Fragile Guide

Release "0.0.1"

Guillem Duran, Vadim Markovtsev

Jun 22, 2021

OUR WORKFLOW

Welcome to the Fragile Tech company guide. The goal of this repository is to share knowledge across the company.

All the documentation will be written in Markdown, and updated following our *Git workflow*.

CHAPTER

ONE

RESOURCES FOR LEARNING MARKDOWN

- Markdown cheat sheet
- Another markdown cheat sheet
- ProWritingAid A grammar checker, style editor, and writing mentor. You can use it to improve your writing skills.

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Workflow Overview

2.1.1 Introduction

This section describes the Agile methodologies that we use to develop our projects. You will find here a detailed description of the tools and processes that we use to collaborate on Open Source projects.

This topic is explained from scratch, assuming that the reader is not familiar with Git nor GitHub. Even that we try to make it as beginner-friendly as possible, developing software is a complex process that takes time to learn and master.

2.1.2 TLDR

We use Git for version control, and we host our projects on GitHub. We communicate asynchronously using Issues, Pull Requests, and code reviews to work collaboratively on the repositories. We manage our GitHub notifications with Octobox, and we use Zoom and Telegram to coordinate a couple of times a week.

2.1.3 Goals and use cases

Our workflow helps us work asynchronously and maximize our productivity. Implementing software development best practices allows us to keep track of all our work, and increase the overall quality of our projects.

Thanks to that we can speed up our development cycle, reduce the number of mistakes in our code base, and fix errors quickly.

A nice side effect of this workflow is that it not only allows the community to keep track of how our code-base evolves, but also makes accessible the whole development and decision-making process behind the released code.

2.1.4 Pros & Cons

- **Pros**
  - Efficient and asynchronous collaboration among developers
  - Complete understanding of the evolution of our code base
  - Documents the design and decision-making processes of the projects
  - Consistent across different projects
  - Agile-friendly and completely transparent
  - Relies on few tools that are free

- **Cons**
  - Complex process that takes time to learn
  - Rough learning curve
  - Taking shortcuts will increase the technical debt
  - Errors and issues with the tooling or processes slow down development a lot.

2.1.5 Requirements and background

There are no hard requirements to understand this topic, but knowing about the following topics is helpful:

- Agile development
- Git and version control
- How to use GitHub
- Software development and code reviews
- Basic knowledge of GNU/Linux

2.1.6 Learning

1. We recommend understanding what Git is first. *These slides* provide a gentle introduction to Git, and the main advantages that Git can offer. After that, reading the *Git handbook* can help you understand better what you can do with Git.
2. Git is learned by practise, go through the interactive labs to learn a lot very quickly.
   - *Fundamentals of Git*
   - *Learn about Git branching*
   - *Free Git course*
3. Learn about GitHub, and what are Issues and Pull requests.
4. Take a look at the *how to* document to learn about its features.
5. Learn the fundamentals of code review.
6. Practise by opening issues and PRs in this repo.

2.1.7 Reference

We recommend using the *Git Cheat Sheet* for looking up commands, and the *Pro Git* book to look up the most arcane features of git.

2.2 Introduction to Git

Git is a powerful tool that permits parallel and asynchronous work while maximizing productivity and collaboration among team members. Despite the countless benefits Git offers, it requires time and dedication to learn and master. In this section, we will try to cover and define the most general Git concepts that may lead to misleading conclusions. Having an accurate understanding of these terms will notably facilitate your learning path. This section summarizes the content of different reviews and guides (highlighted throughout this page); we strongly encourage practicing the core Git utilities using *this lab*, which includes a beginner-oriented tour about the basic concepts you are working on.

2.2.1 The three stages of Git

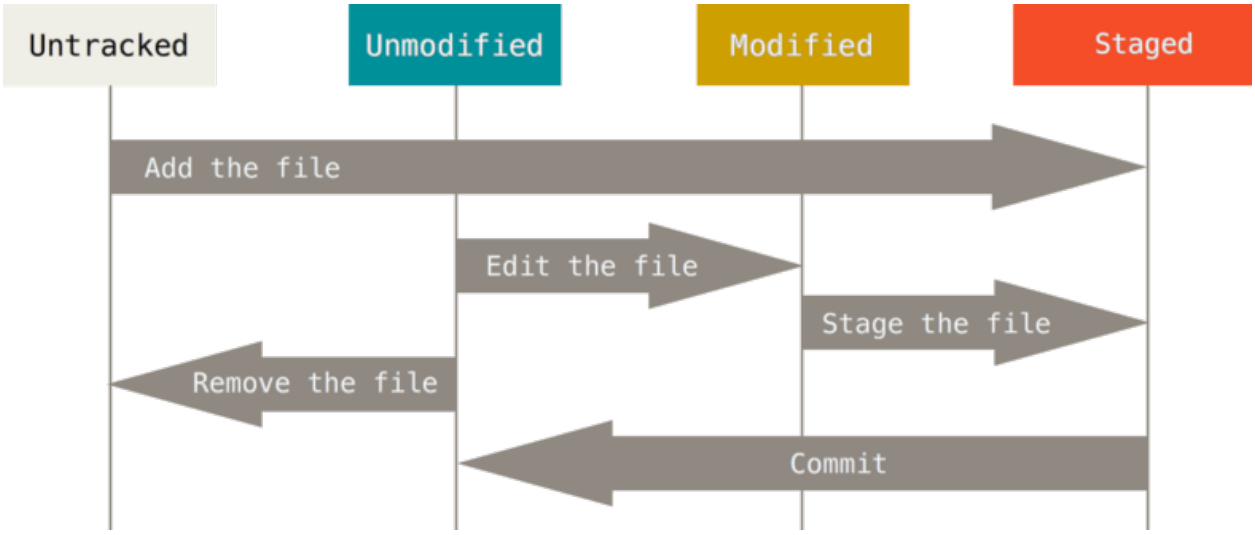

Fig. 1: Three stages of Git.

