Research in Practice:
A Methods Handbook for Research in the Psychological Sciences

Paul D. Kieffaber, Ph.D.

College of William & Mary

Copyright ©2021 Paul D. Kieffaber

PUBLISHED BY PUBLISHER

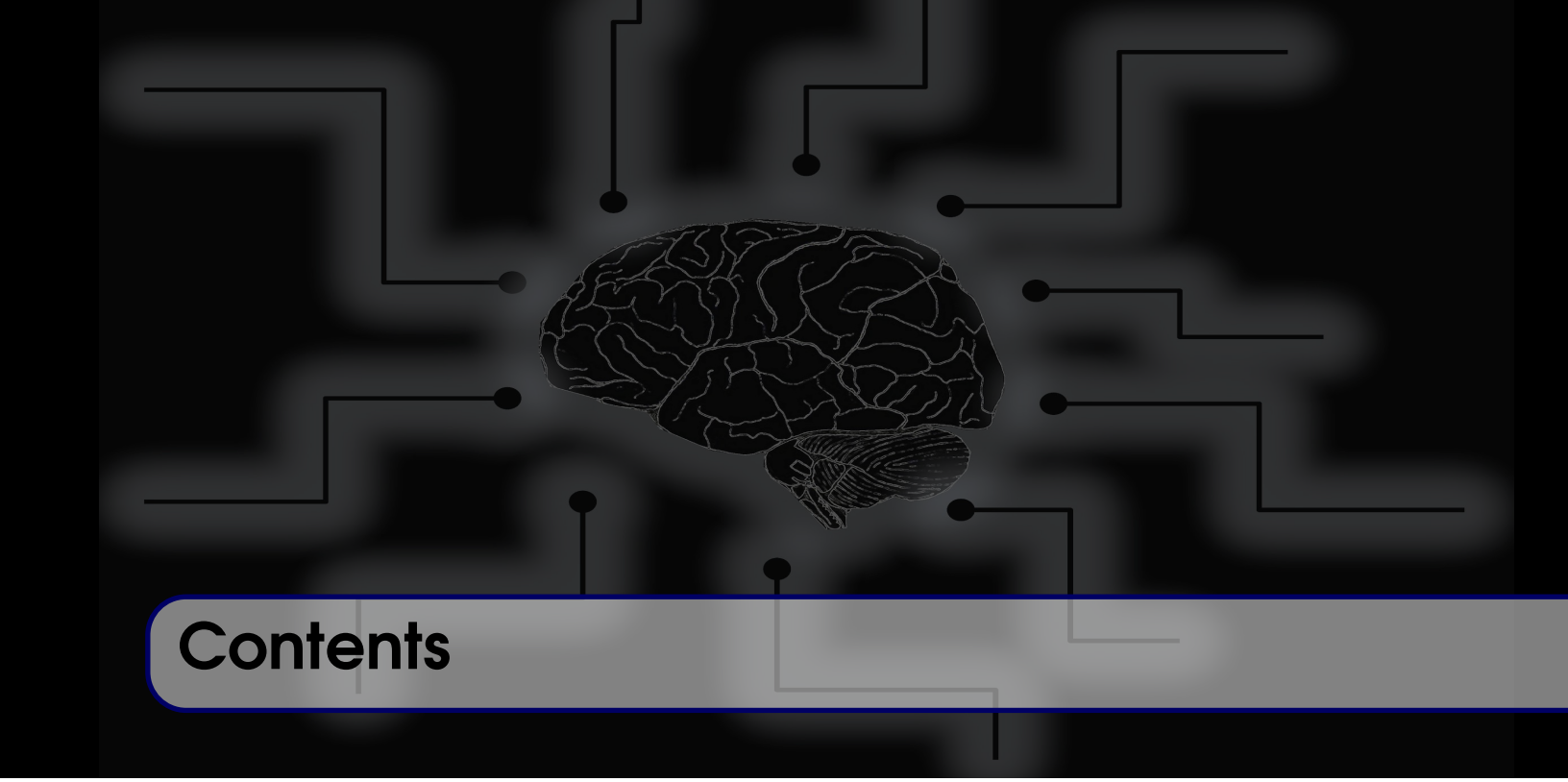
HTTPS://WMPEOPLE.WM.EDU/SITE/PAGE/PDKIEFFABER/HOME

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at `http://creativecommons.org/licenses/by-nc/3.0`. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, March 2019

Author Credits

- Sensation & Perception - Parts of the chapter were adapted by material originally developed with support from the William & Flora Hewlett Foundation, Bill & Melinda Gates Foundation, Michelson 20MM Foundation, Maxfield Foundation, Open Society Foundations, and Rice University. Powered by OpenStax CNX.
- Hello



Contents

PSYC672

1. Reproducible Research with GIT & Git-HUB

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software environment and the complete set of instructions which generated the figures.

— David Donoho

Contents

1.1 What is Reproducible Research?

The quote by David Donoho (Stanford University, 1988) at the top of this chapter is widely cited in discussions of scientific transparency, often paraphrased to emphasize that published articles are merely "advertisements" for the actual scholarship, which lies in the data, the code used to process those data, and the computational methods used to generate the results. In the context of reproducible research, this means that sharing raw data, analysis scripts, and documentation—ideally along with information about software versions and computing environments—is essential for others to be able to verify findings, replicate analyses, or extend the work. Without access to these materials, even well-intentioned scientists cannot fully evaluate the integrity or robustness of the original conclusions. True reproducibility goes beyond publishing results; it requires open and structured sharing of the entire analytic workflow.

Historical Significance

Reproducibility has long been considered a cornerstone of the scientific method, dating back to the Enlightenment era, when thinkers like Francis Bacon and later Karl Popper emphasized the importance of empirical verification. In its most basic form, reproducibility refers to the ability of independent researchers to arrive at the same results using the same data and methodology. This principle ensures that scientific claims are not the result of chance, bias, or unrecognized error. Over time, particularly with the rise of computational science, the concept of reproducibility has evolved to include not only replication of experiments but also transparency in analytical workflows—sharing raw data, statistical code, and computational environments.

Reproducibility is often confused with generalizability, but the concepts are distinct. Generalizibility concerns whether a finding holds true in new settings, populations, or under different conditions. In contrast, reproducibility is narrower: it tests whether the same result can be obtained when the original procedures are followed exactly. A study can be reproducible but not generalizable (e.g., if it was conducted in a very specific sample), or it can be broadly generalizable but not reproducible if the analytic process was not transparent or cannot be followed precisely. Both qualities are important, but without reproducibility, we can't be confident in the validity of the original finding to begin with.

A notable case that highlights the importance of reproducibility and transparency is the 2010 study by Reinhart and Rogoff, which claimed a strong negative relationship between national debt and economic growth. The study was widely cited in economic policy debates around austerity measures. However, when a group of graduate students attempted to replicate the findings, they discovered major coding errors and selective data exclusion in the original Excel spreadsheet **?**. Once corrected, the negative relationship between debt and growth was much weaker than originally reported. This episode not only damaged the authors' reputations but also revealed how the absence of transparent, reproducible methods can lead to widespread misinformation and misguided policy decisions.

Such examples underscore the broader consequences of irreproducible research—not just for scientific integrity, but for public trust, policy, and the allocation of resources. As modern research becomes increasingly complex and data-driven, reproducibility is no longer optional; it is a professional obligation.

Methods to Improve Reproducibility

fill in later

1.2 Introduction to Git

Git is a popular and powerful open-source, version control software. Version control software manages changes in files as they are edited over time and by different users. With Git, users can track changes

made to files and back up old versions of files. When multiple variants of a product need to be managed or different people make changes, Git tracks the differences across branches.

Installing Git

Installing Git on Windows

1. Download Git for Windows
   - Visit the official Git website: https://git-scm.com/
   - Click Download for Windows. This will download an .exe installer.
2. Run the Installer
   - Double-click the downloaded .exe file.
   - When prompted by Windows, allow the installer to make changes.
   - Proceed through the setup wizard. The default options are fine for most users, but pay attention to these key choices:
     (a) Choose "Git from the command line and also from 3rd-party software."
     (b) Select "Use Git Bash only" unless you prefer integrating with Windows Command Prompt.
     (c) Use the default: "Use the OpenSSL library."
     (d) Choose "Checkout Windows-style, commit Unix-style line endings" (default).
   - Click Install, then Finish to complete the setup.
3. Open Git Bash
   - Git Bash is a terminal emulator that gives you a Unix-style shell on Windows. Find it in the Start Menu under "Git" → "Git Bash." Open it to start using Git

Git on macOS Option 1: Using Xcode Command Line Tools (Recommended)

1. Open the Terminal application (in Applications > Utilities).
2. Type the following command:

```
$ git --version
```

   If Git is not yet installed, macOS will prompt you to install Command Line Developer Tools. Click Install. This will install Git along with compilers and other tools needed for software development. Once the install is complete, try the "git –version" command again. You should now see a version number (e.g., git version 2.39.1).

Git on macOS Option 2: Install via Homebrew

1. Open Terminal.
   Install Git by typing:

```
$ brew install git
```

   After installation, verify with:

```
$ git --version
git version 2.39.1
```

Configuring Git (All Platforms)

After installing Git, configure your identity so your commits (changes) are properly attributed.

In your terminal (Git Bash or macOS Terminal), run:

```
$ git config --global user.name "Your Name"|
$ git config --global user.email "your.email@example.com"
```

You can confirm your settings with:

```
$ git config --list
Your Name
your.email@example.com
```

These values will be stored in a global configuration file (usually located at /.gitconfig).

Repositories

A Git repository (or repo) is a structured folder that tracks changes to files using Git. Think of it as a time machine for your project: it stores snapshots of your code, documents, or data at various points in time. You can revisit, compare, or restore earlier versions whenever needed. A Git repository can be "local", meaning that is only stored on your computer, or it can be stored on a shared platform like GitHub, GitLab, or Bitbucket where it can be shared with others.
A Git repository contains:

- Your project files (code, notebooks, scripts, etc.)
- A hidden .git folder that stores all the version history
- Metadata like commit messages, author info, and timestamps
- Branches for parallel development

Git repositories are powerful tools used to manage and track the evolution of a project's files over time. At their core, they enable users to keep a detailed record of changes, making it easy to see what was modified, when, and by whom. This is especially valuable in collaborative settings, where multiple people are working on the same files—Git helps prevent accidental overwrites by keeping each contributor's work organized and coordinated. Repositories also support the use of branches, which allow individuals to experiment with new features or ideas without affecting the main project. If a mistake is made or a change is no longer needed, Git makes it easy to revert to an earlier version.

In data science and research, Git repositories are commonly used to organize R or Python scripts, data cleaning routines, analysis code, manuscripts, reports, figures, and supplementary materials like README files. By storing all of these elements in a version-controlled environment, researchers can ensure transparency, reproducibility, and efficient collaboration throughout the lifecycle of a project.

Navigation Using the Terminal or Command Prompt

Even when using a shared platform like GitHub, you will want to designate a location for a repository on your computer. To do this, simply navigate to the desired location on your computer using the terminal.

1. On macOS or Linux
   - Open the Terminal
     Press Cmd + Space to open Spotlight Search.
     Type Terminal and press Enter.

   - See where you currently are
     When the terminal opens, you're usually in your home directory. You can check using the "pwd" (print working directory) command:

     ```
     $ pwd
     ```

This prints the current working directory (e.g., /Users/yourname).
You can list files and folders in the current directory using the "ls" (list):

```
$ ls
```

Navigate to another directory using the "cd" (change directory) command:

```
$ cd Documents/Projects
```

This moves you to the Projects folder inside Documents.
To move up one level, type:

```
$ cd ..
```

To go directly to your Desktop, type:

```
$ cd  /Desktop
```

Again, use pwd to verify where you are.

2. On Windows (Using Command Prompt)
   - Open Git Bash or Command Prompt
     Press Windows + S, type `git Bash` or `cmd`, and press Enter.
   - See where you currently are
     When the terminal opens, you're usually in your home directory. You can check using the "cd" (current directory) command:

```
$ cd
```

You can list files and folders in the current directory using the "ls" (list):

```
$ dir
```

Navigate to another directory using the "cd" (change directory) command:

```
$ cd C:
Users
YourUsername
Documents
Projects
```

(Replace YourUsername with your Windows username.)
This moves you to the Projects folder inside Documents.
To move up one level, type:

```
$ cd ..
```

To go to a specific location, like the Desktop:

```
$ cd  /Desktop
```

Creating a Repository

1. Create a Project Folder Once you have found a suitable location for your repository, you can create a new folder to hold its contents using the "mkdir" (make directory) command, then navigate to the newly created folder/repository:

```
$ mkdir my-first-repo
$ cd my-first-repo
```

   Once you are inside the newly created folder, you can initialize it as a Git repository using:

```
$ git init
Initialized empty Git repository in /Users/paul/Documents/PSYC672/.git/
```

   This creates a .git folder and marks the directory as a Git repository.
2. Add Files Create or copy some files into the folder. Then stage the files for your first commit. To "stage files for commit" in Git means to mark specific changes (files or parts of files) that you want to include in your next commit. Staging acts like a buffer zone between editing files and permanently recording those changes in the project's history. Imagine you're packing a box to mail. You don't just throw everything in the box and seal it right away. First, you select the items you want to include — that's the staging area. Once you're sure, you seal the box and ship it that's the commit.

```
$ git add .
```

   This stages all files for commit.
   You can stage only some changes from multiple files—useful for separating unrelated changes into clean, logical commits. It helps ensure that only reviewed and intentional edits are committed. Git lets you review what's staged (git diff –staged) before committing.
3. Make Your First Commit

```
$ git commit -m "Initial commit"
```

   In Git, to "commit" files means to permanently record a snapshot of your staged changes into the repository's history. Think of it as saving a version of your project that you can always go back to. The "-m" flag allows you to add a comment describing this commit, in this case "initial commit."
4. View History As you commit changes to a project as it evolves, you can always view a record of those changes by viewing the repository's "log."

```
$ git log
```

Connect to a Remote Repository

Assuming you're using GitHub:

1. Create a free account at https://github.com
   As of 2019, the free account will allow you to create and share unlimited repositories (both public and private), however, private repositories will have some limited functionality when using a free GitHub account.

