
- Interactive Execution: Notebooks support interactive, cell-based execution, similar to Jupyter notebooks. Each code cell outputs results directly below it by default, which is useful for exploratory data analysis and iterative workflows.
- Output Options: Notebooks can be rendered into various formats, such as HTML, PDF, or Word, allowing you to create polished, shareable reports directly from your analysis. It is possible to have code being run and rendered with output directly into a HTML file by choosing `html_notebook`, a type of output that approaches the one of Jupyter notebooks.

````
## Calculate the Mean of a Vector

In this analysis, we calculate the mean of a simple numeric vector.

```{r}
# Define the numeric vector
numbers <- c(1, 2, 3, 4, 5)

# Calculate the mean
mean_value <- mean(numbers)
mean_value
```

The mean of the vector is `r mean_value`.
````

In this notebook, Markdown is used to add a heading and text explanation, while the code chunk calculates and displays the mean. The output is shown directly below the code, creating an interactive, document-like format.

Scripts are optimal for code-centric work with minimal formatting, while notebooks offer a flexible, document-like interface ideal for combining narrative and code in a single file.

### 3.1.5 Creating or editing other files

In addition to R scripts (.R) and RMarkdown notebooks (.Rmd), RStudio supports creating and editing various other file types, making it a versatile environment for different types of content and workflows. To create a new file, go to File > New File and select the desired file type.

Some examples of files you can create and edit in RStudio include:

- Plain Text Files (.txt): Useful for notes, raw data, or configuration files.
- HTML Files (.html): For creating and editing web pages, especially useful for generating custom reports.
- Python Scripts (.py): RStudio supports Python, allowing you to write and execute Python code within the same environment if Python is installed on your computer.
- SQL Files (.sql): You can write and run SQL queries directly in RStudio when working with databases.

To create a file with an extension not listed in RStudio, simply create a .txt file, modify its name and add the desired extension. This flexibility allows you to manage all parts of your project, from data processing to documentation, within RStudio.

> **See also**
>
> For more information about RStudio, consult:
>
> - "RStudio User Guide - RStudio IDE User Guide" (2024)
> - "RStudio IDE :: Cheatsheet" (n.d.)

## 3.2 R syntax and workflow

### 3.2.1 Basic R syntax: variables and data types

In R, variables are created by assigning values using the <- operator. R supports various data types, including:

- Numeric: Numbers, e.g., x <- 10.5
- Integer: Whole numbers, declared with L, e.g., y <- 3L
- Character: Text, surrounded by quotes, e.g., name <- "Alice"
- Logical: Boolean values, TRUE or FALSE, e.g., is_true <- TRUE
- Factor: Categorical data, useful for storing distinct categories, e.g., factor_var <- factor(c("Yes", "No", "Yes"))

Variables store data for manipulation and analysis, forming the building blocks of R programming.

### 3.2.2 Arithmetic operations, logical operations in R.

R supports a range of arithmetic and logical operations:

- Arithmetic Operations: Perform basic math on numbers.
  - Addition: `5 + 3`
  - Subtraction: `5 - 3`
  - Multiplication: `5 * 3`
  - Division: `5 / 3`
  - Exponentiation: `5 ^ 3`
  - Modulus: `5 %% 3` (remainder)
- Logical Operations: Compare values, returning TRUE or FALSE.
  - Equal to: `5 == 3`
  - Not equal to: `5 != 3`
  - Greater than: `5 > 3`

- Less than: `5 < 3`
- Logical **AND**: `TRUE & FALSE`
- Logical **OR**: `TRUE | FALSE`

### 3.2.3 Algorithm structures

R provides basic control structures for implementing algorithms:

- `if` and `else`: Execute code based on a condition.

```
x <- 10
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

```
[1] "x is greater than 5"
```

- `while` loop: Repeat code while a condition is true.

```
count <- 1
while (count <= 5) {
  print(count)
  count <- count + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

- `funtion`: Encapsulate code into reusable blocks.

```
add_numbers <- function(a, b) {
  return(a + b)
}

result <- add_numbers(3, 5)
print(result)
```

```
[1] 8
```

### 3.2.4 Writing and executing R scripts

To create and run an R script:

1. Go to File > New File > R Script in RStudio.
2. Write your code in the editor. For example:

```
# Simple R Script
x <- 5
y <- 10
sum <- x + y
print(sum)
```

3. Highlight the code and press Ctrl + Enter (Windows) or Cmd + Enter (Mac) to execute it in the Console.

### 3.2.5 Using packages

Packages in R are collections of functions, data, and documentation that extend R's capabilities. They allow you to perform specialized tasks without having to write code from scratch. To use a package, you need to first install it and then load it into your R session.