Every file in a Git repository goes through three stages: **modified**, **staged** and **committed**.

- *Modified* is the stage where you can add new features (or *modifications*) to the project. Changes are applied in the working directory (the Git tree in sync with your local machine), leaving unchanged the original code.
- The second step is tracking modified files. The command `git add [files]` updates the Git index using the given `files` and pushes them to the *staging* area. Files included in this "space" are monitored by Git, notifying when new changes are applied. This area is the previous step to commit your latest modifications: it stores the changes that you want to include in the next snapshot.
- The last step consists in *committing* your modifications. `git commit` records a snapshot of the changes kept in the staging area while creating a new timestamp in the history. Adding the option `-m` allows you to pass a commit message. Git maintains a record of all the commits made, generating a "timeline" with committed changes ordered in time. Sitting in a particular commit, all previous ones are called *ancestors*, usually designated with arrows in descriptive illustrations.

Remember

Files in a working directory can be in two states: **tracked**, files already present in the previous snapshot or added to the staging area, or **untracked**, files not present in the last commit and therefore not staged. Git will not include untracked files in the following historical snapshot until you explicitly add them. Git notices new modifications added to the tracked files.

Resources

You will discover more information and material about these concepts in the following links:

- An introductory *tutorial* about the saving process in Git.
- A (maybe too comprehensive) *complete description* of the recording process in Git.
- Tutorial focused on the Git command `reset`, the first part contains an enlightening *description* of Git's internal state management system.

2.2.2 Commit, branches and heads

As previously mentioned, `git commit` is a daily command used to save the relevant changes in our repository. Git preserves a history of which commits were made and when; new commits arise from older commits (called parents or ancestors), which are used as a basis to build new commitments.

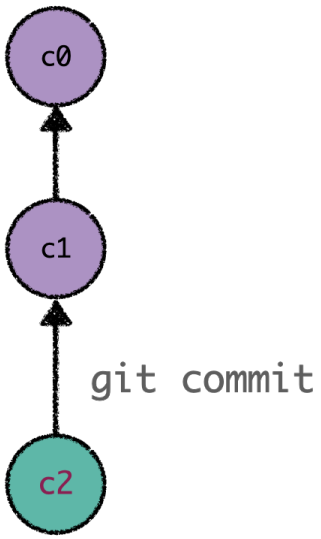

Fig. 2: New committed changes are based off its ancestors.

Most VCS's offer the possibility of creating secure "rooms" where you can play and test new inclusions to your model. Contrary to other VCS's (where these environments copy files from directory to directory), *Git branches* are **pointers** to commits kept in Git history. Branches are not new isolated copies of your project files (like a container for your commits), but they are references to specific commits. Developers usually represent Git branches as independent ramifications (or bifurcations) from the main development line. They are built using the `git branch` command or `git checkout -b`.

Consider the scenario illustrated in the figure below, where you have committed a recent change. When using the command `git checkout -b new_feature` (or `git branch new_feature`), you are creating a new pointer labeled `new_feature` that refers to the last commit while maintaining the repository **unchanged**. That being said, a natural question may arise: what is the usefulness of creating a new reference? Although at first glance this operation might seem useless, branches reach their full potential as we generate new commits.

Moving to the branch that we created (`new_feature`), subsequent commits (for example, *c3*) will be referred to by the latter. Once we have finished our task, we return to our principal branch (`main`) by using the command `git checkout main` (remember, branches are simply references; as illustrated in the figure, `main` is a tip pointing to the commit *c2*). Sitting on `main`, we will develop a **ramification** when committing a new change (in this case, called *c4*). Despite being conceived by the same commit *c2*, *c3* and *c4* are disconnected entities that "exist" in different branches. If we come back to `new_feature`, we can make further commits that depend neither on *c4* nor on `main`. As mentioned at the beginning, we are able to write new code without affecting the main development line.

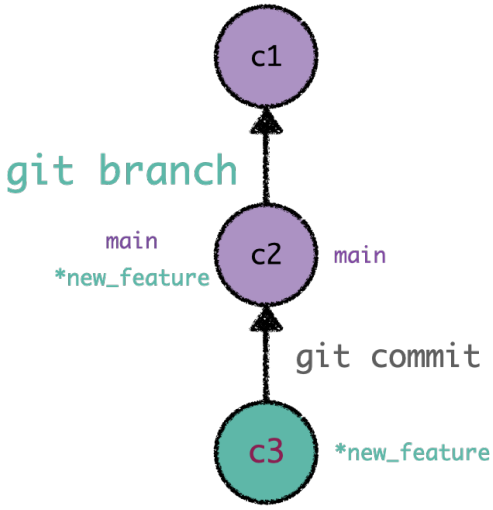

Fig. 3: Branches are **pointers** to specific commits in Git history. Sitting on the commit *c2* (which is referred to by `main`), we can create a new branch named as `new_feature`. All subsequent commits (as *c3*) will be referred to by this newly created branch.

Once we are satisfied with the changes made in our code, we can integrate the multiple commits created in our supporting branch into the mainline by using `git merge`. This sentence generates a dedicated commit that combines the development of the two branches (current and target). Merge commits are unique as they are based on two parent commits. One should notice that merge commits are produced in the current branch; `git merge` updates the source branch with the modifications made on the target branch, leaving the latter unaffected.