2. Create a new GitHub repository

Create a new repository in GitHub

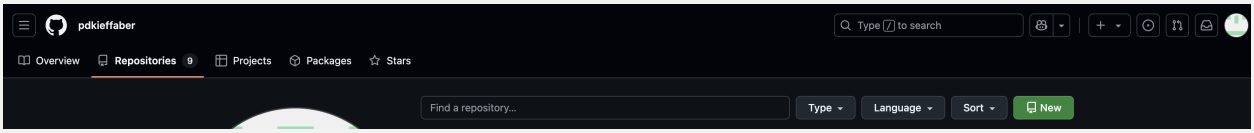


Figure 1.1: To create a new repository, first click the "Repositories" tab on the top left of the page, then click the "New" button.

Click the "New" button on the "Repositories" tab (See Figure **??**) to create a new repository. You'll be prompted to name your repository, optionally add a description, choose whether it will be public or private, and select initialization options (See Figure 1.2. These include adding a README.md file (useful for describing the project), a .gitignore file (to exclude certain files or folders from version control), and a license (to define how others can use your code). If you initialize with a README, your repository will start with that file already committed to the main branch. This setup provides a clean starting point for managing and sharing your project.

Initial Repo Parameters

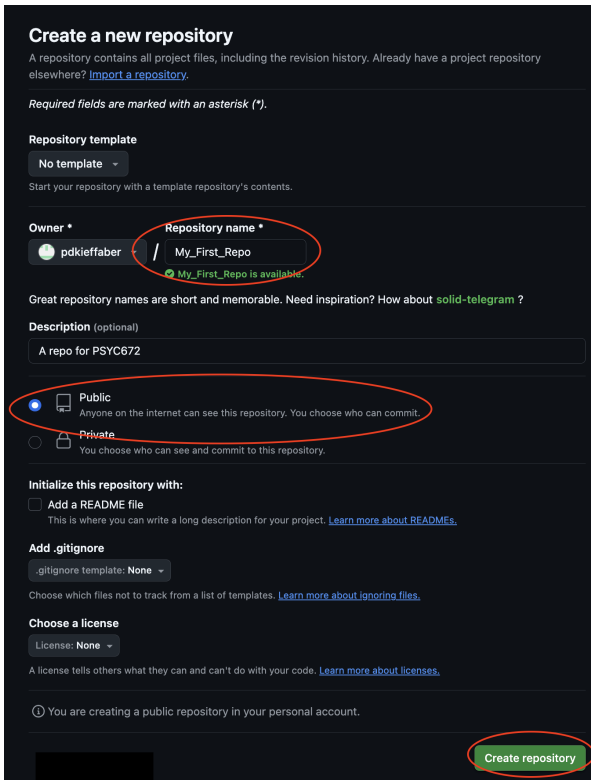


Figure 1.2. Give your new repository a name and designate it as public or private.

Note that at this point in the process, you have simply created one repository on your computer and another in GitHub. The next step is to connect and synchronize your local repository with the one you created on GitHub. This can sometimes be a bit tricky because the two repositories are, at this point, completely different. The first thing you will need is to generate a personal access token. A personal access token is a secure, user-specific alternative to a password that allows you to authenticate with Git hosting services like GitHub when using the command line or external tools. Because GitHub removed password authentication for Git operations over HTTPS in 2021, a personal access token is now required to push, pull, or clone repositories via HTTPS. This token functions like a temporary password and can be granted specific scopes or permissions, such as access to public repositories, private repositories, or workflow automation. You generate it from your GitHub account settings, and once created, you can use it in place of a password when prompted during Git operations. For security, tokens should be stored securely and rotated periodically.

3. Generate a personal access token.
   (a) Verify your email address if it hasn't been verified yet.
   (b) Click your profile photo in the upper-right corner of any GitHub page, then click Settings.
   (c) In the left sidebar, click Developer settings.
   (d) Under Personal access tokens, click Tokens (classic).
   (e) Click Generate new token, then Generate new token (classic).
   (f) Enter a name for the token in the "Note" field.
   (g) Select an expiration date under Expiration.
   (h) Choose the scopes to grant the token. For example, select repo to access repositories from the command line.
   (i) Click Generate token.

   Be sure to save the token somewhere discrete. Remember, it is like your private Git password.
4. Sync Local and Remote Repositories
   You might think of your local copy of the repository as your first "branch." A branch in Git is essentially a parallel version of your project. It is like a timeline of your project, where each branch represents a different line of development. The default branch is usually named "main" (or "master" in older repositories). Branches are created in order to do things like try out new features, fix bugs, or work on different versions of a project without changing the main version. Importantly, each branch tracks its own history of commits. Because the main branch in the newly created repository on GitHub is empty, you are going to move/copy your current branch (what is in your local repository) to the "main" branch on GitHub, overwriting the existing main branch if necessary.
   - Copy the GitHub repo URL (e.g., https://github.com/username/my-first-repo.git)
   - Set your local branch to be the main branch.

     ```
     $ git branch -M main
     ```

   - "Push" your Main Branch to the Remote Repository

     ```
     $ git remote add origin https://github.com/username/my-first-repo.git
     $ git push -u origin main
     ```

   The command above will push your local commits to GitHub. The -u flag sets the upstream (tracking) relationship between your local branch and the remote branch. After running this, your local main branch "knows" it corresponds to origin/main remotely.
   When you git clone, git fetch, git pull, or git push to a private remote repository using HTTPS URLs on the command line, Git will ask for your GitHub username and password. When Git prompts you for your password, enter your personal access token. Password-based authentication for Git has been removed in favor of more secure authentication methods.

Cloning a Repository

The previous example illustrated the process of moving an existing local repository to a GitHub repository that can be shared with others. To begin working on a project that is already hosted on GitHub (or another Git server), you can just clone the repository to your local machine using the git clone command. This creates a complete copy of the remote repository, including its history and all tracked files. For example:

```
$ git clone https://github.com/yourrepo.git
```

This command sets up a folder on your computer containing the project files and a .git directory for

tracking changes. It also links your local copy to the original remote repository, making it easy to fetch updates or push your own changes. In fact, if you are setting up a new repository from scratch, it would be more straightforward to create a new repository on GitHub, then clone that repository to your local machine.

The Basic Git Workflow

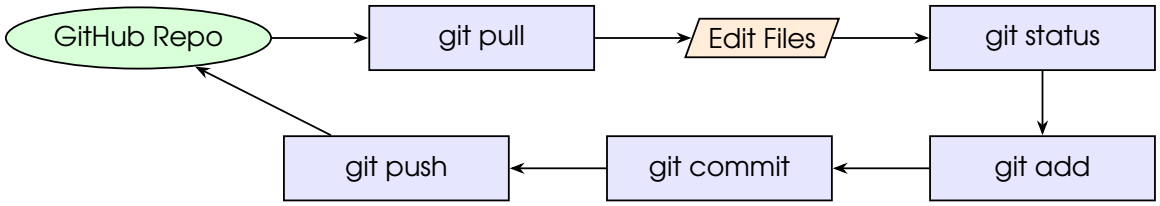


Figure 1.3: Recommended Git Workflow: Pull → Edit → Status → Add → Commit → Push

1. Staying Up to Date
   When working with collaborators (i.e., multiple people accessing the same repository), or even if it is just you working on multiple computers, it is important that you are working with the most recent version of the repository before you try to make any changes. By running "git pull" before making any changes to a repository you update your local branch with any new commits from GitHub and avoid accidentally overwriting someone else's work. Git will try to automatically merge the remote changes with your local changes and alert you if any conflicts exist, and let you know if you are already up to date!
   You can pull the most updated version of a branch using:

```
$ git pull origin main
From https://github.com/pdkieffaber/PSYC672
 * branch main -> FETCH_HEAD
 Already up to date.
```

2. Making Changes
   Once the repository is synchronized/cloned, you can begin making changes. This typically involves editing existing files or creating new ones using your preferred text editor or integrated development environment (IDE). Changes might include writing code, updating documentation, or adding figures and data files. Git will keep track of these modifications automatically, but nothing is saved to version control until you explicitly stage and commit the changes.
3. Checking Status
   At any point, you can check which files have been modified, added, or deleted using the git status command. This provides a quick overview of the current state of your working directory. It highlights which changes are untracked (new files), which have been modified but not yet staged, and which are ready to be committed. Regularly running git status is a good habit, as it helps avoid confusion and ensures you're aware of everything that's changed.

```
$ git status
```

4. Staging Files
   Before committing changes, you need to stage them with the git add command. Staging tells Git which changes you want to include in your next commit. You can stage individual files (e.g., git

add report.md) or all modified files at once using "git add .". This step allows you to group related changes together for clarity and cleaner commit history. *Only staged changes will be included in the next commit.*

```
$ git add .
```

5. Committing Changes
   Once your changes are staged, use git commit to save them to your local repository. A commit is like a snapshot of your project at a specific point in time. You'll include a descriptive message with each commit using the -m flag, such as:

   ```
   $ git commit -m "Fix typo in introduction and add conclusion section."
   ```

   This message helps you (and your collaborators) understand the purpose of the changes later. Each commit is stored in the repository's history, making it easy to review or revert if needed.
6. Pushing to GitHub
   After committing your changes locally, you can upload them to the remote repository (e.g., GitHub) using git push. This command syncs your local branch with the corresponding branch on the server. Most commonly, you'll use:

   ```
   $ git push origin main
   ```

   Here, origin refers to the remote repository, and main is the branch you're pushing to. This step makes your work available to collaborators and serves as a remote backup of your progress.

Branching

Branching is one of Git's most powerful features, allowing developers to diverge from the main line of development and continue work without affecting the main branch. A branch is essentially a pointer to a specific commit, allowing you to isolate changes for a particular feature, bug fix, or experiment. By default, a new Git repository starts with a single branch, often called main, which represents the official version of the project.

When you create a new branch (e.g., git branch feature-login), you're creating a parallel version of your repository where you can make changes independently of the main branch. You can then switch to this branch using "git switch." This makes branching especially useful for collaborative work: each contributor can develop features in their own branch without risk of introducing conflicts into the main project.

After finishing work on a branch, changes can be merged back into the main branch. This is typically done using git merge, which combines the histories of both branches. In cases where changes conflict, Git will prompt the user to resolve these conflicts manually. This ensures that only well-tested and intentional changes are added to the main branch.

Importantly, pushing a new branch to GitHub (e.g., git push) does not overwrite the main branch. Each branch exists independently on the remote as well as locally. This setup supports pull requests and code reviews, which are best practices in team-based development. By maintaining a clean and organized branching strategy, such as Git Flow or GitHub Flow, teams can work more efficiently and with fewer integration headaches.

For example, let's assume you are working with a research team on a project containing some data and an analysis script using R. The current code is working fine for now, but you have been tasked with coding a bootstrapping procedure in R that will enhance the analysis. With Git branching, you can begin

developing your code without interfering with any the core analysis pipeline that will still be used by your collaborators while you are developing your part of the project. Also, by cloning your branch, other team members can implement, test, and revise the bootstrapping logic independently. Once complete and validated, the branch can be merged back into the main branch, integrating the improvements into the official project codebase while maintaining a clean development history. This workflow helps ensure transparency, reproducibility, and minimal disruption to ongoing analyses.

Example Workflow

Assuming you have already cloned the repository to your local machine, the first step in an example workflow would be to execute a pull to make sure you are up to date. Make sure you're on the main branch and up to date:

```
$ git checkout main
Already on 'main'
Your branch is up to date with 'origin/main'.
$ git pull origin main
From https://github.com/pdkieffaber/PSYC672
 * branch main -> FETCH_HEAD
 Already up to date.
```

Now you are ready to start a new branch called "bootstrap-procedure". Let's say you're currently on the main branch and want to start developing bootstrapping the code:

```
$ git checkout -b bootstrap-procedure
Switched to a new branch 'bootstrap-procedure'
```

You're now on your new branch, ready to start making changes—Git will keep these changes isolated from main until you're ready to merge.

Now edit your R scripts (or create new ones) in your local repository to include the new bootstrapping procedure. When you want to make them available to your collaborators on GitHub you'll just need to stage the necessary file(s):

```
$ git add bootstrap_code.R
```

Note that you can use git add . to stage all changed files if appropriate.

Commit the changes you've made in the bootstrap-procedure branch.

```
$ git commit -m "Add initial version of bootstrapping procedure"
[bootstrap-procedure 03f6df2] Add initial version of bootstrapping procedure
1 file changed, 1 insertion(+)
create mode 100644 bootstrap_code.R
```

You can commit multiple times as you iteratively make changes to and test your new code.

Now push your branch to GitHub if you would like others to be able to work on it simultaneously:

```
$ git push -u origin bootstrap-procedure
Enumerating objects:  4, done.
Counting objects:  100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects:  100% (2/2), done.
Writing objects:  100% (3/3), 315 bytes | 315.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:  Create a pull request for 'bootstrap-procedure' on GitHub by
visiting:
remote:  https://github.com/pdkieffaber/PSYC672/pull/new/bootstrap-procedure
remote:  To https://github.com/pdkieffaber/PSYC672
 * [new branch] bootstrap-procedure -> bootstrap-procedure
branch 'bootstrap-procedure' set up to track 'origin/bootstrap-procedure'.
```

This workflow ensures that the main branch always reflects stable code, while branches provide safe spaces for iterative development.

Once you have finished developing the bootstrapping_code.R file and you have completed all testing, your changes can be seamlessly integrated back into the main branch. After merging, the feature branch can be deleted or kept for record-keeping.