#### Installing and loading packages

To install a package, use the install.packages() function. For example, to install the ggplot2 package:

```
install.packages("ggplot2")
```

Alternatively, you may use the GUI Wizard in Tools > Install Packages..., where an autocomplete feature will help selecting packages exact names.

Once installed, load the package with the `library()` function:

```
library(ggplot2)
```

Now you can access the functions within `ggplot2` and any other loaded package. You only need to install a package once, but you must load it in each new session.

#### Package collection: `tidyverse`

The `tidyverse` is a collection of R packages designed for data science, making data manipulation, visualization, and analysis easier and more intuitive (Wickham et al. 2019; "Tidyverse" n.d.). It includes:

- `ggplot2`: For creating data visualizations using a layered approach.
- `dplyr`: For data manipulation, including filtering, summarizing, and arranging data.
- `tidyr`: For reshaping and tidying data.
- `readr`: For reading data files into R quickly.
- `purrr`: For functional programming, allowing you to work with lists and vectors more effectively.
- `tibble`: A modern version of data frames with enhanced printing and subsetting.

The installation and use of the entire tidyverse works as a single package:

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

The tidyverse packages considerably change the way of working with R. Indeed, tidyverse code is now often used for introducing data science in R, since it is much easier to read and learn for beginners. However, it brings with it dependencies (i.e. other packages) and sometimes hide certain potentials that can only be explored with base R. In this course, we try to keep a balanced perspective by offering a glimpse of more than one R code solutions.

#### Package collection: `tesselle`

The `tesselle` collection is a suite of R packages specifically designed for teaching archaeological data analysis and modelling. These packages provide tools for handling and analysing spatial and temporal patterns in archaeological datasets, making it easier to derive insights from complex data, particularly count data, compositional data and chronological data (Frerebeau 2023; "Tesselle: R Packages & Archaeology" n.d.).

Install the complete suite with:

```
install.packages("tesselle")
```

We will look into more details about this collection in Chapter 6 and Chapter 5.

## 3.3 Basic Data Structures in R

- Vectors, matrices, data frames, lists.
- Basic operations on data structures (indexing, subsetting, adding/removing elements).

## 3.4 Data Manipulation in R

- Importing data: reading data from CSV files, using canonical datasets (`iris`, `archdata::DartPoints`).
- Basic data manipulation using base R (`subset`, `merge`, `apply` functions).
- Introduction to `dplyr` package for data manipulation (filtering, selecting, mutating data).

## 3.5 Data Visualization

- Introduction to plots: histograms, bar plots, scatter plots.
- Creating plots in R with base R graphics.
- Creating multiple plot figures with `layout`.
- Creating plots in `ggplot2`.
- Creating multiple plot figures with `gridExtra::grid.arrange`.
- Base R graphics and `ggplot2`: comparative
- Saving plots: open and closing graphic devices in R.

## 3.6 (EXTRA)Interactive Visualizations

- Introduction to creating interactive visualizations.
- Example: Building an interactive plot using `plotly` and `knitr`.

Hands-on Practice

- Create a new project in RStudio, placing it at the root directory of your own repository (cloned local branch).

- Create a Rmarkdown document or Notebook and give it a title, e.g. “Introduction to R: an exercise”.
- In this document, create a R code chunk to start working and create new chunks and markdown commentaries at every computation step, as you see fit.
- Create a data.frame named “stone_tools_data” directly in R with the following characteristics (based on Carlson 2017, p. 26):
  - Set of six stone tools with inventory number
  - Recording of dimensions (length, breadth, thickness), material type, and whether the material is local or non-local.
  - Data per object:
    - LN15:
      - Length: 18
      - Breadth: 9
      - Thickness: 3
      - Material type: chert
      - Material provenance: local
    - LN17:
      - Length: 14
      - Breadth: 7
      - Thickness: 2
      - Material type: chert
      - Material provenance: local
    - LN18:

- Length: 21
- Breadth: 10
- Thickness: 3
- Material type: obsidian
- Material provenance: local

* LN21:
  - Length: 14
  - Breadth: 7
  - Thickness: 3
  - Material type: chert
  - Material provenance: non-local