Closely related to the fact that branches act as a pointer is the concept of `HEAD`. `HEAD` is the name used to refer to the commit we are working on. It frequently points to the most recent commit. We can change the position of `HEAD` by using the command `git checkout` (actually, when you are applying this sentence for switching between branches, what Git is doing is migrating the tip `HEAD` from one branch to the other). We can move `HEAD` to a specific commit (and detached it from a branch) using the command `git checkout` plus the label that identifies the commit. In the same way, we are able to move the position of a branch by typing `git branch -f [branch_name] [position]`.

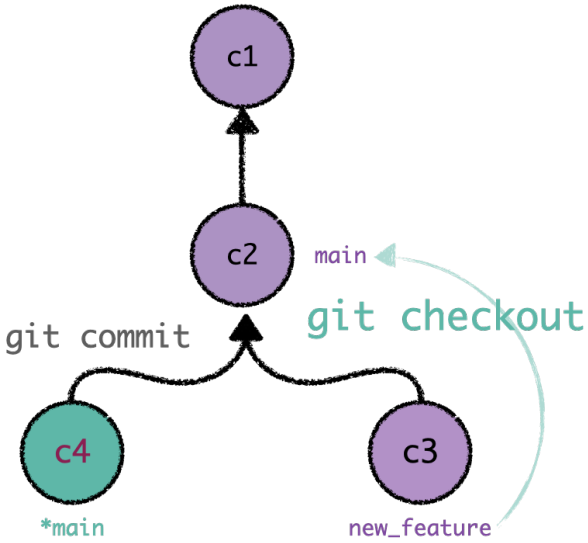

Fig. 4: Once we have created the new commit *c3*, we go back to the branch `main`. A new commit *c4* will generate a bifurcation or fork in the Git history. Commits from different branches (*c3* and *c4*) are disconnected.

Resources

If you are interested in delving into the ideas explored in

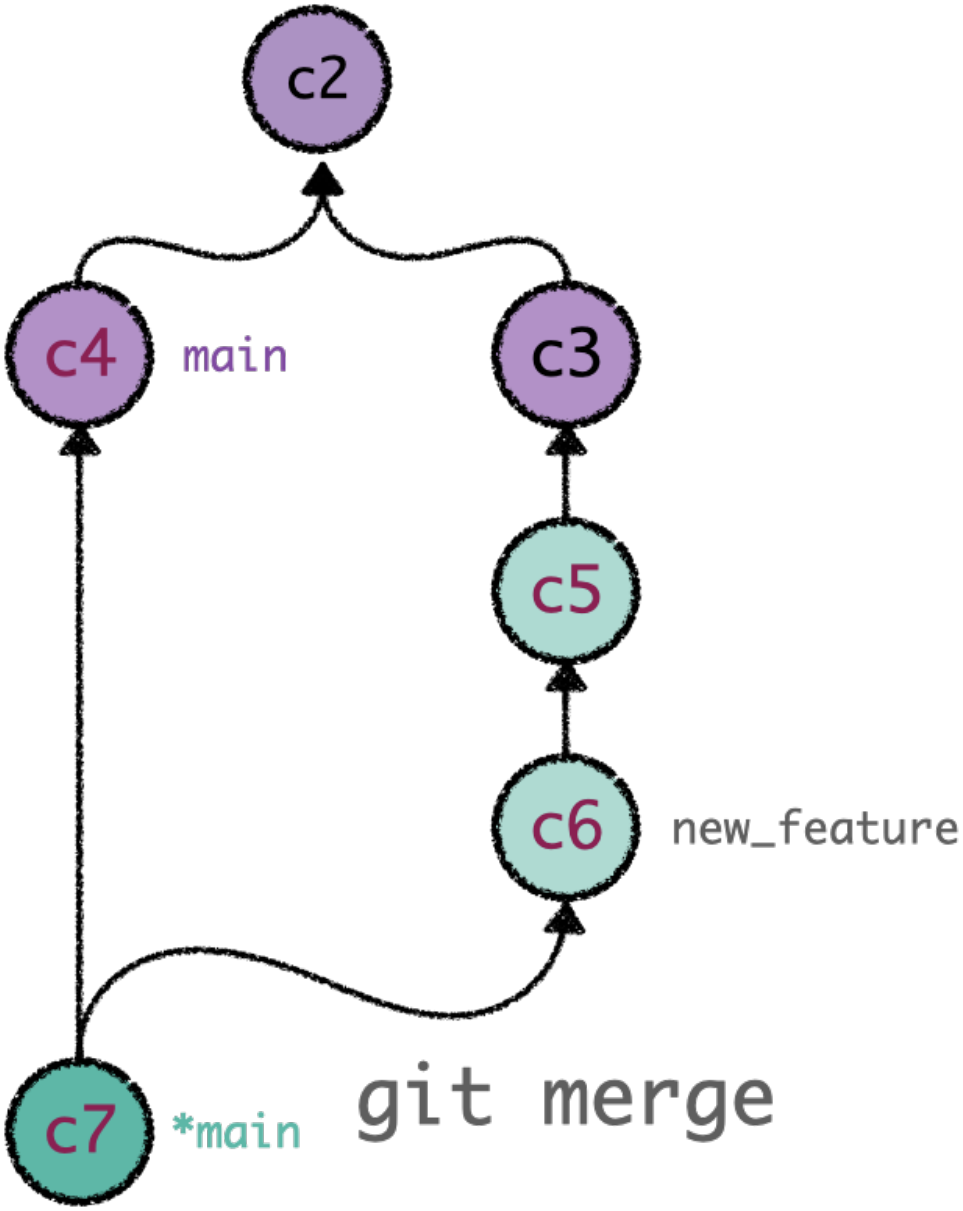

Fig. 5: `git merge` generates a unique commit that unifies the two history lines.

this section, you should read the following links:

- You will find all the information you need about Git branches on this *page*.
- Enlightening *tutorial* about the `merge` command. It includes the multiple options this command offers and the different merge strategies.