Start the merge process by first switching back to the main branch and pull the latest changes:

```
$ git checkout main $ git pull origin main
```

Next, use git merge to merge the changes from your branch:

```
$ git merge bootstrap-procedure
```

If there are no conflicts, Git will automatically merge the changes.

Finally, push the updated main branch to GitHub:

```
$ git push origin main
```

Once merged and verified, you can (optionally) delete the branch locally:

```
$ git branch -d bootstrap-procedure
```

...and remotely:

```
$ git push origin -delete bootstrap-procedure
```

1.3 Collaboration with Git

Forking Vs. Branching

Branching is the standard approach for managing changes within a single Git repository, especially when working on new features, bug fixes, or experiments. A branch is like a parallel version of your project where you can make changes without affecting the main codebase. When you're ready, you can merge your changes back into the main branch (often called main or master). Branches are lightweight and

designed for collaborative development within a shared repository — they're perfect for teams working closely together with shared write access.

Forking, on the other hand, creates a personal copy of an entire repository under your own GitHub account. It's commonly used in open-source or large-scale collaborative projects where contributors do not have write access to the original repository. When you fork a repo, you can make changes freely in your copy and later submit a pull request to suggest your changes to the original project. Forking is ideal for contributing to public projects, exploring changes independently, or creating a personal adaptation of a tool or dataset.

In short, use branches when you're collaborating within a team or organization where everyone has access to the same repository. Use forks when you're contributing to someone else's project or want to maintain a separate version. Both support experimentation and version control, but forks provide an additional layer of ownership and permission separation.

You cannot create a fork directly from the Git command line — forking is a feature specific to GitHub (and similar platforms like GitLab or Bitbucket), not Git itself.
To fork a repository (via GitHub):

1. Go to the repository page on GitHub.
2. Click the "Fork" button in the upper right corner.
3. GitHub creates a copy of the repository under your own account.

After you've forked it on GitHub, you can clone your forked repository to your computer:

```
$ git clone https://github.com/your-username/forked-repo-name.git
$ cd forked-repo-name
```

Pull Requests

A pull request (PR) is a way to propose changes you've made in a branch (or forked repository) to be merged into another branch, typically the main branch of the original project. Pull requests are central to collaboration on GitHub, allowing teams to review, discuss, and integrate code changes in a structured and trackable way.

Before you can create a pull request, your changes must be pushed to a branch on GitHub. For example, if you've created a new branch locally called feature/bootstrap-analysis:

```
$ git checkout -b feature/bootstrap-analysis
# MAKE YOUR CONTRIBUTIONS...
$ git add .
$ git commit -m "Add bootstrap analysis to R script"
$ git push origin feature/bootstrap-analysis
```

Once your branch has been pushed to GitHub, go to your repository's GitHub page in your browser. GitHub will often show a banner suggesting you create a pull request for the newly pushed branch. Click "Compare & pull request". If no banner appears, go to the "Pull requests" tab and click "New pull request". Choose the branch you want to merge into (typically main) and the branch you want to merge from (e.g., feature/bootstrap-analysis).

When prompted, give your pull request a descriptive title and detailed description of the changes you made. This helps reviewers understand the intent of your changes. For example:

```
   Title:  Add bootstrapping functionality to analysis pipeline
Description:This PR adds a bootstrapping procedure to the analysis.R file for
```

```
resampling reaction times.  It uses 10,000 resamples and returns 95% confidence
intervals.  Also includes updated unit tests in test_analysis.R.
```

Finally, click "Create pull request". You can now request specific reviewers, assign labels, and link related issues.

1.4 Version Control for Experiments and Data Analysis

Version control is not just for software developers, it can help to manage the evolution of analysis scripts, experimental materials, and documentation. Researchers can leverage Git and GitHub to make every stage of the scientific process more organized, transparent, and reproducible. By versioning analysis scripts, experimental code, stimuli, and lab notebooks, researchers create an audit trail that enhances collaboration and scientific rigor. This section explains how Git can be used to manage different aspects of experimental research.

Tracking Analysis Scripts with Git

In many areas of research, analysis scripts often evolve over time as analytic strategies change, new datasets are collected, or new subjects are added to an existing dataset. Without version control, tracking the provenance of an analysis pipeline or understanding the impact of a small code change can become almost impossible. Git allows researchers to:

1. Track line-by-line changes in R, Python, or MATLAB scripts.
2. Experiment with new methods in branches without disrupting the main pipeline.
3. Revert to previous versions when errors or regressions are discovered.
4. Tag important milestones, such as versions used in a submitted manuscript.

Example Workflow: Add and commit your new script:

```
$ git add analysis_v1.R
$ git commit -m "Initial version of analysis script"
```

Make updates and track changes:

```
$ git add analysis_v2.R
$ git commit -m "Refactored script to include new preprocessing step"
```

Tag manuscript version:

```
$ git tag -a v1.0 -m "Version used in preprint submission"
```

Managing Experimental Designs and Stimuli

In behavioral and cognitive experiments, stimuli files (images, audio clips, text), presentation scripts (e.g., in PsychoPy, E-Prime, jsPsych), and configuration settings change frequently during the development and piloting phases. Using Git to manage these files helps you avoid the confusion of having multiple folders labeled "Final", "Final2", or "Final_final", makes collaboration with lab members or co-authors more transparent, and allows you to roll-back to working versions if bugs are introduced.

When using Git to manage experimental designs and stimuli, it is generally recommended that you follow a standard repository structure with separate folders for stimuli, data, scripts, and results.
Suggested Repository Structure for Experiments:

```
experiment-project/
|
|-- stimuli/
|    |-- images/
|    |-- audio/
|    |-- text/
|
|-- data/
|    |-- (add to .gitignore if large)
|
|-- scripts/
|    |-- present_experiment.py
|    |-- experiment_config.json
|
|-- results/
|    |-- (add to .gitignore if large or auto-generated)
|
|-- README.md
|-- LICENSE
```

1.5 Using Markdown with Git/GitHub

What is Markdown?

Markdown is a lightweight markup language created by John Gruber in 2004 to make writing formatted text simple and readable in plain text. Markdown files use the .md file extension and are widely adopted for technical documentation, README files, project wikis, and more. The appeal of Markdown lies in its easy-to-learn syntax and its ability to be rendered as rich HTML or PDF documents by a variety of tools and platforms.

Why Use Markdown with Git?

Markdown and Git are a natural pairing for scientific and technical documentation:

- Version Control: Git tracks every change to your Markdown files, letting you revert, compare, or collaborate on documentation just as you would with code.
- Transparency: All edits are timestamped and attributed, supporting reproducibility and accountability.
- Collaboration: Multiple users can work on the same files, resolve conflicts, and review changes via pull/merge requests.
- Web Integration: Many platforms (e.g., GitHub, GitLab) automatically render .md files as formatted web pages, making your documentation instantly readable.

Features Included: Headings
Bold and Italic text
Lists (ordered and unordered)
Links
Images
Inline code
Code blocks

1.6 Tracking & Reviewing Changes

Once you start version-controlling your work with Git, it's important to understand how to inspect changes, compare revisions, and understand the history of your project. These capabilities are essential for debugging, documenting, and collaborating in scientific research. Git offers a suite of powerful tools for reviewing file changes, commit history, and differences between branches.

Checking What Has Changed: `git status`

The first step in tracking changes is to ask Git what's new, modified, or staged for commit:

```
$ git status
```

This command will show:

- Untracked files: new files not yet added to Git
- Modified files: tracked files that have changed but not yet staged
- Staged changes: files that are ready to be committed

SHOW EXAMPLE OUTPUT AND EXPLAIN

Seeing the Details of Your Changes: `git diff`

Sometimes you may want to see exactly what lines have changed in your files:

```
$ git diff
```

This will display the changes in the working directory that are not staged. Once you stage them (git add), the changes disappear from git diff. To see staged changes:

```
$ git diff -cached
```

You can also compare specific files:

```
$ git diff analysis_script.R
```

Or compare different commits or branches:

```
$ git diff main experiment-branch
```

SHOW EXAMPLE OUTPUT AND EXPLAIN

Reviewing Commit History: `git log`

To see a record of commits, `git log` shows commit hashes, authors, dates, and messages:

```
$ git log
commit 3a81b2f3018d2cbbd540e8d1e0db457ca170c9c4 (HEAD -> main)
Author:  Paul Kieffaber <paul@example.edu>
Date:  Tue Jun 18 14:03:00 2025 -0400
Refactored bootstrapping code into separate function
commit c71ae6c8481732ccf7c993dcfcdf5e1a24c5fd9c
Date:  Mon Jun 17 16:20:34 2025 -0400
Added initial analysis script for Cognitive Control Study
```

For a simpler view:

```
$ git log -oneline
```

Or with a graphical view of branches:

```
$ git log -oneline -graph -all
```

Viewing Specific File History

If you want to see the history of a single file:

```
$ git log - analysis_script.R
```

To view changes made to that file:

```
$ git diff HEAD^HEAD - analysis_script.R
```

To see who last edited each line of a file:

```
$ git blame analysis_script.R
```

This can be helpful when debugging or reviewing collaborative contributions.

1.7 Handy Tips and Best Practices for Using Git and GitHub

While Git's core functionality—tracking changes and managing versions—is already powerful, several features and best practices can make your workflow cleaner, more efficient, and easier to manage, especially in collaborative or research-focused projects.

Use `.gitignore` to Exclude Unwanted Files

In most projects, there are files that should not be tracked by Git—such as temporary outputs, compiled files, or large datasets. A `.gitignore` file can be created using any text editor and it tells Git to ignore the files and folders designated in that file. Create a .gitignore file in the root of your repository, and Git will apply these rules when adding and committing files. This helps keep your version history clean and prevents accidentally uploading sensitive or unnecessary data.
Below is an example of a `.gitignore` file for a research project:
NOTE: the hashtag symbol indicates a descriptive comment

```
# Ignore everything in the results folder and all .log files
/results/
/*.log

# Ignore all .csv and .tsv files in the data folder
/data/*.csv
/data/*.tsv
```

Commit Often and Write Clear Messages

Frequent commits make it easier to track changes and revert specific modifications if needed. A good commit message should be concise but descriptive, stating what was changed and why.
Poor message:

```
$ git commit -m "05/23/26 - updated file"
```

Better message:

```
$ git commit -m "05/23/26 - Added log transformation to reaction time analysis
for normality"
```

Use `git tag` to Mark Significant Milestones

Git tags are a powerful way to mark specific points in your repository's history—typically used to indicate releases, milestones, or versions (e.g., v1.0, paper-submission, final-analysis).
What Does Git Tag Actually Do?

- Git tags a specific commit, not a branch or a file directly.
- It does not tag individual files (like marking one .R or .py file)—instead, it marks the snapshot of the entire repository at a particular commit.
- Because tags point to commits, and branches also point to commits, a tag can be thought of as a "bookmark" for a particular state of your repo, regardless of which branch it was on.

There are two types of tags, lightweight tags and annotated tags.
Lightweight Tag →Just a name that points to a commit

```
$ git tag manuscript-submission
```

Annotated Tag→Contains a message, tagger name, and date—useful for documentation and releases.

```
$ git tag -a manuscript-submission -m "Version for journal submission"
```

If you are working on a local repository, you will need to push the tag to GitHub:

```
$ git push origin manuscript-submission
```

To go back to a tagged version in Git, you can use the git checkout command—but how you do it depends on what you want to achieve.
Case 1: Temporarily inspect a tagged version (detached HEAD)
If you just want to look at or run the code at the tagged state (without making changes):

```
$ git checkout manuscript-submission
```

NOTE: This puts you in a "detached HEAD" state. In Git, HEAD is a pointer that tells you where you currently are in your repository—specifically, it points to the latest commit on the current branch. In other words, "HEAD" is your current location in the project history. For example, If you're on the main branch, HEAD points to the latest commit on main. A "Detached HEAD" state is when HEAD is pointing directly to a commit, not to a branch. When you checkout a tagged commit from the past, you're not on a branch, so you can't commit changes unless you create a new branch. This state is useful only for inspecting exactly what the repo looked like at the time of the tag.

Case 2: Create a new branch from the tag (to make changes)
If you want to modify code based on a tagged version:

```
$ git checkout -b manuscript-revision manuscript-submission
```

This creates a new branch called manuscript-revision starting from the commit the tag manuscript-submission points to. You can now make changes (i.e., commits, merge changes, etc.) to this new branch.