* LN23:
  - Length: 17
  - Breadth: 8
  - Thickness: 3
  - Material type: obsidian
  - Material provenance: local

* LN24:
  - Length: 16
  - Breadth: 8
  - Thickness: 2
  - Material type: obsidian

        - Material provenance: non-local

- Check that the data and data types are coherent with the specifications. Save it as a CSV file and load it back as a new R object (e.g. "stone_tools_data2"). Compare.
- Create a plot showing the counts of objects made of chert and obsidian. Save it as a PNG file.
- Create a new variable ("type_and_provenance") that combines type and provenance and create a plot showing the counts in each category. Save it as a PNG file.
- Create a single figure displaying the variable distribution of the three dimensions measured. Save it as both a PNG and a SVG file.
- Create a plot displaying the relationship between length and breadth. Save it as a PNG file.
- Create a plot displaying the relationship between length and breadth, this time marking (point type, colour) objects by their "type_and_provenance". Save it as both a PNG and a EPS file.
- (EXTRA) Create a figure to help explore the question: Do stone tools of different material and provenance tend to be of different size?
- (EXTRA) Duplicate the Rmarkdown document, repeating the steps using `tidyverse` functions.
- Commit all changes and push to the remote using RStudio.
- Q&A and troubleshooting.

# 4 Best practices in programming

## 4.1 Code Organisation

- **Modular Programming**
  - Importance of modularity: breaking down code into functions and modules.
  - Example: Creating in-script custom functions.
  - Example: Creating and importing custom R scripts.
- **Code Structuring**
  - Structuring a data science project: folder organization, separating code, data, and outputs.
  - Example: Setting up a basic project structure in R and RStudio.

## 4.2 Writing Clean and Readable Code

- **Naming Conventions**
  - Using meaningful and consistent names for variables, functions, and files.
  - Example: Best practices in naming conventions in R (tidyverse style guide).
- **Commenting and Documentation**
  - Importance of comments and inline documentation.
  - Example: Writing using `roxygen2` in R for documenting functions.
- **Avoiding Magic Numbers and Hardcoding**

  - Use of constants and configuration files.
  - Example: Using constants in R.

## 4.3 Writing Efficient and Scalable Code

- **Vectorization**
  - Avoiding loops by using vectorized operations for efficiency.
  - Example: Implementing vectorized operations in R (base R, `dplyr`).
- **Memory Management**
  - Managing memory usage and avoiding memory leaks.
  - Example: Best practices for handling large datasets in R (using `data.table`).

## 4.4 (EXTRA)Testing and Validation

- **Writing Unit Tests**
  - Importance of testing: ensuring code correctness.
  - Example: Writing basic unit tests in Python (`unittest` or `pytest`) and R (`testthat`).
- **Data Validation**
  - Validating data inputs and outputs, ensuring data integrity.
  - Example: Implementing data validation checks in data processing scripts.

## 4.5 (EXTRA)Error Handling and Debugging

- **Error Handling Techniques**
  - Using tryCatch in R.

  - Writing meaningful error messages.
  - Example: Implementing error handling in a data processing script.

- **Debugging Tools**
  - Introduction to debugging tools: `browser` in R.
  - Example: Walkthrough of a debugging session in R.

## 4.6 (EXTRA)Code Reusability and Sharing

- **Creating Reusable Code**
  - Writing functions and libraries for reuse across projects.
  - Example: Creating a simple R package.

- **Sharing Code**
  - Sharing code with others: publishing packages, sharing notebooks.
  - Example: Publishing an R package on CRAN/GitHub.

Hands-on Practice

- **Refactoring Code (30min)**
  - First attempt: Refactoring a sample script to follow best practices (clean code, modularity, documentation).
  - Second attempt: try using a Large Language Model (LLM) to refactor.

- **Collaborative Exercise (40min)**
  - Simulating a collaborative workflow with Git: making and reviewing pull requests.
  - Groups of two or three
  - Re-use one of the repositories created in GitHub (Session 2) and populated by R code and output files (Session 3).

- Mutual reviews and change suggestions.
- Discussion, accepting/rejecting changes, and merge decision

- **Open discussion (10min)**
  - Addressing common challenges in applying best practices to real-world projects.