2.2.3 Joining histories

Once you have finished your work and the new modifications satisfy your criteria, you are in position to integrate all these changes into the mainline of the project and share your thoughts with your teammates. Git offers two primary utilities to incorporate novel code from one branch into another, `git merge` and `git rebase`. Despite sharing a common goal, the methodology and philosophy behind these two are very different.

Remember

Git branches are pointers to specific commits in Git history. They are not new repositories or folders.

2.2.3.1 Git Merge

The command `git merge` combines the content from two different branches into a single, unified commit. The process is the following: `git merge` takes two branch pointers and identifies their common commit ancestor. Once Git locates this point, it generates a unique "merge commit" (on the branch we are currently working on) that combines all committed changes from their common parent. This new commit is unique in the sense that it depends on two parent commits. One should notice that `git merge` only modifies the current branch; the target branch history remains unaltered (as illustrated in the previous figure).

2.2.3.2 Git Rebase

`git rebase` is the second method used to integrate changes. Rebasing compresses the content of the source branch into a single patch and integrates it on top of the target branch. In this way, we transfer the finished work from one branch to another, rewriting and flattening the history. It appears as if the entire sequence of commits had been created from start to finish on the same branch, achieving a linear project history. Internally, what Git is doing is replicating the commit sequence of the source branch onto the target base, creating new commits for each commit in the original branch. The primary motivation for using `git rebase` is to achieve a linear project history.

Good practices

An excellent exercise to avoid merging conflicts when pushing your content, either to the main branch or to a remote repository, is rebasing your changes with the most recent version of `main`. In this way, you guarantee that your changes are based on the most updated version of the repository, facilitating the merging process.

Consider the following scenario: you have worked for several days on an additional feature for a program. Meanwhile, the mainline has been updated multiple times by your teammates, leaving your files obsolete. If you want to push your work to remote and open a Pull Request to merge your latest additions, a good practice is rebasing your content with the most updated version of the remote repository:

1. First, you have to download the remote main branch to include all these new modifications added by your colleagues: `git pull origin main`