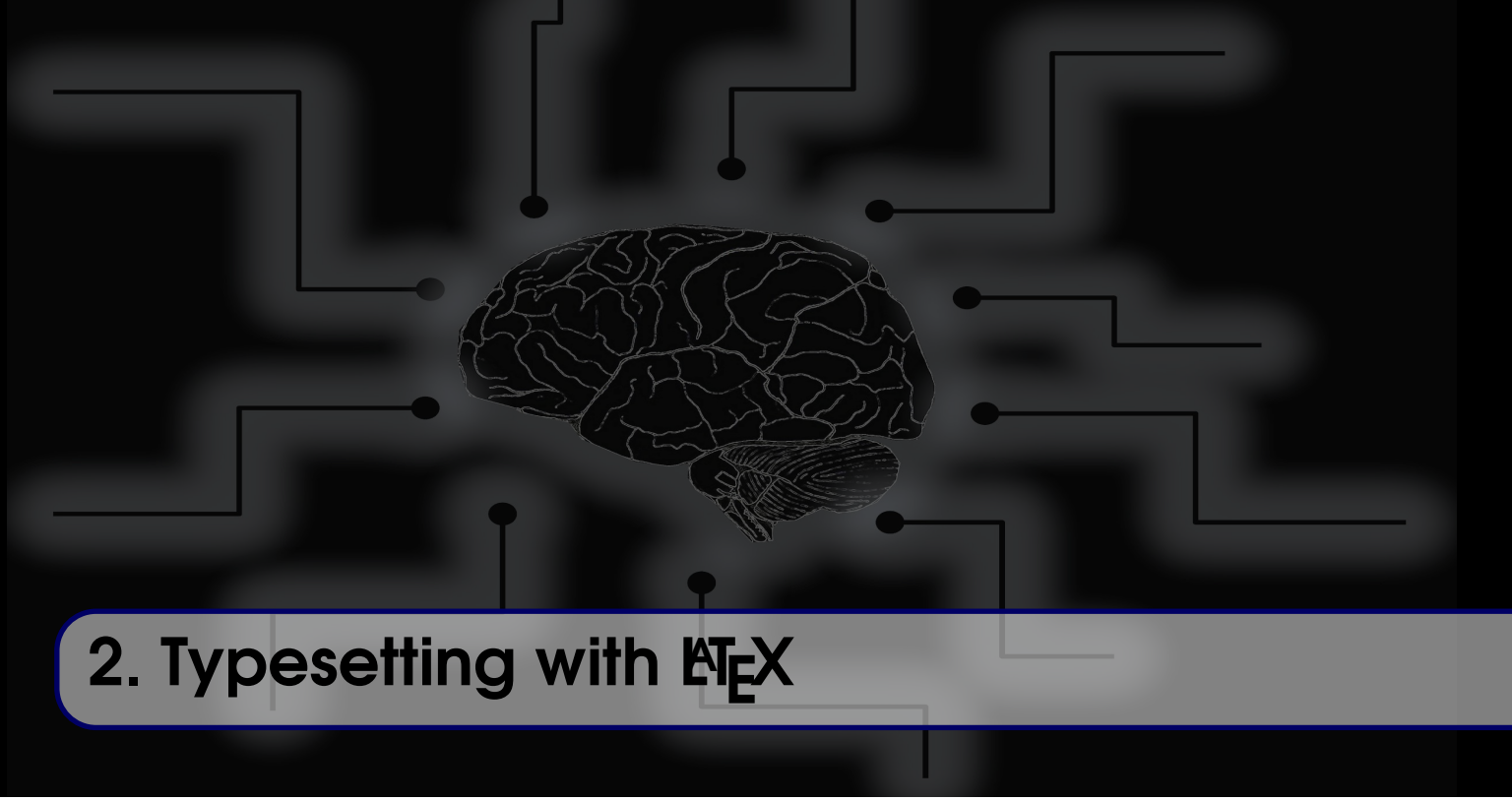
1.8 Assignment

1. Initial Setup by Instructor
   Create a GitHub repo (public or private) with:
   - A LaTeX template
   - A bibliography file (refs.bib)
   - A repo with the folder structure:
     research-report/
     |—- main.tex
     |—- intro.tex
     |—- methods.tex
     |—- results.tex
     |—- discussion.tex
     |—- main.tex
     |—- references.bib
     |—- README.md
2. Student Instructions
   (a) Fork the PSYC672 repo
      i. Visit https://github.com/pdkieffaber/PSYC672.
      ii. Click "Fork" (top right)
      iii. This will create a new fork in your GitHub account.
   (b) Clone the fork to your computer:

```
$ git clone https://github.com/STUDENT-USERNAME/PSYC672.git
$ cd PSYC672
```

   (c) Edit the README.txt file, adding the line, "*First Last*-FAll, 2025" (replace *First* with your first name and *Last* with your last name)
   (d) Commit changes and push:

```
$ git add .
$ git commit -m "Added my introduction section"
$ git push origin main
```



2. Typesetting with LaTeX

Style means the right word. The rest matters little.

— Jules Renard

Contents

2.1 What is LaTeX ?

LaTeX, pronounced "Lay-tech" or "Lah-tech", is a high quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is widely used in academia for the communication and publication of scientific documents in many fields, including mathematics, statistics, quantitative psychology, computer science, engineering, chemistry, physics, economics, linguistics, philosophy, and political science. It also has a prominent role in the preparation and publication of books and articles that contain complex elements. LaTeX is available for free at https://www.latex-project.org/.

When using LaTeX, the writer uses plain text as opposed to the formatted text found in WYSIWYG ("what you see is what you get") word processors like Microsoft Word, Google Docs, LibreOffice Writer and Apple Pages. In LaTeX, the writer uses markup tagging conventions to define the general structure of a document (such as article, book, and letter), to stylize text throughout a document (such as bold and italics), and to add citations and cross-references.

To make clear how LaTeX is different from WYSIWYG word-processing software you are likely already familiar with, consider that In most publishing environments, the author first submits his or her text for publication. Next, a human designer determines what the layout of the publication will be (i.e., headers, headings, images, etc.). Finally, a typesetter combines the author's text and instructions from the designer to create the finished product. In this example, LaTeX takes on the role of designer and TeX the role of typesetter. This is quite different from the WYSIWYG approach that most modern word processors, such as MS Word or LibreOffice, take. With these applications, authors specify the document layout interactively while typing text into the computer. They can see on the screen how the final work will look when it is printed, taking on the roles of author, designer, and typesetter.

Why LaTeX ?

The first question asked by most WYSIWYG users upon hearing a description of LaTeX is, Why? It is my humble opinion that using LaTeX is a bit like bringing a calculator to a math test. If you know how to use it, it can make things so much easier - so much easier it feels like cheating. When you know how to use LaTeX, you won't have to remember anything about APA formatting. LaTeX will take care of everything for you. All you have to do is focus on your writing. Still, it is common for WYSIWYG and LaTeX users to debate about the pros and cons of each approach. Thus, a few of the most commonly promoted advantages and disadvantages of using LaTeX are listed below.

Advantages

- LaTeX frees the author to focus almost entirely on content
- Many layouts are available, producing professionally crafted documents
- Output is consistent and replicable
- Excellent typesetting of mathematical formulae
- Easy generation of complex structures (footnotes, references, TOC, bibliographies)
- User-friendly over time: once you learn the basics, it's easy to get help online
- Most LaTeX editors have a user-friendly interface
- Rich documentation due to long history
- High-quality illustrations with PSTricks or TikZ
- Better at preparing complex tables

Disadvantages

- Fairly steep learning curve
- Collaborators unfamiliar with LaTeX may have difficulty reviewing your manuscripts
- Many features require additional libraries (e.g., to track changes)
- Custom layout changes can be time-consuming to implement
- No native spell-check or grammar-check

Getting LaTeX

LaTeX is free software under the terms of the LaTeX Project Public License (LPPL). LaTeX is distributed through CTAN servers or comes as part of many easily installable and usable TeX distributions provided by the TeX User Group (TUG) or third parties. Below are the recommended distributions for the most popular operating systems.

Platform	Instructions
Linux	Installing the `TeX Live` distribution is advisable, as many Linux distributions only contain older versions. See the official TeX Live documentation for package status and installation details.
Mac OS	The `MacTeX` distribution contains everything you need, including a complete TeX system with LaTeX itself and user-friendly editors to write documents.
Windows	Check out the `MiKTeX`, `proTeXt`, or `TeX Live` distributions. They all contain a complete TeX system with LaTeX and editors for writing documents.
Online (cloud-based)	LaTeX online services like `Overleaf`, `Papeeria`, `Datazar`, and `LaTeX Base` let you edit, view, and download LaTeX files and resulting PDFs in your browser. These are great for beginners or collaboration.

Using LaTeX

Once you have started a new blank document, or decided on a template, it's time to start editing. Once you open a project, the screen will be divided into three sections. On the far left, you will find a view of the current working directory for the project where you will see all of the files associated with the current project and find options for upload files (e.g., figures and images). The middle section is the editor where you will write and make changes to the content of your document. Finally, the right section is the output viewer, which allows you to preview your document after being compiled with TeX formatting, but without having to save it as a PDF. By default it is set to automatically update as changes are made, but it can be toggled to a manual option. The project screens in Overleaf and the TexMaker IDE (just for comparison) are illustrated below.

Common LaTeX Editors

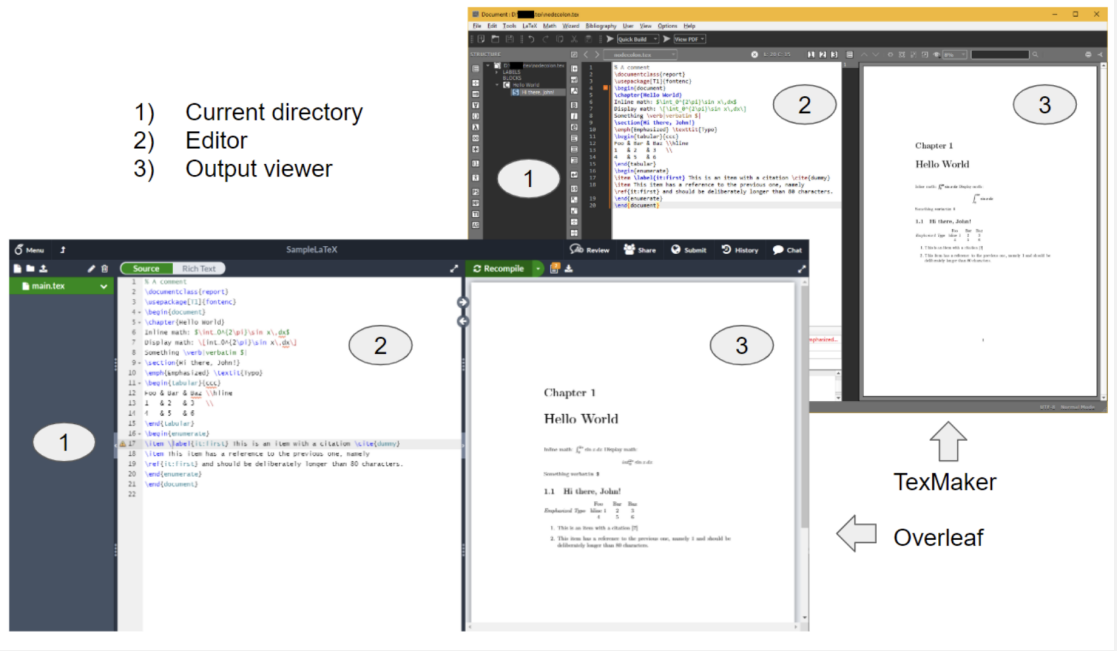


Figure 2.1: Most editors have three panels: (1) A file directory, (2) an editor, and (3) an ouput viewer.

When editing a project there are two options to select from in Overleaf: Source and Rich Text. Both can use the toolbar that inserts code for you for particular actions, but will look different in their set-up. The Rich Text option creates a WYSIWYG type interface where formatting is mostly done by clicking buttons rather than coding by hand. This can be useful when getting familiar with LaTeX, but this tutorial will focus primarily on Source editing.

2.2 LaTeX Preliminaries

Before you start adding content to your document, there are a few important things to know about the LaTeX environment.

Spaces

"Whitespace" characters, such as blank or tab, are treated uniformly as "space" by LATEX. Several consecutive whitespace characters are treated as one "space". Whitespace at the start of a line is generally ignored, and a single line break is treated as "whitespace". An empty line between two lines of text defines the end of a paragraph. Several empty lines are treated the same as one empty line. The text below is an example. On the left hand side is the text from the input file, and on the right hand side is the formatted output.

LaTeX	Formatted Output
`It  does not matter  whether you enter one or several  spaces after a word.` `An empty line starts a new paragraph.`	It does not matter whether you enter one or several spaces after a word. An empty line starts a new paragraph.

Comments

When LaTeX encounters a % character while processing an input file, it ignores the rest of the line, the line break, and all whitespace at the beginning of the next line. This can be used to write notes into your document that will not show up in the printed version.

LaTeX	Formatted Output
`This is a %stupid great example of using the % character to add comments to a document.`	This is a great example of using the % character to add comments to a document.

Special Characters

There are a number of special characterssymbols that are reserved characters that either have a special meaning in LaTeX or are not available in all the fonts. If you enter them directly in your text, they will normally not print, but rather coerce LaTeX to do things you did not intend or potentially cause errors when compiling your document. Some of those characters, the LaTeX command required to produce them in your formatted output, and a brief description are shown below:

Symbol	LaTeX	Description	Use
%	\%	Percent	The comment command
$	\$	Dollar sign	Used for inline math environments
{ or }	\{	Curly braces	Used to create groups
#	\#	Number sign	Define function parameters in macros
&	\&	Amperstand	Separate columns in tabular environments
\	\textbackslash	Backslash	Used to call macros
_	_	Underscore	Indicates underscore in math mode
^ or ~	\^ or \~	Caret or tilde	Used for accents (in math) or non-breaking space (text).

Note that the backslash character can not be entered by adding another backslash in front of it (\\). This double-backslash sequence is used for line breaking. Printing a backslash character requires using the \textbackslash command instead.

Emphasizing Text

Simple text formatting helps to highlight important concepts within a document and make it more readable. Using italics, bold or underlined words can change the perception of the reader. Three of the most common ways of emphasizing text are illustrated below.

LaTeX	Formatted Output
`Use \textbf{for bold-face}`	Use **for bold-face**
`Use \textit{for italics}`	Use *for italics*
`Use \underline{for underlining text}`	Use for underlining text
`Formatting commands can also be \textbf{\textit{nested}}`	Formatting commands can also be ***nested***

Quotation Marks

Single quotation marks are produced in LaTeX using ‘ and ’. Double quotation marks are produced by typing " and ". (The undirected double quote character " that you might be used to produces double right quotation marks: it should never be used where left quotation marks are required (see Example below).

LaTeX	Formatted Output
``Hello world´´	“Hello world”
`Hello world´	‘Hello world’
"Hello world"	"Hello world"

Macros

In LaTeX, a macro is a command that automates formatting or functionality. Macros often begin with a backslash (\) and may take one or more arguments. They help you keep your document clean, reusable, and consistent.

There are two kinds of macros beginners often encounter:

1. Built-in macros like \section{}, \textbf{}, \emph{}, which are used for formatting
2. Environment macros like \begin{itemize}...\end{itemize} that are used for structures like lists, tables, or math. Examples of some of the most commonly used environment macros are illustrated below.