# Part III

# Data Science in R

# 5 Count data and seriation

## 5.1 Introduction to Count Data in Archaeology

- **Understanding Count Data**
  - Definition and significance of count data in archaeological contexts (e.g., artifact counts, feature frequencies).
  - Example: Overview of typical archaeological datasets involving count data.
- **Challenges in Analysing Count Data**
  - Issues with skewness, overdispersion, and zero inflation.
  - Example: Common problems encountered in archaeological count data analysis.

## 5.2 Basic Statistical Methods for Count Data

- **Poisson and Negative Binomial Distributions**
  - Introduction to Poisson distribution and its application to count data.
  - Example: Fitting a Poisson model using `glm` in R.
  - Introduction to the Negative Binomial distribution for overdispersed data.
  - Example: Fitting a Negative Binomial model using `MASS::glm.nb`.
- **Goodness-of-Fit Testing**
  - Assessing the fit of count data models.
  - Example: Performing a chi-square goodness-of-fit test in R.

## 5.3 Introduction to Seriation

- **What is Seriation?**
  - Overview of seriation techniques and their importance in archaeology for ordering artefacts or sites chronologically.
  - Example: Historical applications of seriation in archaeology.
- **Seriation Techniques**
  - Introduction to different seriation methods (e.g., frequency seriation, contextual seriation).
  - Example: Basic seriation using traditional methods.

## 5.4 Using the `tesselle` Package for Seriation

- **Introduction to `tesselle`**
  - Overview of the `tesselle` package and its tools for seriation.
  - Example: Installation and loading of `tesselle`.
- **Practical Seriation in R**
  - Performing seriation.
  - Example: Applying seriation to an archaeological dataset (e.g., pottery styles, stratigraphic data).
- **Visualizing Seriation Results**
  - Creating visual representations of seriation results.
  - Example: Plotting seriation outputs.

Hands-on Practice

- **Case Study: Seriation of Archaeological Artefacts**
  - Step-by-step walkthrough of a seriation analysis using count data.

  - Example: Seriation of pottery fragments or lithic tools using `tesselle`.

- **Q&A and Troubleshooting**
  - Addressing common issues in count data analysis and seriation in archaeological contexts.

# 6 Compositional data

## 6.1 Introduction to Compositional Data in Archaeology

- **Understanding Compositional Data**
  - Definition and examples of compositional data in archaeology (e.g., proportions of different materials, chemical compositions).
  - Example: Typical archaeological datasets that include compositional data.
- **Challenges in Analysing Compositional Data**
  - Issues with relative proportions and the "closed" nature of compositional data.
  - Example: Limitations of traditional statistical methods on compositional data.

## 6.2 Basic Concepts in Compositional Data Analysis

- **Overview of methods in Multivariate statistics**
- **Log-Ratio Transformations**
  - Introduction to log-ratio transformations (CLR, ILR, ALR) for compositional data.
  - Example: Applying a centred log-ratio (CLR) transformation in R (e.g. `nexus::transform_clr`).
- **Visualizing Compositional Data**
  - Techniques for visualizing compositional data (ternary plots, bar charts).
  - Example: Creating a ternary plot using the `isopleuros::ternary_plot` and `ggtern`.

## 6.3 Exploratory Data Analysis

- **Principal Component Analysis (PCA)**
  - Introduction to PCA tailored for compositional data.
  - Example: Performing PCA on compositional data using `tesselle::nexus::pca`.
  - Biplots and screeplots
  - Example: Plotting PCA results as a biplot and add visualisation aids using `tesselle::dimensio` functions.

- **Cluster Analysis**
  - Overview of clustering techniques for exploring groupings in compositional data.
  - Example: Applying hierarchical clustering on transformed compositional data using `tesselle`.
  - `ggplot2`: dendrograms with `ggraph`

- **Correspondence Analysis**
  - Correspondence Analysis for compositional data.
  - Example: Performing Correspondence Analysis using `tesselle::ca`.

Hands-on Practice

- **Case Study: Analysis of Compositional Data from Archaeological Sites**
  - Step-by-step walkthrough of an exploratory analysis of compositional data.
  - Example: Analysing chemical compositions of ceramics or soils using `tesselle`.

- **Q&A and Troubleshooting**
  - Addressing challenges in visualizing and analyzing compositional data in archaeological research.

# Part IV

# Databases

# 7 Databases (I)

## 7.1 Introduction to Databases in Archaeology

- **What is a Database?**
  - Definition and importance of databases in archaeology (data storage, retrieval, and management).
  - Example: Common uses of databases for managing archaeological data (e.g., artifacts, excavation records).
- **Types of Databases**
  - Overview of relational vs. non-relational databases.
  - Example: Advantages of relational databases for structured archaeological data.