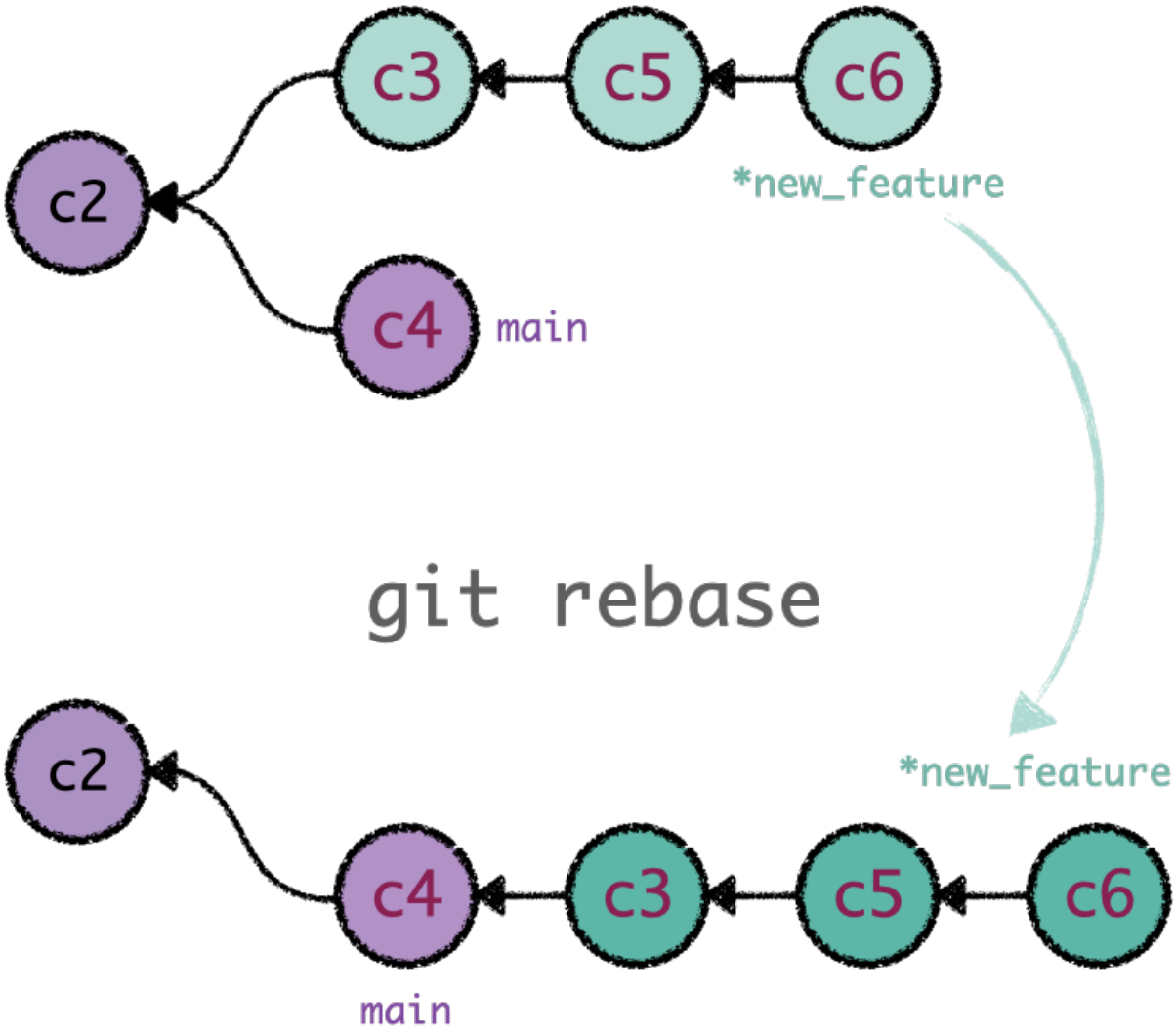

Fig. 6: `git rebase` captures the content from one branch and reapplies it on top of the other. In that way, rebasing changes the base of our branch to a different commit.

2. Once you have your local main branch updated, you have to switch to your local working environment: `git checkout feature`
3. Sitting on your branch, rebase your modifications with *main*: `git rebase main`

Eureka! Now your branch is based on the remote main branch and it is ready to be pushed without any problem.

2.2.3.3 Git Rebase -i

An appealing option `git rebase` offers is the interactive mode. The flag `-i` initializes the interactive mode, which allows you to rebase the commits individually as well as modifying their properties. In this way, you can determine how the sequence of commits is transferred to the new base. Besides, this mode opens the door to rewriting your Git history by using the command `git rebase -i HEAD~` plus the number of commits you want to rewrite. With this command, instead of rebasing against a distinct branch, you are "shifting" your base to the same point while allowing you to remodel the structure of your commits. `git rebase -i` is extensively used (and widely recommended) before pushing your modifications to remote; you can rectify your changes, reorganize your history, and keep the number of commits to a minimum.

`git rebase` not only allows you to rewrite your history, but it reinforces the concept of branches as pointers. Next to `git rebase` must always be a **reference**: either the label that refers to an individual commit (to rewrite your history), or a target branch to integrate your changes linearly.

Resources

You will be able to find more literature about merging utilities in the following links:

- *Tutorial* about `git merge`.
- In-depth *tutorial* about `git rebase`, including the interactive mode.
- *Conceptual discussion* about the two merging strategies.
- Immerse in the most complete *source* about Git.
- We encourage to exercise `merge` and `rebase` tools by using the *interactive tutorials*.

2.2.3.4 Rewriting history

Before pushing your modifications to your shared repository, having a clean, organized history is strongly recommended to help the reviewing process to your comrades (and, of course, to facilitate your understanding of the working flow). Git offers multiple workflow customization tools that give you total control over the project development. We mentioned `git rebase -i`, but there are more options to restructure your Git commits, such as `git commit --amend`. These *two* essential *references* contain **everything** you should know about history-rewriting commands. These sources expose the topic in such a precise and straightforward manner that we are unable to include something relevant to the discussion.

2.3 Collaborating on a project

Git workflow is the basis that supports the entire development process, promoting the collaboration among team members while avoiding redundant tasks in a distributed environment. It consists in a set of steps outlined in the following document:

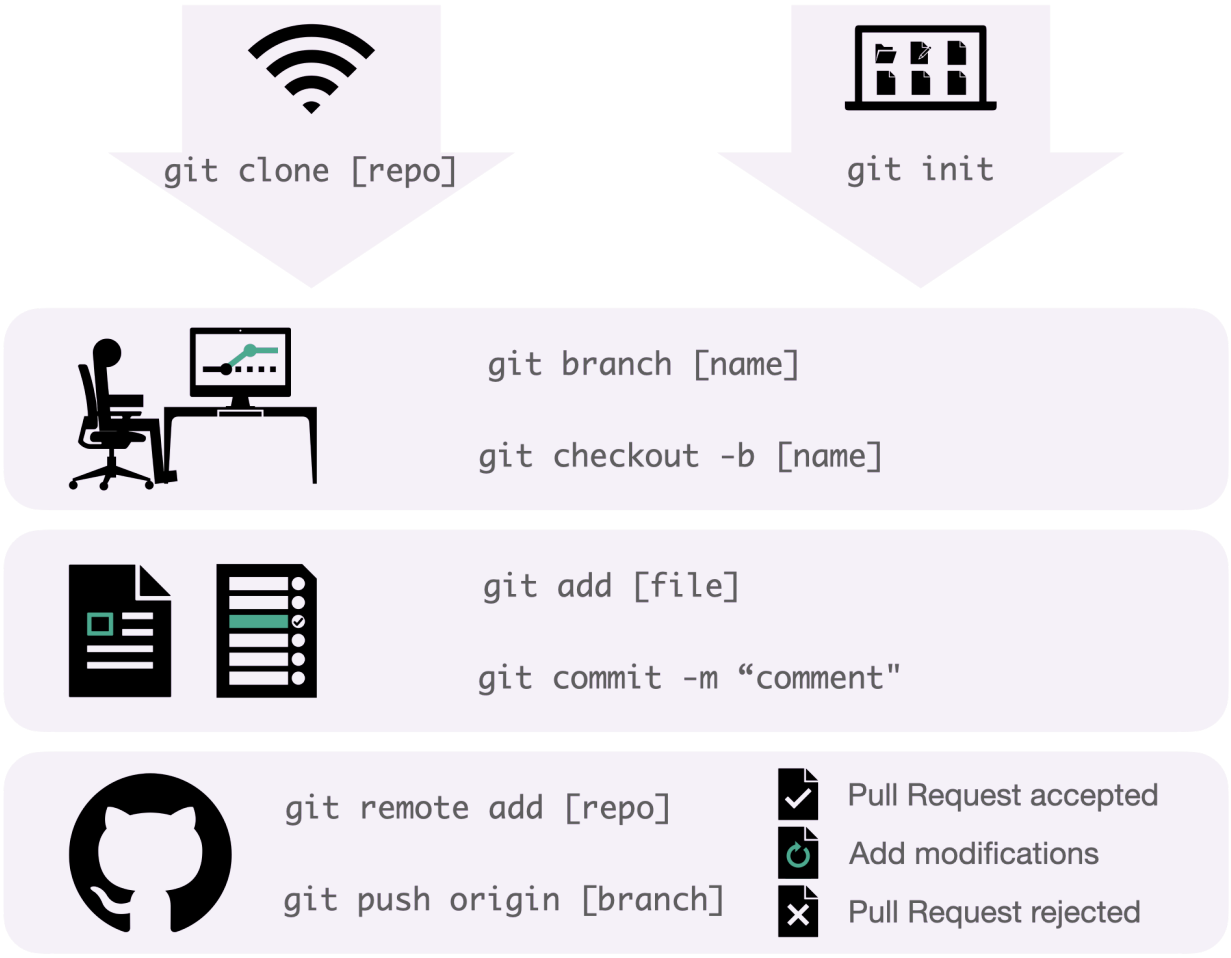

Fig. 7: Git workflow

1. SETUP: Download an online repository using `git clone [url repo]`, or start your own local Git repository using `git init`.
2. BRANCH: Create your safe environment using `git branch [branch-name]`. This isolates your local work and allows testing additional features.
3. STAGE & COMMIT: Stage your recent work using `git add [modified-file]` and save it with `git commit -s -m "[comment]"`.
4. SHARE: Time to upload your project. Add a Git URL using an alias with `git remote add [alias] [url repo]` in case that your project had been started locally. Upload your local branch commits to the remote repository using `git push [alias] [local-branch]`.
5. REVIEW: Launch a new **Pull Request** to publicize your modifications and start a reviewing process. This opens a discussion with your teammates concerning the correctness of your changes. If the proposal succeeds, your

modifications will be merged into the main line. Reviewers can ask you to include extra adjustments to approve your code.

Remember

When committing, the tag `-s/--signoff` is necessary in our organization to pass the *DCO* automatic check in the CI pipeline.

2.3.1 Open an issue

We use issues to keep track of the evolution of our software projects. We should create an issue describing the new functionality/fixes that we want to add to the code base. This ensures that every relevant contribution can be discussed and documented.

The Issues will document all the discussion and design decisions that we make when deciding *what* we want to fix/improve in our projects. In the Pull Requests, we will discuss *how* that changes are implemented.

Minor modifications can be discussed directly in the Pull Request. There is no need to open an Issue describing changes that take less than 4 hours to implement, such as:

- Fixing typos or minor improvements in the documentation.
- Small bugfixes.
- Refactoring variable names.
- Adding an additional test.

2.3.2 Create a new branch

On your local clone of the repository, create a new branch from the latest upstream master branch commit of the repository you need to work on.

We do not enforce any particular branch naming policy, but we recommend using prefixes, like `fix`, `feature` or `docs`, for example.

2.3.3 Development and Pull Requests

Work on your local repository as you see fit. When the time comes to make the Pull Request:

- Rearrange your commits to improve its readability: squash similar changes into individual commits and try to keep the overall commit count at a minimum.
- Choose a short, descriptive title that summarizes the contents of the Pull Request.
- If the Pull Request is addressing an Issue, make sure to reference it in the first Pull Request comment.
- Assign the Pull Request to yourself and ask for a review.

2.3.4 Code review and requested changes

Reviewers

The project maintainer or a designated reviewer will review Pull Requests.

After a code review, add the requested changes in a new commit. This is useful because it's possible to check again only the specific changes that the reviewer requested. When your Pull Request is approved, rearrange all the commits to have a descriptive history of your changes. Also, rebase your Pull Request with the master branch.

2.3.4.1 Avoiding cascading delays when submitting Pull Requests that depend on one another

If you want to submit several Pull Requests that depend on one another, and you don't want to wait for the revisions of the former to submit the latter, you can, as an exceptional measure, submit them without waiting to rebase them against master, so each of them will depend on the changes of the previous ones.

Always add a comment to the Pull Request showing that it depends on other Pull Requests that must be merged first.

To merge this kind of Pull Request, you need to know how to use the `rebase -i` git command:

- (Maintainers only) Merge into master the first Pull Request in the dependency chain.
- Because this Pull Request is merged using squash, you need to rewrite your Pull Request commit history removing old commits from the previous Pull Request and adding the squashed one from master. Using `rebase -i` command is a simple task.

2.4 FAQ

2.4.1 What is Git?

A fundamental key in collaborative projects is having control (or a tracking history) over the modifications made on the project. Git is a popular distributed version control system broadly used in software development.

Git facilitates the addition of novel content through the use of *branches*, bifurcations in the tracking history of the project that allows the creation (or update) of extra features in a secure, fully controlled sandbox.

Each project has a default branch (usually named `master` or `main`) that act as the source of truth for the project's code. We develop new features in different branches, and we integrate them into the default branch after a review process.

2.4.2 What is the difference between Git and GitHub?

As mentioned, Git is a popular *Version Control System* used to manage a wide variety of projects. Git is installed and maintain in your local machine, and it keeps a record of your programming versions. What makes Git unique from the rest of its competitors is its branching model, which allows you to test and try new ideas without touching the mainline.