Environment Name	LaTeX	Formatted Output
Enumerate (numbered list)	\begin{enumerate} \item Step one \item Step two \item Step three \end{enumerate}	1. Step one 2. Step two 3. Step three
Itemize (ordered list)	\begin{itemize} \item Apples \item Oranges \item[\S] Bananas \end{itemize}	• Apples • Oranges § Bananas
Center (text centering)	\begin{center} Centered text \end{center}	Centered text
Quote	\begin{quote} "A short quote." \end{quote}	“A short quote.”
Equation	\begin{equation} E = mc^2 \end{equation}	$E = mc^2$ (2.1)

There are also optional parameters that can be passed to a macro to change its behavior. Optional parameters are placed inside brackets. In the example above, the command \item[\S] does the same as \item, except that the input parameter \S, instructs LaTeX to use the section sign as the bullet for this item.

Creating Macros

LaTeX is stocked with a huge number of macros that cover almost any formatting task. Nevertheless, it is sometimes useful to define your own macros to simplify repetitive and/or complex formatting that may be unique to your work.

New macros are defined using the \newcommand macro, which takes a minimum of two parameters. The first parameter is the name you want to give the new macro and the second tells LaTeX what the macro should do. For present purposes, we will focus on an example that might be useful in the Psychological

Sciences. When reporting statistical results, there are specific rules regarding spacing and text formatting, that can be somewhat tedious to produce, especially when many results are being reported. For example, the results of an F-test (ANOVA) takes the general form, F(DFb,DFw) = #, p<0.05, where DFb/DFw refer to the "between" and "within" subject degrees of freedom respectively, # refers to the value of the F statistic, and the labels of the computed statistics (e.g., F and p) should be italicized. One way to simplify the process of repeatedly reporting the results of F-tests would be to automate the process of typesetting the result (see Example below).

LaTeX	Formatted Output
`\newcommand{\Ftest}[4]{\textit{F}(#1,#2) = #3, \textit{p}$<$#4}`	This will not print, but will create a macro called "Ftest" that takes four input parameters.
`There were significant differences in performance between students in the various classroom environments, \Ftest{3}{55}{9.9}{0.05}.`	There were significant differences in performance between students in the various classroom environments, $F(3,55) = 9.9, p < 0.05$.

Note that once you have created a new macro for the reporting of your statistical results, formatting the output not only becomes very easy for multiple results, but you will ensure perfect consistency in the reporting of results throughout the paper. Moreover, should you like to change the way results are reported in your document after it is finished, all that is needed is to change the way your macro formats the output and then re-compile the document!

2.3 Creating a LaTeX Document

When LaTeX processes an input file, it expects it to follow a particular structure. The input document must start with the command, `\documentclass{...}`. This command specifies what class of document you intend to write. After that, additional commands can be used to influence the style of the whole document, or load packages that add specific features (i.e., macros) to the LaTeX environment using the command, `\usepackage{...}`. These "setup" commands at the beginning of the document are sometimes called the "preamble". The body of the document begins with the command, `\begin{document}`. Finally, at the end of the document, use the command `\end{document}` to tell LaTeX that this is the end of the LaTeX input. Anything that follows this command will be ignored by LaTeX. Two simple LaTeX documents are illustrated below.

LaTeX	Formatted Output

`% A VERY simple document`
`\documentclass{article}`
`\begin{document}`
`Hello World!`
`\end{document}`
	Hello World!

More complex documents simply require the author to define parameters like the title of the document and the name of the author etc.

LaTeX

```
% A more complex document
\documentclass[a4paper,11pt]{article}
% define the title
\author{H.~Partl}
\title{Minimalism}
\begin{document}
% generates the title
\maketitle
% insert the table of contents
\tableofcontents
\section{First section}
Well, and here begins my lovely
article.
\section{Second section}
\ldots{} and here it ends.
\end{document}
```

Formatted Output

Minimalism

H. Partl

Contents

1 First section

Well, and here begins my lovely article.

2 Second section

... and here it ends.

The Document Class

Notice in the example above, the first bit of information LaTeX needs to know when processing an input file is the type of document the author wants to create. As is illustrated above, this is specified with the `\documentclass` command. There are many classes provided in the default LaTeX distributions as well as additional classes (e.g., the APA style class) that can be added. The `\documentclass` command also takes a number of optional parameters that customize the behavior of the document class. The options are bracketed and separated by commas. Some of the most commonly used classes and options are listed in the Table below.

Common Document Classes

Class	Description
`article`	Ideal for articles in scientific journals, presentations, short reports, program documentation, etc.
`proc`	A class for proceedings based on the `article` class.
`minimal`	As small as it can get. It only sets a page size and a base font. Mainly used for debugging.
`report`	For longer reports containing several chapters, small books, PhD theses, etc.
`slides`	For slides. The class uses big sans serif letters.
`book`	For real books.

Common Document Class Options

Option	Description
`10pt`, `11pt`, `12pt`	Sets the size of the main font in the document. If no option is specified, 10pt is assumed.
`a4paper`, `letterpaper`, ...	Defines the paper size. Default is `letterpaper`. Also supports `a5paper`, `b5paper`, `executivepaper`, `legalpaper`.
`titlepage`, `notitlepage`	Specifies whether a new page should be started after the title. `article` does not by default; `report` and `book` do.
`onecolumn`, `twocolumn`	Typesets the document in one or two columns.
`landscape`	Changes layout to landscape orientation.

LaTeX Packages

While writing your document, you will almost certainly find that basic LaTeX cannot solve all of your problems. For example, if you want to include graphics, use colored text or source code from a separate file into your document, you will need to enhance the capabilities of LaTeX by adding so-called "packages". Packages are activated with the `\usepackage[options]{package}` command, where package is the name of the package you want to add and options is a list of keywords that trigger special features from that package. The `\usepackage` command goes into the preamble of the document (i.e., before the `\begin{document}` command.

One example of a commonly used package is the "graphicx" package. Latex cannot manage images by itself, so the `graphicx` package is needed to help place images in LaTeX documents. To use it, include the following line in the preamble of your document, `\usepackage{graphicx}`. Then, images can be inserted into the body of your document using the `\includegraphics{filename}` command, where "filename" is the name of the image file (excluding the extension).

Note that If your image files are stored in a separate directory, you need to include the path to the file or include the `\graphicspath{directory}` command in the preamble, where "directory" refers to the path to the directory containing the images. Below are several examples of ways to set up the graphics path.

LaTeX	Description
\graphicspath{ {images/} }	%Path relative to the .tex file containing the \includegraphics commands . . . in this case, a folder named "images"
\graphicspath{ {./images/} }	%Path relative to the main .tex file . . . in this case, a folder named "images"
\graphicspath{ {./images1/}{./images2/} }	%Multiple paths relative to the main .tex file . . . in this case folders named "images1" and "images2"
\graphicspath{ {C:/user/. . . /images/} }	%Filepath in Windows
\graphicspath{ {/home/user/images/} }	%Filepath in Mac OS & Linux

2.4 Images in LaTeX

We just discussed how working with images in LaTeX requires the `graphicx` package to be loaded and use of the `\includegraphics` command. In fact, the The `\includegraphics` command also takes a number of optional parameters that can help you achieve additional formatting effects in your finished document, including sizing, scaling, and rotation of the image. Several examples of those optional parameters are

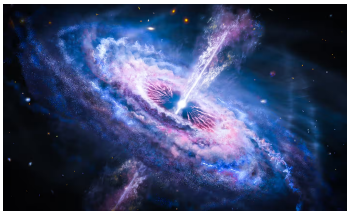
illustrated below.

Input

```
%Example setting the image width to .6 of
the linewidth
\documentclass{article}
\usepackage{graphicx}
\graphicspath{ {./images/} }

\begin{document}
The universe is immense and it seems to be
homogeneous,
in a large scale, everywhere we look at.

\includegraphics[width=0.6\linewidth]{univer

There's a picture of a galaxy above
\end{document}
```

Formatted Output

The universe is immense and it seems to be homogeneous, in a large scale, everywhere we look at.



There's a picture of a galaxy above.

```
% Example using image scaling and rotation
\documentclass{article}
\usepackage{graphicx}
\graphicspath{ {./images/} }

\begin{document}
The universe is immense and it seems to be
homogeneous,
in a large scale, everywhere we look at.

\includegraphics[scale=.1,
angle=45]{universe}

There's a picture of a galaxy above
\end{document}
```

The universe is immense and it seems to be homogeneous, in a large scale, everywhere we look at.

There's a picture of a galaxy above.

Image Positioning

In the previous section we covered how to include images in your document, but the combination of text and images may not always look as expected. To increase control over image placement and add other options with respect to images we need to introduce a new environment. The "figure" environment is used to display images as floating elements within the document. This means you include the image inside the figure environment and you don't have to worry as much about it's placement because the figure environment takes care of that for you.

Still, there are times one may want to override the figure environment defaults and exercise more control over the way the figures are displayed. Luckily, optional parameters can be passed to the `figure` macro to adjust the figure positioning. For example, the command, `\begin{figure}[h]`, sets the positioning parameter to "h", which tells to LaTeX that you want the figure right "here". Below is a table illustrating some of the possible positioning values.

Parameter	Position Description
h	Place the float *here* — approximately where it occurs in the source code (not guaranteed to be exact).
t	Position at the top of the page.
b	Position at the bottom of the page.
p	Put the float on a special page for floats only.
!	Override LaTeX's internal rules for what's considered a "good" float placement.
H	Place the float **exactly** at the spot in the code. Requires the 'float' package. Similar to h!, but more forceful — can sometimes cause layout issues.

Note that the default alignment of figures is left. The addition of a `\centering` *command inside the figure environment will center the picture.*

Wrapping Text Around Figures

The figure environment places figures in-line with text, meaning that the figures occupy the entire line on which they are placed and text will appear above and/or below the figure. It is also possible to wrap text around the figures using the "wrapfig" package. Before you can use the `\wrapfigure` environment, you have to import the "wrapfig" package by adding the line `\usepackage{wrapfig}` to the preamble.

When you initialize a wrapfigure environment, there are two optional parameters that should be provided. The first parameter instructs LaTeX about the positioning of the environment on either the left (`{l}`) or right (`{r}`) and the second parameter instructs LaTeX about the desired width of the figure box (not the width of the image itself, that must be set with the `\includegraphics` command). It is often most convenient to set the width relative to the text width, but standard units can also be used (cm, in, mm, etc). An example of the wrapfigure environment is illustrated below.

LaTeX

```
%This figure will be on the right
\begin{wrapfigure}{r}{0.25\textwidth}
\centering
\includegraphics[width=0.25\textwidth]{image}
\end{wrapfigure}

Participants.  Participants will be recruited from the subject pool at William &
Mary and from the local community in a targeted way so as to ensure representation
of minority groups in the sample.  Specifically, we aim to recruit a sample of
participants that consists of 40% White (N=100), 20% African American/Black (N=50),
20% Asian (N=50), and 20% Hispanic/Latino (N=50).  Each participant will be asked to
complete an assessment of demographic and psychological variables....
```

Formatted Output

Participants. Participants will be recruited from the subject pool at William & Mary and from the local community in a targeted way so as to ensure representation of minority groups in the sample. Specifically, we aim to recruit a sample of participants that consists of 40% White (N=100), 20% African American/Black (N=50), 20% Asian (N=50), and 20% Hispanic/Latino (N=50). Each participant will be asked to complete an assessment of demographic and psychological variables. Demographic data collected will include, but not necessarily be limited to, age, education, gender, race/ethnicity, history of concussion, history of psychological disorders, and birth order.

Psychological Variables. All psychological variables will be measured using the shortest, validated version of available standardized instruments. Personality traits will be measured using the BFI-10 [58]. Depression will be measured using the CES-D-10 [59]. Anxiety will be measured using the STAI-6 [60]. Importantly, any one (or combination) of these demographic and personality variables can be used by students and/or researchers to subset the recorded EEG data in the service of their specific research question(s).

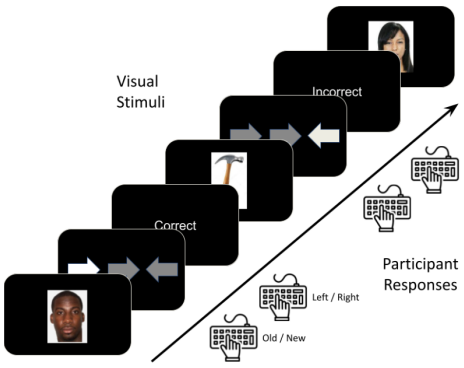


Figure 3: Illustration of the EEG task.

EEG Task. Each participant will also be asked to complete a simple, computerized task while EEG is recorded from 64 scalp electrodes placed in accordance with the standard 10-5 system [61]. One challenge of EEG/ERP research is that measuring a single ERP requires that specific stimuli and/or stimulus conditions are repeated many times over the course of an experiment. In other words, measuring ERPs related to memory requires a task where images are repeated, and measuring ERPs related to aspects of cognition, like conflict processing and executive control, requires a separate task with a distinct set of stimuli and responses. Asking a participant

Captions, Labels, & Crossreferences

You may have noticed in the `wrapfigure` example above that the figure included a title and caption below it. This is yet another strength of LaTeX– it's ability to easily add captions to images and also use labels for crossreferencing throughout the document. Captioning images is when a brief description is included with the image. Crossreferences are pointers in the text of the document, instructing the reader where to look

(e.g., “see Figure 1”).

Captions

Captioning is very easy, just add the texttt
caption{Some caption} command inside the figure or wrapfigure environment, replacing “Some caption” with your desired text. The placement of the caption depends on where you place the command; if it’s above the `includegraphics` command then the caption will be on top of it, if it’s below then the caption will also be set below the figure. A simple example of a figure caption is illustrated below. You can do a more advanced management of the caption formatting. Check the further reading section for references.