## 7.2 Relational Database Concepts

- **Basic Concepts**
  - Introduction to tables, records, fields, primary and foreign keys.
  - Example: Organizing excavation data in relational tables (e.g., site locations, stratigraphy, finds).
- **Normalization**
  - Introduction to database normalization (avoiding redundancy, ensuring data integrity).
  - Example: Structuring artifact data into normalized tables (e.g., artifact type, material, condition).

- **Entity-Relationship (ER) Models**
  - Creating ER models to visualize relationships between archaeological datasets.
  - Example: Designing an ER diagram for an archaeological database (e.g., site, context, finds).

## 7.3 Introduction to SQL (Structured Query Language)

- **Basic SQL Commands**
  - Overview of SQL syntax and common commands (e.g., `CREATE`, `INSERT`, `SELECT`, `UPDATE`).
  - Example: Creating tables for archaeological data and inserting records.
- **Creating a Database with SQL**
  - Step-by-step process of building an archaeological database using SQL.
  - Example: Creating an artifact catalogue with fields such as artifact ID, type, material, and context.
- **Indexing and Keys**
  - Explanation of primary keys, foreign keys, and indexing for optimizing database performance.
  - Example: Defining keys to link excavation sites to finds in different tables.
- **Basic Querying with SQL**
  - Writing basic queries to retrieve data from a database.
  - Example: Using `SELECT` statements to extract information about finds or excavation layers.
- **Filtering and Sorting Data**
  - Using `WHERE`, `ORDER BY`, and `GROUP BY` clauses to filter and sort data.

  - Example: Querying artifacts by material type or sorting excavation records by date.

- **Joining Tables**
  - Using `JOIN` statements to combine related tables (e.g., linking artifact data with stratigraphy).
  - Example: Writing queries to retrieve artifacts based on their excavation context.

## 7.4 Database Tools for Archaeology

- **Accessing Databases from R**
  - Introduction to R packages (`DBI`, `RSQLite`, `RPostgres`) for interacting with databases.
  - Example: Connecting to an SQLite or PostgreSQL database from R for archaeological data analysis.

- **SQLite for small projects**
  - Introduction to SQLite as a lightweight database solution for archaeological projects.
  - Example: Creating a simple SQLite database for site data using R (RSQLite, e.g. https://cran.r-project.org/web/packages/RSQLite/vignettes/RSQLite.html) or Python.

- **PostgreSQL for larger projects**
  - Overview of PostgreSQL for larger archaeological datasets.
  - Example: Setting up a PostgreSQL database in R for managing excavation records.

- **Querying Databases in R**
  - Writing SQL queries in R and retrieving data for further analysis.
  - Example: Querying an excavation database from R and visualizing the results with `ggplot2`.

- **Data Wrangling and Cleaning**
  - Using R to clean and manipulate database data for analysis.
  - Example: Using `dplyr` in R to filter and transform queried data.

Hands-on Practice

- **Building a Small Archaeological Database**
  - Step-by-step walkthrough of creating a database schema for a hypothetical excavation project (using RSQLite).
  - Example: Creating tables for stratigraphy, finds, and context.
- **Inserting and Managing Data**
  - Practical examples of adding and managing archaeological data.
  - Example: Inserting excavation records into the database using SQL.
- **Q&A and Troubleshooting**
  - Addressing challenges in database design and SQL queries.

# 8 Databases (II)

## 8.1 Using GIS Databases for Archaeology

- **Introduction to Spatial Databases**
  - Overview of spatial databases and their use in archaeology (e.g., storing geospatial excavation data).
  - Example: Introduction to PostGIS for spatial data management in PostgreSQL.
- **Linking GIS Data to Databases**
  - Using databases to manage spatial data for archaeological analysis.
  - Example: Storing site coordinates and linking them to excavation records in a spatial database.
- **Querying Spatial Data**
  - Writing spatial queries to retrieve geospatial data from a database.
  - Example: Querying sites within a specific radius or extracting artifact distributions across layers.

Hands-on Practice

- **Querying and Analyzing Archaeological Data**
  - Walkthrough of writing SQL queries to extract meaningful insights from archaeological data.
  - Example: Retrieving and analyzing artifact distributions based on context and stratigraphy.
- **Combining Database and Spatial Data**
  - Practical examples of integrating database queries with GIS data for archaeo-

logical site analysis.