Having said that, what is GitHub's role? *GitHub* is a Git repository hosting service. It is a cloud-based database where people can track and share with other developers their local Git repositories. Unlike Git, GitHub presents a graphical user interface to administer your projects using its built-in control tools.

Although they are already very interesting utilities on their own, the real collaborative environment boosts when Git and GitHub work together. You can share your projects through GitHub, giving your teammates the opportunity to

review and comment on your additions. As the project evolves, new branches are created to introduce new changes, allowing the group to work on distinct features without overwriting each other's work during the process. This philosophy promotes coordination among comrades, making possible a collaborative development process.

2.4.3 What is a Pull Request?

After finishing writing the new feature of your collaborative project, you will be eager to merge your fresh content with the main branch. However, before adding the new features to the mainline, your code should pass a reviewing process for discussing with your colleagues the potential and suitability of your changes. Thus, before merging the pushed branch into the base one, you must open a **Pull Request** to notify others about the additions you have been working on, compare the differences between your branch and the repository base branch, and debate about further commits to improve your work.

After opening a Pull Request, you will see a discussion panel where your collaborators can review your changes, suggest new additions or recommendations, and even contribute to the Pull Request with new commits. You can also add new content by pushing new commits from your branch to the open Pull Request.

2.4.4 What is an Issue?

Issues are used to suggest new additions, tasks, enhancements, or questions related to the repository. Any collaborator can open a new issue (in the case of public repositories, any GitHub member can create a new issue), which opens a discussion thread where team members can address the topic.

Resources

One can find more information about Pull Requests and Issues along these pages:

- How to collaborate with *Pull Requests*.
- Reference guide for *creating a Pull Request*.
- *Available features* in Issues page.
- How to write a *proper issue*.

2.4.5 What is a branch?

When developing new features, coders work in safe spaces where their changes do not affect the project's mainline. Most VCS's offer a secure, virtual environment so that programmers can work freely without worrying about "corrupting" the project. Git goes one step further in providing these isolated environments by promoting the use of branches. Git branches are merely pointers, references to committed changes. As one develops new features and stores them, the branch represents the tip of that series of commits. In this way, Git branches are notably lightweight, as they are not containers keeping incoming commits (remember, they are solely references to specific commits in Git history).

Consider the scenario where you want to correct a bug. Immediately, you spawn a new branch from the project's mainline. This new branch creates a bifurcation, isolating all incoming additions while protecting the main base from unstable code. Once you are satisfied with your changes and how the Git history looks, you can safely merge your fresh features into the mainline.

2.5 How to

2.5.1 GitHub

2.5.1.1 Log in using ssh

GitHub offers three options for downloading the contents of a remote repository. We recommend using the *ssh key* option to clone external repositories, especially when logging into GitHub using two-factor authentication. Follow these steps to *create a new ssh key* and *add it to your GitHub account*.

2.5.1.2 Log in with a password when two-factor authentication (2FA) is enabled

One can generate *access tokens* and use them instead of your password to log in to GitHub. Nevertheless, depending on your account security settings, access tokens may produce errors, preventing downloading from remote repositories.

2.5.1.3 Reference pull requests and issues in your comments

- Use *Cross Links* to connect different conversations within a repository. When commenting, type `#` to bring up a list of suggested **issues** and **pull reqests**, and select the issue or pull request number to reference it. These references are automatically converted to shortened links to the issue or pull request.
- If you are opening a pull request that fixes the question proposed in an issue, add `closes #[issue/PR number]` in the pull request message. This will close automatically the issue when the reviewer approves and merges the pull request.

2.5.2 Git

2.5.2.1 *Git push* and *Git pull*. Remote branches

Once you are done introducing your new additions to the project, and your files are ready to be uploaded, the easiest way to create a new remote branch to host your local work is using `git push [remote alias] [local branch name]`. This will create a remote branch (named after the source branch) that will store the modifications commited on your *local branch*. If one is interested in having a remote branch labelled differently, Git allows you to choose the name of the remote branch by adding to `push` the tag `[source local branch]:[destination remote branch]`. In this way, the sentence `git push origin local feature:remote feature` will push to the remote repository (referred by *origin*) the changes introduced in your local branch *local feature* to the remote branch *remote feature*. In case that you are suffering some problems when pushing your commits, you can always override the remote branch with the contents pushed from local by using the command `git push -f`. Since the remote branch is, by default, the "true" Git history, what this command does is telling Git that the true source is instead the local branch, giving it preference over the remote one. One must be extremely careful when using this option, as it will erase the contents of the remote branch.

You can download a remote repository using `git pull`. Sitting on the target branch where you want to save the remote content, you can write `git pull [remote alias] [remote branch name]` to download the last version of the source branch.

Resources

A complete description of `git pull`/`git push` and the options they offer can be found in the following links:

- *Syncing tutorial* and references therein.

- *Git push* documentation page.
- *Git pull* documentation page.

2.5.2.2 Moving across commits

`git checkout [commit reference]` moves the current tip to a previous committed change, the latter being specified by a **hash** number. Nevertheless, this method is rather tedious and inconvenient, the use of *relative references* being a more appropriate option. Starting from a specific branch or commit, one can move to previous snapshots by using either `^` (this command moves the tip upwards one commit at a time) or `~<number>` (this commands moves the tip upwards as many times as the given value). Considering the scenario where the last commit is referred by `HEAD`, one can write `git checkout HEAD~3` in order to relocate the tip three commits back.