Input

```
%Example of a caption
\documentclass{article}
\usepackage{graphicx}
\graphicspath{ {./images/} }

\begin{document}
The universe is immense and it seems to be
homogeneous,
in a large scale, everywhere we look at.

\begin{figure}[h]
\includegraphics[width=0.6\linewidth]{universe}
\captionA distant galaxy
\end{figure} \end{document}
```

Formatted Output

The universe is immense and it seems to be homogeneous, in a large scale, everywhere we look at.

Figure 2.2: A distant galaxy

LaTeX automatically numbers figures sequentially based on the order in which they appear in the document. When a figure environment (`\begin{figure}...\end{figure}`) is used in combination with the `\caption{...}` command, LaTeX assigns a number to the figure and displays it alongside the caption (e.g., “Figure 1”). If multiple figures are used, LaTeX increments the number with each new figure environment, ensuring consistent and automatic numbering throughout the document. This numbering also allows figures to be referenced using texttt\label{} and `\ref{}` commands, making cross-referencing easy and accurate. Figure numbering resets for each chapter if the document class supports chapters (like book or report), unless configured otherwise.

Labels and References

Figures, like many other elements in a LaTeX document (such as equations, tables, and plots), can be easily referenced within the text. To do this, include a `\label{fig:mylabel}` command inside the figure environment–typically right after the `\caption`. Later, you can refer to the figure by using that label. It is good practice to use a prefix (e.g., fig:) in your labels to distinguish figure references from those to tables (tab:), equations (eq:), or sections (sec:).

There are three common commands used for cross-referencing:

- `\label{fig:mesh1}` – Assigns a label to the figure.
- `\ref{fig:mesh1}` – Inserts the figure number (automatically generated and updated by LaTeX).
- `\pageref{fig:mesh1}` – Displays the page number where the figure appears.

Important: The `\caption` command is required in order for LaTeX to assign a number to the figure, which is necessary for referencing it with `\ref`.

An example of a simple document that uses \textttref to reference a figure in the text is illustrated below. The strength of using figure labels and cross-references is that all of the figure numbering is handled automatically by LaTeX, meaning that any changes that are made to the document will automatically lead

the appropriate changes being made in your document the next time you typeset it. For example, let's say you have seven figures in your document and, during revisions, you decide to remove Figure 1. Rather than having to manually re-number all of your figures and find/change all in-text references, LaTeX does all of that for you.

Input

```
%Cross-referencing a figure
\documentclass{article}
\usepackage{graphicx}
\begin{document}

\begin{figure}[h]
\centering
\includegraphics[width=0.25\textwidth]{universe.png}
\caption{A nice plot}
\label{fig:mesh1}
\end{figure}

As you can see in the figure \ref{fig:mesh1}, the
function grows near 0.
Also, on page \pageref{fig:mesh1} is the same example.
\end{document}
```

Formatted Output

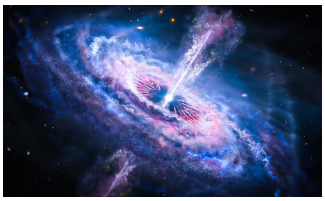


Figure 2.3: A nice plot

As you can see in the figure 2.3, the function grows near 0. Also, on page 42 is the same example.

2.5 Tables in LaTeX

Table Basics

Tables are common elements in most scientific documents, LaTeX provides a large set of tools to customize tables, change the size, combine cells, change the color of cells and so on. As was illustrated briefly in the discussion of LaTeX environments above, the `tabular` environment is the default LaTeX environment for creating tables. You must specify a parameter when beginning the `tabular` environment, designating the number of columns and defining how text will be displayed in each column. For example, the parameter `{c c c}` tells LaTeX that there will be three columns and that the text inside each one of them should be centered. By default, the values `c`, `l`, & `r` will instruct LaTeX to produce text that is either centered, left justified, or right justified respectively. When entering data into the tabular environment, columns are separated by the ampersand (&) character and rows are separated by the "new line" command (\\). A simple example of a table is illustrated below.

LaTeX

```
 \begin{tabular}{l c r}
one & two & three \\
four & five & six \\
seven & eight & nine
\end{tabular}
```

Formatted Output

one	two	three
four	five	six
seven	eight	nine

You can also add vertical and horizontal lines to the tabular environment using the "|" character and \hline commands (see Figure below).

LaTeX

```
 \begin{tabular}{|l| c| |r|}
\hline
one & two & three \\
four & five & six \\
\hline
seven & eight & nine
\end{tabular}
```

Formatted Output

one	two	three
four	five	six
seven	eight	nine

Whereas the tabular environment is the default for arranging content in tabular form, formatted tables, complete with titles and captions are created using a combination of the `table` and `tabular` environments. Tables are objects that are not part of the normal text, and are usually "floated" to a convenient place, like the top of a page. By default, tables will not be split between two pages. Like the `figure` environment, the `table` environment takes an optional placement parameter giving the author more control over the positioning of a table within the document. There are four options for the positioning of tables: (1) `h` = "here" - at the position in the text where the table environment first appears, (2) `t` = "top" - at the top of a text page, (3) `b` = "bottom" - at the bottom of a text page, and (4) `p` = "page" of floats - on a separate float page, which is a page containing no text, only floats. The body of the table is made up of whatever text, LaTeX commands, etc., you wish. For example, the `\caption` command allows you to title your table and the ommand can be used for cross-referencing (see example below).

LaTeX

```
 \begin{table}[h]
\caption{Reaction times over training in the experimental and control groups.}
\label{tab:table1}
\begin{tabular}{l|c|c|c}
& \textbf{Week 1} & \textbf{Week 2} & \textbf{Week 3} \\
\hline
Experimental & 1008.435 & 986.76 & 859.1 \\
Control & 996.23 & 901.67 & 1002.23 \\
\hline
\end{tabular}
\end{table}
```

Formatted Output

Table 2.1: Mean reaction in the experimental and control groups.

	Week 1	**Week 2**	**Week 3**
Experimental	1008.435	986.76	859.1
Control	996.23	901.67	1002.23

Sometimes it's desirable to make a row or column span several cells. The `multirow` and `multicolumn` packages provide this functionality (add \usepackage{multirow} and/or \usepackage{multicolumn} to the preamble). For a cell to span multiple rows, use the `\multirow` command with its three parameters:

\multirow{NUMBER_OF_ROWS}{WIDTH}{CONTENT}

Where the NUMBER_OF_ROWS is an integer indicating the number of rows to be spanned, WIDTH can be used to designate the width of the cell (or use to indicate the default width), and CONTENT is the desired cell contents.

If you want a cell to span multiple columns, use the `\multicolumn` command with its three parameters: `\multicolumn{NUMBER_OF_COLUMNS}{ALIGNMENT}{CONTENT}`

Of course it's also possible to combine the two features, to make a cell spanning multiple rows and columns. To do this, simply use the multicolumn command and add a `\multirow` command as the content. We then have to add another multicolumn statement for as many rows as we're combining. An example is illustrated below.

LaTeX

```
\begin{table}[h]
\caption{Reaction times over training in the experimental and control groups.}
\label{tab:table1}
\begin{tabular}{l| l| c| c| c|}
& & \multicolumn{3}{c}{Week} \\
& & \textbf{One} & \textbf{Two} & \textbf{Three} \\
\hline
\multirow{2}{*}{Group} & Experimental & 1008.435 & 986.76 & 859.1 \\
& Control & 996.23 & 901.67 & 1002.23 \\
\hline
\end{tabular}
\end{table}
```

Formatted Output

Table 2.2: Reaction times over training in the experimental and control groups.

		Week		
		One	**Two**	**Three**
Group	Experimental	1008.435	986.76	859.1
	Control	996.23	901.67	1002.23

Additional Table Formatting

You may have noticed while following the examples above, that some of the aesthetic elements of the table are somewhat lacking. In fact, the tables we've created to this point aren't even compatible with the requirements of APA style. This section will cover some additional formatting options so that you can generate tables appropriate for publication in APA style.

Recall that APA style tables use separators rather sparingly. In fact there are typically no vertical column separators and row separators often do not span the entire table. Unfortunately, the default table and tabular environments do not offer much flexibility when it comes to stylizing separators, however, there are other packages available to do this. One such package is `booktabs`, which can be used by adding the `\usepackage{booktabs}` command to the preamble. The `booktabs` package provides several additional commands for inserting rule lines in a table. Four of those commands are listed here and illustrated in the figure below:

1. `\toprule` = add a top rule to the table
2. `\midrule` = add a middle rule to the table
3. `\cmidrule{c1-c2}` = add a midrule spanning only integer columns c1 to c2
4. `\bottomrule` = add a bottom rule to the table

LaTeX

```
\begin{table}[h]
\caption{Reaction times over training in the experimental and control groups.}
\label{tab:table1}
\begin{tabular}{l l c c c}
\toprule
& & \multicolumn{3}{c}{Week} \\
& & \textbf{One} & \textbf{Two} & \textbf{Three} \\
\cmidrule{3-5}
\multirow{2}{*}{Group} & Experimental & 1008.435 & 986.76 & 859.1 \\
& Control & 996.23 & 901.67 & 1002.23 \\
\bottomrule
\end{tabular}
\end{table}
```

Formatted Output

Table 2.3: Reaction times over training in the experimental and control groups.

		Week		
		One	**Two**	**Three**
Group	Experimental	1008.435	986.76	859.156
	Control	996.2	901.67	1002.23

Another potential issue when using the default tabular environment occurs when using real (non-integer) numeric data. In most cases, it is considered good practice to (1) use a consistent number of decimal places for all numeric data and (2) to align numbers at the decimal point. Fortunately, both of these practices can be controlled using the `siunitx` package, which is loaded by adding `\usepackage{siunitx}` to the preamble.

After loading the `siunitx` package, it is also necessary to designate how you would like it to control the alignment. This is accomplished using the setup command `\sisetup`, which is also added to the preamble. An example of using the `sisetup` command to round tabular data to two decimal places is shown below.

```
\sisetup{
  round-mode          = places, % Rounds numbers
  round-precision     = 2,      % to 2 places
}
```

The siunitx package also makes available a new alignment parameter, "S", that can be used in the tabular environment when designating how cell contents should be aligned. This S parameter will instruct LaTeX to align numbers at their decimal point and center the headers! Use of the `siunitx` package and S parameter is illustrated below.

LaTeX

```
\usepackage{siunitx}
\usepackage{booktabs}
\begin{table}[h]
\caption{Reaction times over training in the experimental and control groups.}
\label{tab:table1}
\begin{tabular}{l l C C C}
& & \multicolumn{3}{c}{Week} \\
\cmidrule{3-5}
& & {One} & {Two} & {Three} \\
\cmidrule{3-5}
\multirow{2}{*}{Group} & Experimental & 1008.435 & 986.76 & 859.1 \\
& Control & 996.23 & 901.67 & 1002.23 \\
\bottomrule
\end{tabular}
\end{table}
```

Formatted Output

Table 2.4: Reaction times over training in the experimental and control groups.

		Week		
		One	Two	Three
Group	Experimental	1008.44	986.76	859.10
	Control	996.23	901.67	1002.23

2.6 APA Style in LaTeX

Most journals in the social and behavioral sciences require manuscripts to be formatted in compliance with the American Psychological Association's Publication Manual, which is updated periodically. In October 2019, the American Psychological Association (APA) introduced the 7th edition of the APA Publication Manual, which replaces the 6th edition published in 2009. There were a number of changes in the 7th edition including the way in which citations and references are displayed as well as to the way section headers are formatted.

When changes like this are made, it is usually the author's responsibility to track down all the relevant changes and implement them manually. Not so when using LaTeX! Recall that every LaTeX document begins with the texttt\documentclass command. The general form of this command is `\documentclass[options]{class}`, where the class parameter specifies the name of a special "class" (.cls) file to use for the document. It is also possible to create your own class file, as is often done by journal publishers, who can simply provide you with their class file, which tells LaTeX how to format your content. But there is also a class that has been developed to mimic APA style. The options parameter customizes the behavior of the document class. An example that could be used to start your document using the `apa7` document class (APA-style 7th edition) is shown below:

```
\documentclass[12pt,a4paper,man,natbib]{apa7}
```

Note that several optional parameters have also been passed to the `\documentclass` command. The "12pt" parameter sets the font size to 12 points. The "a4paper" parameter determines the paper size. A4

is the standard size of typing paper adopted by the International Standards Organization. It measures 210 mm wide and 297 mm longl (about 8 1/4 x 11 3/4 inches). It is used in most countries of the world, except the US and some neighboring countries where letter-size paper (8 1/2 x 11 inch) is used. The "man' option formats the document as a typical manuscript suitable for submission to most journals requiring the APA format. Other options include "jou" (journal article format) and "doc" (standard LaTeX document). Finally, the `natbib` option designates that the "`natbib`" package will be used for formatting citations & references.

Writing Your APA Style Paper

The Preamble

So far, the preamble–the part of the document between the `\documentclass` and `\begin{document}` commands–has mostly been used to load additional packages, but there are other important components of the preamble that are used by the `apa7` document class. These commands are used to establish important components of the paper like the title, the abstract, the running head, and the author's name(s). Below is short list of some of the most commonly used commands, but there are many optional commands that are described in the `apa7` documentation.

- `\title{document-title}`: The title of the document
- `\shorttitle{short-title}`: A shortened version of the title (used for page headers)
- `\author{author(s)}`: Author name(s) (see apa7 documentation for more info)
- `\affiliation{affiliation(s)}`: Author affiliation(s)
- `\abstract{abstract-text}`: The abstract of the article
- `\keywords{keywords}`: Keywords (typeset after the abstract)
- `\authornote{author-note}`: The Author Note, containing contact information, acknowledgements, etc.

Maketitle

All of the information in the preamble is used by the `\maketitle` command to the title page, page headers, author list, author affiliations, author Note (if provided), and abstract according to whether `jou`, `man`, or `doc` mode has been specified. This command should be on the line after `\begin{document}`, with the first line of the text of your document immediately following.