  - Example: Using a database to visualize the spatial distribution of artifacts across an excavation site.

- **Q&A and Troubleshooting**
  - Addressing challenges in querying and analyzing archaeological databases.

# Part V

# GIS

# 9 GIS (I)

(Basic concepts, installation and setting up, loading files)

Cool archive video about ARCInfo (1988): https://youtu.be/7xqNyUOIRCs?si=FulmlUVzaThGE9BU

(Dooley n.d.)

Hands-on Practice

# 10 GIS (II)

(get a map image displaying data from shape and raster files)

> Hands-on Practice

# References


"Arches Project Open Source Data Management Platform." 2015. https://www.archesproject.org/.

"Ariadne Research Infrastructure." n.d. *Ariadne Research Infrastructure*. Accessed October 14, 2024. https://www.ariadne-research-infrastructure.eu/.

"Automatically Generated Release Notes." n.d. *GitHub Docs*. Accessed October 11, 2024. https://docs.github.com/en/repositories/releasing-projects-on-github/automatically-generated-release-notes.

Baker, Monya. 2016. "1,500 Scientists Lift the Lid on Reproducibility." *Nature* 533 (7604): 452–54. https://doi.org/10.1038/533452a.

"Basic Syntax | Markdown Guide." n.d. Accessed October 11, 2024. https://www.markdownguide.org/basic-syntax/.

Batist, Zachary, and Joe Roe. 2024. "Open Archaeology, Open Source? Collaborative Practices in an Emerging Community of Archaeological Software Engineers." *Internet Archaeology*, no. 67 (July). https://doi.org/10.11141/ia.67.13.

"CARE Principles for Indigenous Data Governance." 2024. *Wikipedia*. https://en.wikipedia.org/w/index.php?title=CARE_Principles_for_Indigenous_Data_Governance&oldid=1246386873.

———. n.d. https://static1.squarespace.com/static/5d3799de845604000199cd24/t/5da9f4479ecab221ce848fb2/1571419335217/CARE+Principles_One+Pagers+FINAL_Oct_17_2019.pdf.

"CARE Principles for Indigenous Data Governance | ARDC." 2022. *Https://Ardc.edu.au/*. https://ardc.edu.au/resource/the-care-principles/.

Carroll, Stephanie Russo, Ibrahim Garba, Oscar L. Figueroa-Rodríguez, Jarita Holbrook, Raymond Lovett, Simeon Materechera, Mark Parsons, et al. 2020. "The CARE Principles for Indigenous Data Governance" 19 (1): 43. https://doi.org/10.5334/dsj-2020-043.

Cioara, Andrei. 2018. "How I Organize My GitHub Repositories." *Medium*. https://andreicioara.com/how-i-organize-my-github-repositories-ce877db2e8b6.

"Cloning and Forking a Repository — Pythia Foundations." n.d. Accessed October 28, 2024. https://foundations.projectpythia.org/foundations/github/github-cloning-forking.html.

"Coalition for Archaeological Synthesis." 2024. *CfAS*. https://www.archsynth.org/.

Community, The Turing Way. 2022. "The Turing Way: A Handbook for Reproducible, Ethical and Collaborative Research." Zenodo. https://doi.org/10.5281/ZENODO.3233853.

"Coursera | Degrees, Certificates, & Free Online Courses." n.d. Accessed October 8, 2024. https://www.coursera.org/.

"Created New Organization in GitHub and Zenodo Did Not Send a Request for Accessing It · Issue #1596 · Zenodo/Zenodo." n.d. *GitHub*. Accessed October 11, 2024. https://github.com/zenodo/zenodo/issues/1596.

"Creating GitHub Releases Automatically on Tags." 2024. https://jacobtomlinson.dev/posts/2024/creating-github-releases-automatically-on-tags/.

danijar. 2019. "Can I Arrange Repositories into Folders on Github?" Forum post. *Stack Overflow*. https://stackoverflow.com/q/11852982/6199967.

"Difference Between Fork and Clone in GitHub." 2021. *GeeksforGeeks*. https://www.geeksforgeeks.org/difference-between-fork-and-clone-in-github/.

Dooley, Andy MacLachlan, Adam Dennett and Claire. n.d. *CASA0005 Geographic Information Systems and Science*. Accessed October 9, 2024. https://andrewmaclachlan.github.io/CASA0005repo/index.html.