2.7 Headings

Once you have set up the important elements of the document in the preamble and initiated the document formatting using `\maketitle`, It is time to write your document. APA style dictates that manuscripts be divided into several major sections including the Introduction, Method, Results, Discussion, References, Tables (if any), Figures (if any), and Appendices (if any). APA style also dictates the specific formatting for the headers demarking these sections as well as any subsections. Fortunately, the `apa7` document class already contains instructions to make headings consistent with APA style. Heading levels 1-3 are designated using the LaTeXcommands illustrated below.

LaTeX	Formatted Output

`\section{Method}`
`\subsection{Participants}`
`\subsubsection{Exclusion criteria}`
	Method
	Participants
	Exclusion criteria

If needed, heading levels 4 and 5 need to be created manually using the boldface (`\textbf`) and italics (`\textit`) commands.

2.8 Reference Management

Although a number of options now exist for reference management in WYSIWYG word processors like Microsoft Word, Google Docs, and Libre Office, easy automatic reference management remains a strength of LaTeX. There are, at minimum, three components to reference management in LaTeX: (1) storing a library of references, (2) creating inline citations, and (3) adding the full bibliographic references at the end of the document.

Reference Library

BibTeX is the default reference manager in LaTeX and automates most of the work involved in managing references for use in LaTeX files. You need to type each reference only once, and your citations and reference list are automatically formatted consistently, in a style of your choosing. Like most things in LaTeX, the reference library is stored in a plain text file, usually given the extension .bib (e.g., MyLibrary.bib). The reference library (.bib file) is a simple text file defining the properties of each reference. You can type this information yourself, or collect it automatically from the web! Note that many GUI-based reference managers like Zotero, Mendeley, RefWorks, etc. include options for exporting your library in BibTex format.

An example of the contents of a .bib file is illustrated for book and article references below. Note that the very first element of a BibTeX reference is a user-defined label (e.g., Arrow61) that must be unique to each reference in the file and will be used when generating inline citations.

```
@article{Arrow61,
   author={Arrow, Kenneth J. and Leonid Hurwicz and Hirofumi Uzawa},
   title={Constraint qualifications in maximization problems},
   journal={Naval Research Logistics Quarterly},
   volume={8},
   year=1961,
   pages={175-191}
}
@book{Charalambos94,
   author* = {Charalambos D. Aliprantis and Kim C. Border},
   year = {1994},
   title = {Infinite Dimensional Analysis},
   publisher = {Springer},
   address = {Berlin}
}
```

Inline Citations

Basic Commands

When writing the body of your document, citations to your references are accomplished using variations of the `\cite` command. The `natbib` package has two basic citation commands, `\citet` and `\citep` for textual and parenthetical citations, respectively. There are also starred versions of these commands, `\citet*` and `\citep*` that force LaTeXto print the full author list even when there are many authors. All of these may take one or two optional arguments to add some text before and/or after the citation (see Examples below).

LaTeX	Formatted Output
`\citet {jon90}`	Jones et al. (1990)
`\citet [chap.~2]{jon90}`	Jones et al. (1990, chap. 2)
`\citep {jon90}`	(Jones et al., 1990)
`\citep [chap.~2]{jon90}`	(Jones et al., 1990, chap. 2)
`\citep [see][]{jon90}`	(see Jones et al., 1990)
`\citep [see][chap.~2]{jon90}`	(see Jones et al., 1990, chap. 2)
`\citet *{jon90}`	Jones, Baker, and Williams (1990)
`\citep *{jon90}`	(Jones, Baker, and Williams, 1990)

Table 2.5: Examples of `natbib` citation commands and their formatted outputs.

Multiple Authors

Multiple citations may be made by including more than one citation key with the `\cite` command (see Examples below).

LaTeX	Formatted Output
`\citet {jon90,jam91}`	Jones et al. (1990); James et al. (1991)
`\citep {jon90,jam91}`	(Jones et al., 1990; James et al., 1991)
`\citep {jon90,jon91}`	(Jones et al., 1990, 1991)
`\citep {jon90a,jon90b}`	(Jones et al., 1990a,b)

Table 2.6: Examples of `natbib` citation commands with multiple references.

Partial Citations

In author–year schemes, it is sometimes desirable to be able to refer to the authors without the year, or vice versa. This is provided with the extra commands illustrated below.

LaTeX	Formatted Output
`\citeauthor {jon90}`	Jones et al.
`\citeauthor *{jon90}`	Jones, Baker, and Williams
`\citeyear {jon90}`	1990
`\citeyearpar {jon90}`	(1990)

Table 2.7: Examples of `natbib` citation commands for authors and years.

Bibliographic References

In APA style, all the sources you cited throughout the text of your paper (inline citations) are listed together, and more fully, in the References section, which comes after the main text of your paper. Each reference should appear in alphabetical order and be formatted with what is called a hanging indent. Fortunately, no more needs to be said about the formatting of bibliographic references because the `natbib` package takes care of it. Assuming that your MyLibrary.bib file, containing your reference library, is in the same folder

as your main .tex file, then In order to produce your References section of a manuscript, simply type the following just before the \end{document} command, which designates the end of the document.

```
\bibliography{MyLibrary.bib}
```

2.9 Drawing with TikZ

What is TikZ?

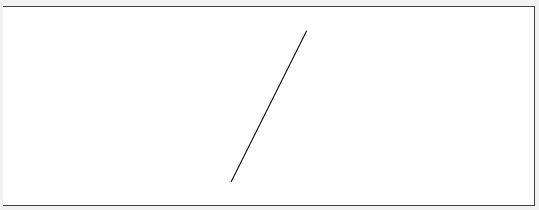
TikZ is a set of higher-level macros that use PGF for producing vector graphics (e.g., technical illustrations and drawings) from a geometric/algebraic description, with standard features including the drawing of points, lines, arrows, paths, circles, ellipses and polygons. The PGF/TikZ interpreter can be used from within the LaTeXenvironment. The TikZ project has been under constant development since 2005, with most of the development being done by the program's original author, Till Tantau.

In order to use TikZ you must include the package using the \usepackage{tikz} command inside the preamble of your LaTeX document. In the body of the document, you can create a new picture using \begin{tikzpicture} and \end{tikzpicture}. Everything in between these two commands is interpreted as a TikZ image.

TikZ is a very large program which can do lots of things. You will find commands to draw hierarchical trees, to draw lots of different types of shapes, to do some elementary programming, to align elements of a picture in a matrix frame, to decorate nodes, to compute the intersections of paths, etc. The main message is that this is just a brief introduction. You will find the complete PGF/TikZ manual at https://drive.google.com/file/d/1t9pvhhqOzJ9WO6PTVYqso_4-IOphU7AM/view?usp=sharing.

Drawing Lines

Let's start with the \draw command, which tells LaTeXto create a path between given coordinates.Coordinates are written as (x,y), where x is the horizontal position and y is the vertical position, usually in centimeters by default.

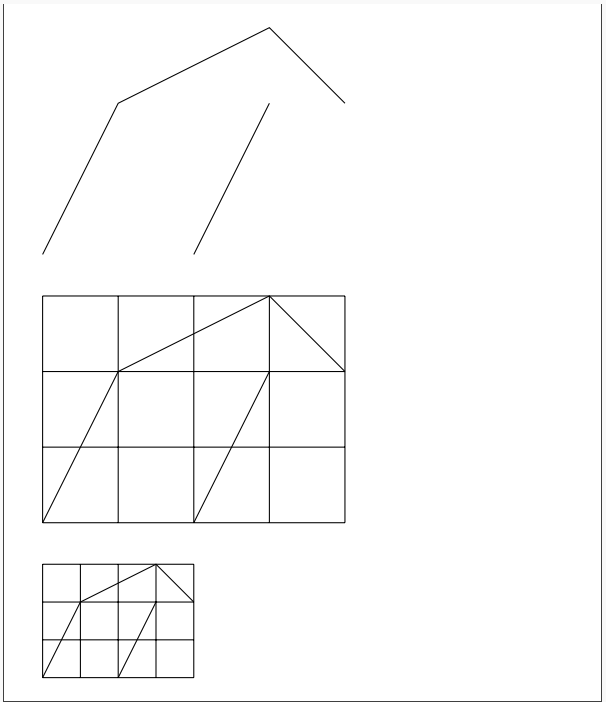
LaTeX	Formatted Output
\begin{tikzpicture} \draw (0,0) -- (1,2); \end{tikzpicture}	

Multiple lines can be drawn in sequence (see examples below). When drawing multiple lines, it is sometimes easier when you can see the grid used for drawing. You can draw a grid using the grid() command. Rather than changing all of the points in your drawing, use the scale option to change the overall size of the tikzpicture environment.

LaTeX

```
% draw multiple lines
\begin{tikzpicture}
\draw (0,0);
\draw (0,0) -- (1,2) -- (3,3) -- (4,2);
\draw (2,0) -- (3,2);
\end{tikzpicture}

% adding the grid command can make it easy to
see the points for drawing
\begin{tikzpicture}
\draw (0,0) grid (4,3);
\draw (0,0) -- (1,2) -- (3,3) -- (4,2);
\draw (2,0) -- (3,2);
\end{tikzpicture}

% Use the scale option to change the size of
your drawing
\begin{tikzpicture}[scale=.5]
\draw (0,0) grid (4,3);
\draw (0,0) -- (1,2) -- (3,3) -- (4,2);
\draw (2,0) -- (3,2);
\end{tikzpicture}
```

Formatted Output



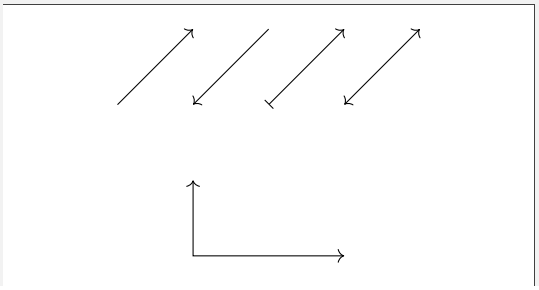
There are also a number of options that can be passed to the `draw` command modifying the:

Path Endpoints

LaTeX

```
% Line endpoints example
\begin{tikzpicture}
\draw [->] (0,2) -- (1,3);
\draw [<-] (1,2) -- (2,3);
\draw [|->] (2,2) -- (3,3);
\draw [<->] (3,2) -- (4,3);
\draw [<->] (1,1) -- (1,0) - (3,0);
\end{tikzpicture}
```

Formatted Output



Path Color, Style, & Weight

LaTeX Input

```
\begin{tikzpicture}
% Color, style, & weight examples
\draw [->, thin, red] (0,2) -- (1,3);
\draw [<-, thick, red] (1,2) -- (2,3);
\draw [|->, ultra thick, dashed] (2,2) -- (3,3);
\draw [<->, line width=0.1cm, blue] (3,2) -- (4,3);
\draw [<->, line width=0.5mm, cyan, dotted] (4,2) -- (5,3);
\end{tikzpicture}
```

Formatted Output

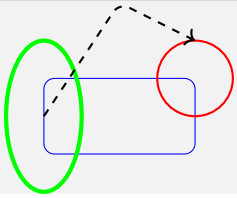
Drawing Shapes

You are not limited to straight lines in TikZ. You can draw shapes like rectangles, circles, polygons and stars, etc. The following example depicts only a few. You can also add the "rounded corners" option to lines and shapes to improve their appearance.

LaTeX Input

```
\begin{tikzpicture}
\draw [blue, rounded corners] (0,0) rectangle (2,1);
\draw [red, thick] (2,1) circle [radius=0.5];
\draw [green, ultra thick] (0,0.5) ellipse [x radius = 0.5cm, y radius = 1cm];
\draw [->, thick, dashed, rounded corners] (0,.5) -- (1,2) -- (2,1.5);
\end{tikzpicture}
```

Formatted Output



You can also fill shapes and paths (when they are closed). Notice the difference between the first two squares in the example below. In the first red square, we indicated that the fill color should be red, but did not give a color for the `\draw` line, so it is black. In the second red square, we indicated a color option that is also red, so the line produced by `\draw` matches the fill color.

LaTeX Input

```
\begin{tikzpicture}
\draw [fill=red,ultra thick] (0,0) rectangle (1,1);
\draw [fill=red,ultra thick,red] (2,0) rectangle (3,1);
\draw [blue, fill=blue] (4,0) -- (5,1) -- (4.75,0.15) -- (4,0);
\draw [fill] (7,0.5) circle [radius=0.1];
\draw [fill=orange] (9,0) rectangle (11,1);
\draw [fill=white] (9.25,0.25) rectangle (10,1.5);
\end{tikzpicture}
```

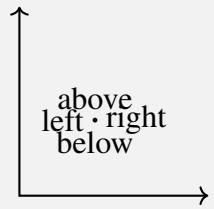
Formatted Output

Labeling Coordinates in a TikZ Picture

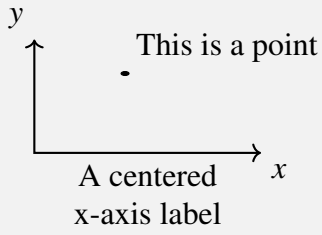
When creating figures with TikZ, you often want to add labels or annotate parts of your drawing — for example, naming points, adding descriptions, or marking axes. This is easy to do with the `\node` command. Think of `\node` as a way to place text (or other content) at a specific coordinate or relative position inside your TikZ picture. Unlike `\draw`, which creates lines and shapes, `\node` creates a "container" for text or symbols (see examples below).