Editor, CSRC Content. n.d. "Metadata - Glossary | CSRC." Accessed October 14, 2024. https://csrc.nist.gov/glossary/term/metadata.

"FAIR Principles." n.d. *GO FAIR*. Accessed October 7, 2024. https://www.go-fair.org/fair-principles/.

Frerebeau, Nicolas. 2023. *Tesselle: Easily Install and Load 'Tesselle' Packages*. Pessac, France: Université Bordeaux Montaigne. https://doi.org/10.5281/zenodo.6500491.

"Functional Documentation with Markdown and Version Control." 2017. *Leon Hassan*. https://blog.leonhassan.co.uk/functional-documentation-with-markdown-and-version-control/.

"Git - Gitglossary Documentation." n.d. Accessed October 11, 2024. https://git-scm.com/docs/gitglossary.

"Git Definitions and Terminology Cheat Sheet." n.d. Accessed October 11, 2024. https://www.pluralsight.com/resources/blog/cloud/git-terms-explained.

"GitHub Glossary." n.d. *GitHub Docs*. Accessed October 28, 2024. https://docs.github.com/en/get-started/learning-about-github/github-glossary.

"GRN · German Reproducibility Network." n.d. Accessed October 7, 2024. https://reproducibilitynetwork.de/.

Hiebel, Gerald, Gert Goldenberg, Caroline Grutsch, Klaus Hanke, and Markus Staudt. 2021. "FAIR Data for Prehistoric Mining Archaeology." *International Journal on Digital Libraries* 22 (3): 267–77. https://doi.org/10.1007/s00799-020-00282-8.

"Home – CAA/SSLA." n.d. Accessed October 14, 2024. https://sslarch.github.io/.

"Home | CIDOC CRM." n.d. Accessed October 14, 2024. https://www.cidoc-crm.org/.

"How to Get Familiar with Forking & Cloning GitHub Repos." 2023. *DEV Community*. https://dev.to/joshhortt/how-to-get-familiar-with-forking-cloning-github-repos-46nc.

Iatropoulou, Katerina. n.d. "OpenAIRE." *OpenAIRE*. Accessed October 7, 2024. https://www.openaire.eu/.

"Introduction to Programming Languages." 2018. *GeeksforGeeks*. https://www.geeksforgeeks.org/introduction-to-programming-languages/.

"Issue a Doi with Zenodo." n.d. *Github for Collaborative Documentation*. Accessed October 11, 2024. https://cassgvp.github.io/github-for-collaborative-documentation/docs/tut/6-Zenodo-integration.html.

jimmy. 2022. "How to Organize GitHub Repositories." *Backrightup*. https://backrightup.

com/blog/how-to-organize-github-repositories/.

Lamprecht, Anna-Lena, Leyla Garcia, Mateusz Kuzak, Carlos Martinez, Ricardo Arcila, Eva Martin Del Pico, Victoria Dominguez Del Angel, et al. 2020. "Towards FAIR Principles For research software." *Data Science* 3 (1): 37–59. https://doi.org/10.3233/DS-190026.

"Learn R, Python & Data Science Online." n.d. Accessed October 8, 2024. https://www.datacamp.com.

Lien-Talks, Alphaeus. 2024. "How FAIR Is Bioarchaeological Data: With a Particular Emphasis on Making Archaeological Science Data Reusable." *Journal of Computer Applications in Archaeology* 7 (1). https://doi.org/10.5334/jcaa.154.

"Markdown Guide." n.d. Accessed October 7, 2024. https://www.markdownguide.org/.

Marwick, Ben. 2017. "Open Science in Archaeology," January. https://doi.org/10.17605/OSF.IO/3D6XX.

"Metadata." 2024. *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Metadata&oldid=1250210606.

"Module-5-Open-Research-Software-and-Open-Source/Content_development/Task_2.md at Master · OpenScienceMOOC/Module-5-Open-Research-Software-and-Open-Source." n.d. *GitHub*. Accessed October 11, 2024. https://github.com/OpenScienceMOOC/Module-5-Open-Research-Software-and-Open-Source/blob/master/content_development/Task_2.md.

National Academies of Sciences, Engineering, Policy and Global Affairs, Engineering Committee on Science, Board on Research Data Information, Division on Engineering and Physical and Sciences, Committee on Applied and Theoretical Statistics, Board on Mathematical Sciences Analytics, et al. 2019. "Understanding Reproducibility and Replicability." In *Reproducibility and Replicability in Science*. National Academies Press (US). https://www.ncbi.nlm.nih.gov/books/NBK547546/.

NFDI4Objects. 2024. "NFDI4Objects." *NFDI4Objects*. http://195.37.32.58:4000/.

Nicholson, Christopher, Sarah Kansa, Neha Gupta, and Rachel Fernandez. 2023. "Will It Ever Be FAIR?: Making Archaeological Data Findable, Accessible, Interoperable, and Reusable." *Advances in Archaeological Practice* 11 (1): 63–75. https://doi.org/10.1017/aap.2022.40.

Obregon, Alexander. 2024. "What Is Markdown? Uses and Benefits Explained." *Medium*. https://medium.com/@AlexanderObregon/what-is-markdown-uses-and-benefits-explained-947300e1f955.

"Online Courses - Learn Anything, On Your Schedule | Udemy." n.d. Accessed October 8, 2024. https://www.udemy.com/.

"Open Source." 2024. *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Open_source&oldid=1248809486.

*Open Source Archaeology: Ethics and Practice*. 2015. De Gruyter Open Poland. https://doi.org/10.1515/9783110440171.

"Open-Archaeo." n.d. *Title*. Accessed October 14, 2024. https://open-archaeo.info/.

"OpenAtlas." n.d. Accessed October 14, 2024. https://openatlas.eu/.

R Core Team. 2024. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

"Referencing and Citing Content." n.d. *GitHub Docs*. Accessed October 11, 2024. https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content.

"RStudio IDE :: Cheatsheet." n.d. Accessed October 14, 2024. https://rstudio.github.io/cheatsheets/html/rstudio-ide.html?_gl=1*11i4y2y*_ga*MjAyMzI2NjEwMC4xNzIzODI3Mjk2*_ga_2C0WZ1JHG0*MTcyODkxNzc2NS4xMy4xLjE3Mjg5MTkwNTIuMC4wLjA.

"RStudio User Guide - RStudio IDE User Guide." 2024. *RStudio User Guide*. https://docs.posit.co/ide/user/.

Signell, Rich. 2013. "How to Handle Releases of Markdown Document on Github." Forum post. *Stack Overflow*. https://stackoverflow.com/q/19727632/6199967.

"Spatial Without Compromise · QGIS Web Site." n.d. Accessed October 7, 2024. https://qgis.org/.

Suhail, Muhammad Ahmed. 2024. "Structuring and Organizing My Github: A Developer's Guide." *Medium*. https://medium.com/@muhammadahmedsuhail007/structuring-and-organizing-my-github-a-developers-guide-7353610f04fd.

"Tesselle: R Packages & Archaeology." n.d. *Tesselle*. Accessed October 14, 2024. https://www.tesselle.org/.

"The Open Source Definition." n.d. *Open Source Initiative*. Accessed October 14, 2024. https://opensource.org/osd.

"Tidyverse." n.d. Accessed October 14, 2024. https://www.tidyverse.org/.

"Welcome to Python.org." 2024. *Python.org*. https://www.python.org/.

"What Is FAIR?" n.d. Accessed October 14, 2024. https://www.howtofair.dk/what-is-fair/.

"What Is Markdown? Syntax, Examples, Usage, Best Practices." 2021. https://www.knowledgehut.com/blog/web-development/what-is-markdown.

"What Is Metadata and How Does It Work?" n.d. *WhatIs*. Accessed October 14, 2024. https://www.techtarget.com/whatis/definition/metadata.

"What Is Programming? And How To Get Started." 2024. *Coursera*. https://www.coursera.org/articles/what-is-programming.

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. "Welcome to the Tidyverse." *Journal of Open Source Software* 4 (43): 1686. https://doi.org/10.21105/joss.01686.

Wilkinson, Mark D., Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, et al. 2016. "The FAIR Guiding Principles for Scientific Data Management and Stewardship." *Scientific Data* 3 (1): 160018. https://doi.org/10.1038/sdata.2016.18.

"Zenodo - Research. Shared." n.d. Accessed October 11, 2024. https://help.zenodo.org/docs/profile/linking-accounts/.

Zestyclose-Low-6403. 2023. "How to Organize Repos Within an 'Organization'?" Reddit {Post}. *R/Github*. www.reddit.com/r/github/comments/188d324/how_to_organize_repos_within_an_org