LaTeX **Formatted Output**

```
\begin{tikzpicture}
\draw [thick, <->] (0,2) -- (0,0) -- (2,0);
\draw[fill] (1,1) circle [radius=0.025];
\node [below] at (1,1) {below};
\node [above] at (1,1) {above};
\node [left] at (1,1) {left};
\node [right] at (1,1) {right};
\end{tikzpicture}
```




```
\begin{tikzpicture}[xscale=3, yscale=1.5]
\draw [thick, <->] (0,1) -- (0,0) -- (1,0);
\node [below right] at (1,0) {$x$};
\node [above left] at (0,1) {$y$};
\draw [fill] (.4,.7) circle [radius=.5pt];
\node [above right, align=left] at (.4,.7) {This is a point};
\node [below, align=center] at (.5,0) {A centered \\x-axis label};
\end{tikzpicture}
```



Path Diagrams

Path diagrams are frequently used in the Psychological Sciences to represent relationships between variables (e.g., correlation, mediation, moderation), and to illustrate statistical models and theoretical frameworks. When creating path diagrams, using LaTeX with TikZ offers several advantages over point-and-click software like PowerPoint. TikZ produces vector-based graphics that integrate seamlessly into your LaTeX document, ensuring consistent fonts, mathematical notation, and styles throughout your work without the need to manually match formatting. Unlike PowerPoint, where precise positioning can be tedious and prone to misalignment, TikZ allows you to define elements and their relationships using exact coordinates or relative positioning commands, guaranteeing reproducibility and pixel-perfect accuracy. This approach is especially valuable for academic work, where diagrams often need to be updated, scaled, or reused — with TikZ, you can modify a few lines of code rather than rebuilding the figure from scratch. Furthermore, because TikZ diagrams are text-based, they are version-control-friendly, enabling collaboration and easy tracking of changes. In short, LaTeX+TikZ combines precision, scalability, and reproducibility in a way that point-and-click tools cannot match, making it the preferred choice for professional, publication-quality path diagrams.

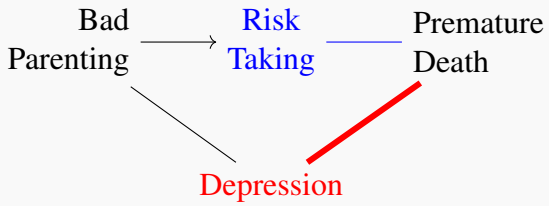
In TikZ, the `\path` command is the most general way to describe a sequence of points, curves, and shapes — but by itself, it doesn't actually produce any visible output. Think of `\path` as defining a route or shape, but LaTeXhas not yet been told what to do with it... however, when `\path` is combined with the `draw` argument or provided a color argument, its behavior is similar to the `\draw` command. `\path` is sometimes preferred because it is more general: it can define complex geometric paths without immediately rendering them, allowing those paths to be reused, transformed, or drawn/fill-stroked later in different ways. This separation of path definition from path rendering can make code more modular, easier to maintain, and better suited for situations where the same geometry needs to be styled multiple times.

The example below illustrates how nodes and paths can be combined to produce flexible diagrams in your LaTeXdocument.

LaTeX Input

```
\begin{tikzpicture}
\node[align=right](1)at(0,0){Bad\\Parenting};
\node[blue,align=center](2)[right=of 1]{Risk\\Taking};
\node[align=left](3)[right=of 2]{Premature\\Death};
\node[red](4)[below=of 2]{Depression};
\path[->](1)edge(2);
\path[blue](2)edge(3);
\path[red,line width=.8mm](4)edge(3);
\path(1)edge(4);
\end{tikzpicture}
```

Formatted Output



You may have noticed that there are many different options that can be passed to the `\node` and `\path` commands. Rather than explain them all, let's just consider the TikZ command below:

```
\node [blue, align=center](2)[right=of 1]{Risk\\Taking};
```

This command places a labeled node in a TikZ picture. Let's break down each part:

`\node` This is the basic command used to create a node, which is a point in the graphic that can contain text or shapes.

`[blue,align=center]` The first optional argument specifies styling options, in this case:

- `blue` — sets the color of the text (and the node border if drawn) to blue.
- `align=center` — centers multi-line text inside the node.

`(2)` This assigns the *name* `"2"` to the node, which allows referencing this node later when drawing edges or positioning other nodes.

`[right=of 1]` The second optional argument (another set of brackets) is a positioning directive:

- `right=of 1` — places this node to the right of the node named `1`.
- Note that this requires the `positioning` library, loaded with `\usetikzlibrary{positioning}` in the preamble.

`{Risk\\Taking}` This is the node content enclosed in curly braces:

- `Risk\\Taking` — the double backslash creates a line break, so the text is displayed on two centered lines.

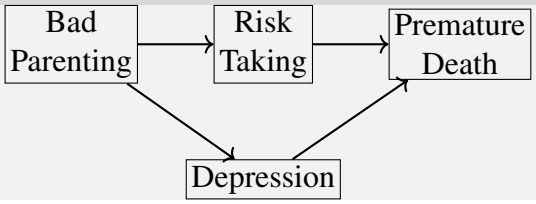
In summary, when used inside a `tikzpicture` environment (with a node named `1` defined), this will create a blue-colored, two-line label positioned directly to the right of the node named `1`.

In practice of course, it is usually the case that nodes and arrows will all have the same (or similar) formatting. Below is and example of a simple path diagram illustrating the prediction that both risk taking and depression may be mediators of the relationship between bad parenting and an increased likelihood of premature death.

LaTeX Input

```
\begin{tikzpicture}
\node[draw, align=center, minimum width=0.7cm, inner sep=2pt](1) at (0,0) Bad\\Parenting;
\node[draw, align=center, minimum width=0.7cm, inner sep=2pt](2)[right=of 1]Risk\\Taking;
\node[draw, align=center, minimum width=0.7cm, inner sep=2pt](3)[right=of 2]Premature\\Death;
\node[draw, align=center, minimum width=0.7cm, inner sep=2pt](4)[below=of 2]Depression;
\path[->,thick](1)edge(2);
\path[->,thick](2)edge(3);
\path[->,thick](4)edge(3);
\path[->,thick](1)edge(4);
\end{tikzpicture}
```

Formatted Output

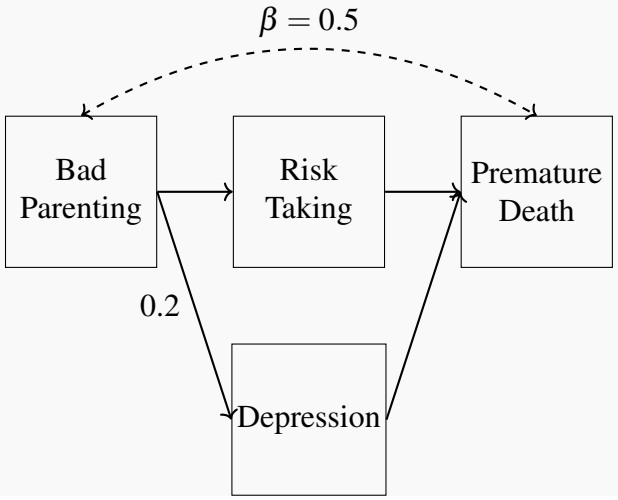


Again, a major advantage of using TikZ in this context is the precision with which elements can be controlled. For example, the text of each box is perfectly equally padded with space within the box and all lines leading from/to the boxes attach at exactly the same points (predetermined by TikZ), and we have only scratched the surface! Below is a more complex example that adds model estimates from the regression and some curved lines.

LaTeX Input

```
\begin{tikzpicture}
\node [draw, align=center, minimum width=2cm, minimum height=2cm, inner sep=2pt] (1) at(0,0)Bad\\Parenting;
\node [draw, align=center, minimum width=2cm, minimum height=2cm, inner sep=2pt] (2) [right=of 1] {Risk\\Taking};
\node [draw, align=center, minimum width=2cm, minimum height=2cm, inner sep=2pt] (3) [right=of 2]{Premature\\Death};
\node [draw, align=center, minimum width=2cm, minimum height=2cm, inner sep=2pt] (4) [below=of 2]{Depression};
\path[->,thick](1)edge(2);
\path[->,thick](2)edge(3);
\path[->,thick](4.east)edge(3.west);
\path[->,thick](1.east)edge node[pos=0.5, left=1pt] {0.2}(4.west);
\path[<->,thick,dashed](1.north)edge[bend left=30] node[above] {$\beta=0.5$}(3.north);
\end{tikzpicture}
```

Formatted Output



There are a few things to note about the example above:

- The `minimum width` and `minimum height` arguments were used to make the box size constant across nodes.
- Two of the paths include the `node` command that is used to add text to a path. Note that the `node` command also takes arguments used to position the text appropriately.
- The math environment (e.g., $...$) can be used to insert equations or Greek letters along paths.

Experimental Design Layouts and Task Schematics (UNDER CONSTRUCTION)

Experimental design layouts are crucial diagrams in psychological science for visually organizing and representing the structure of experiments, including factors, conditions, and participant groups. These layouts help clarify complex designs such as within-subjects, between-subjects, or mixed factorial experiments by depicting how participants experience different conditions and how variables are manipulated. By mapping these elements graphically, researchers can better plan studies, communicate methodology, and understand interactions among factors. TikZ allows precise control over node placement and connections to illustrate experimental setups clearly and professionally.

Experimental Design Layouts

Putting it All Together

The previous sections introduce all of the commands needed to create an entire LaTeXdocument in APA style. For those new to markup languages like LaTeX, creating your first complete document can seem like a daunting task. One way to simplify the process is to organize each section of the document into separate files. For example, many LaTeXusers will place the `documentclass` command and the preamble into a file called `main.tex`. The introduction, method, results, and discussion would each be contained in separate `.tex` files an included in the `main` document using the `\include{doc.tex}` command. Visit https://www.overleaf.com/read/sdyytvnmjfst in any browser to see a complete example of a LaTeXmanuscript following the APA 7th edition style.

2.10 Tips & Tricks

Placing Figures Inline with Text

Before the release of the APA 7th edition guidelines, tables and figures were expected to appear at the end of the document, each on their own page. This remains the default behavior of the apa7 document class. However, APA 7 also allows authors to place figures and tables inline with the text, on the same page where they are first mentioned. This is often preferable when you want the reader to immediately see the figure or table without flipping pages.

To override the default behavior, add the `floatsintext` option to your `\documentclass` line. For example:

```
\documentclass[a4paper,man,natbib,floatsintext]{apa7}
```

Once this is done, you can place your figure or table environments exactly where you want in the flow of the text, and LaTeX will try to keep them close to where they are introduced.

By default, LaTeX tries to keep the caption with the corresponding figure or table, but if there's a page break issue or a lot of surrounding text, they can sometimes get separated or placed awkwardly. To force them to stick together:

Always put the `\caption{...}` inside the float environment (figure or table). For example, using:

```
\begin{figure}[htbp]
  \centering
  \includegraphics[width=0.7\textwidth]{myfigure}
  \caption{An example figure that will always keep its caption.}
  \label{fig:example}
\end{figure}
```

will prevent page breaks between caption and table/figure description

Another option is to add `\usepackage{caption}` to the preamble and use the skip option to control spacing, which also helps keep things tight:

```
\usepackage[skip=5pt]{caption}
```

If LaTeXstill tries to split up the elements of a figure or table, wrap the float in a minipage to lock the figure/table and caption into one unbreakable unit:

```
\begin{figure}[H] % requires \usepackage{float}
  \begin{minipage}{\linewidth}
    \centering
    \includegraphics[width=0.7\textwidth]{myfigure}
    \caption{Caption that will never be separated from the figure.}
    \label{fig:example}
  \end{minipage}
\end{figure}
```

For tables specifically: If your table has a description below it (common in APA), include the description inside the float as normal text after the `\caption{...}`. This way LaTeXconsiders the whole thing one object.

```
\begin{table}[htbp]
  \centering
  \caption{Participant demographics}
  \begin{tabular}{ll}
    \hline
    Age & Mean (SD) \\
    \hline
    25 & (3.2) \\
    \hline
  \end{tabular}
  \vspace{2mm}
  \footnotesize\textit{Note.} Data represent means and standard deviations.
\end{table}
```

2.11 Additional Resources

An advantage of LaTeXis that it has been around for a long time and has a huge, global user base. That means that most questions you have can be answered using any internet search engine. In fact I recommend using the internet as much as possible when you encounter questions and/or problems. However, if you are looking for some additional readings/references, below is a short list of some good resources:

- LaTeXon Wikibooks: https://en.wikibooks.org/wiki/LaTeX
- The Not So Short Introduction to LaTeX2E by Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl: https://archive.org/details/lshort
- Guide to LaTeX(4th edition) by Helmut Kopka and Patrick W. Daly: http://www.worldcat.org/oclc/844889428
- Overleaf LaTeXTips: https://www.overleaf.com/help/category/latex_tips
- Overleaf Video Tutorials: https://www.overleaf.com/help/category/video_tutorials

2.12 Assignment

Overview

In this assignment, you will learn to create a professionally formatted psychological research report using LaTeX, focusing on the formatting guidelines of the American Psychological Association (APA) 7th edition. Learning to use LaTeX for your research reports not only enhances your technical skills but also helps you produce clean, consistent, and reproducible documents that are widely valued in the academic community.

Instructions

To complete this assignment, you will use the APA template in the `Latex_and_Git` directory of your personal branch of the forked GitHub repository found at https://github.com/pdkieffaber/PSYC672. In order to get credit for this assignment, you should:

1. Modify the document preamble (main.tex) to be consistent with your project.
2. Modify the Introduction section (intro.tex) to include a brief 1-2 page background related to your current or anticipated project that includes at least two references to relevant published work. The introduction should include a clear statement of the primary hypothesis and end with a brief description of how that hypothesis will be addressed in the current study.
3. Modify the Method section (method.tex) to fit with your research methods. Bullet points are fine.
4. Modify the Results section (results.tex) to fit with your data analysis. Bullet points are fine.
5. Modify the Discussion section (discussion.tex) to fit with your results (actual or expected). Just a couple sentences about how your research will contribute to the broader literature is fine.
6. Push your modifications to your personal branch of the PSYC672 GitHub repo.