Hash

All committed changes are identified by a unique hash number (`git log` displays a complete list containing all commits and their references). Git usually requires only a few characters of the hash to identify the corresponding commit.

2.5.2.3 Interactive rebase

`Git rebase --interactive` is broadly used to review commit changes and apply further modifications to old commits. The unflagged command, `git rebase`, is used to compress the changes made on the source branch and apply them on top of the target branch, achieving in this way a flat Git history. `git rebase` also avoids merging conflicts when pushing new content to remote repositories, as it allows the coder to rebase his/her local changes on top of the most recent version stored in remote.

The interactive mode (called via `git rebase -i`) gives the user the opportunity to rebase individual commits interactively. More interestingly, it allows rebasing changes committed on the current branch. This opens the door to revisit old commits and correct small errors, such as erroneous commit messages. One can amend or squash multiple commits, delete non-desired changes, reset `HEAD` to a specific label, relabel the current `HEAD` with a new name, etc. In order to modify old commits, one should write `git rebase -i HEAD~#` where "#" is the number of previous commits to be rebased. For further reading, we refer *this tutorial* about rebasing Git history with the interactive mode.

2.5.2.4 Rewrite Git history

Before pushing local changes to remote, a recommended practice is rewriting Git history. This facilitates the reviewing process, while helping you to understand and clarify the working flow. We have already mentioned one tool for rewriting Git history (`git rebase -i`), but Git offers other utilities for customizing the historical commit changes. Read this *this reference* to understand the tools available to achieve your desired Git history.

2.5.2.5 Defer changes

You have probably needed to switch branches while working on some feature of a project. Git does not allow you to checkout to another branch without committing the modified files. One can use `git stash` to temporarily shelve these changes, so you can freely move through the Git history, work on new features and, once you have finished, come back and *unstash* the modifications. This command becomes extremely useful when your code is not ready to be committed, but you need to switch to a different working environment. As mentioned, `git stash` saves your uncommitted changes for later use, allowing you to create new commits, modify new files or switch branches. In order to re-apply the archived changes, you should use `git stash pop` (or `git stash apply` if you are interested in maintaining them in your stash). Keep in mind that Git will not stash untracked or ignored files. For more information about this topic, we recommend reading *this page*.

2.5.2.6 Inspecting the state of the repository

To retrieve information about untracked files, the staging area and the working directory, one should use the `git status` command, which displays all modified files on your working directory. On the other hand, `git log` command shows the Git history of your project, listing all committed changes made on your current branch. This *page* explains the multiple options and tools that `git log` can offer.

2.6 Bibliography

2.6.1 Git

2.6.1.1 Introduction to Git

Slides and articles recommended for a gentle introduction to Git.

2.6.1.2 Interactive labs to learn Git

Online courses and step-by-step demonstrations explaining the most popular Git commands and collaborative processes

2.6.1.3 Git Basics

2.6.1.4 Reference

2.6.1.5 How To

2.6.2 GitHub

2.6.2.1 Pull Requests (PR)

2.6.2.2 Code reviews

2.6.2.3 Accessing GitHub

2.6.2.4 Reference

2.7 Environment setup

This document describes how to set up your computer and install all the tools required to develop Open Source projects at Fragile Tech.

We recommend installing any GNU/Linux OS, but this guide will target a fresh installation of Ubuntu 20.04

2.7.1 Communication tools

2.7.1.1 Zoom

Videoconference tool. We will use Zoom for our weekly meeting every Wednesday at 18:30, and to catch up if needed.

2.7.1.1.1 Installation

- Download the .deb package from here
- Open the download file and click install on the graphic interface.

2.7.1.2 Calendly

Configure a Calendly account and create a calendar with the hours that you plan to spend working on the Fragile Guide.

You can schedule a meeting with Guillem if you need any kind of support.

2.7.2 Development tools

2.7.2.1 PyCharm CE

IDE for developing Python projects.

2.7.2.1.1 Installation

```
sudo snap install pycharm-community --classic
```

2.7.2.1.2 Setup

Custom fonts

1. Download the Hack fonts (optional)
2. Install the fonts
3. Open settings (`file->settings` or `Ctrl+Alt+s`) go to `Editor->Font` and choose your desired font and font size.

Change theme

1. Open settings and go to `Editor->Appearance & behavior->Appearance`
2. Select the theme you like the most. (I recommend `Darcula`.)

Visual guides

1. Open settings and go to `Editor->Code Style`
2. Set `Hard wrap` to `99` and `Visual guide` to `80`.

Configure documentation

1. Open settings and go to `Editor->Tools->Python integrated tools`.
2. Set `Testing->Default runner` to `pytest`.
3. Set `Docstrings->Dosctring format` to `Google` and check both tick boxes.

Resize font using the mouse wheel

1. Open settings and go to `Editor->General`
2. In the section `Mouse control` tick the box `Change font size with ctrl+Mouse Wheel`.

2.7.2.2 GitKraken

Git GUI client.

2.7.2.2.1 Installation

```
sudo snap install gitkraken --classic
```

2.8 Python style guide

2.8.1 Introduction

This guide documents coding and style conventions for contributing to FragileTech Python projects.

2.8.2 Code Style

1. Contributing to existing external projects - do not change the style.
2. Follow the Python Style guide.
3. Use PEP8 with 99 char line length limit.
4. Class methods order:
    1. `__init__`
    2. Attributes
    3. Static methods
    4. Class methods
    5. Public methods
    6. Protected (_) methods
    7. Private (___) methods
5. Use double quotes `"`. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string.
6. Favor f-strings when printing variables inside a string.
7. Do not use single letter argument names; use X and Y only in Scikit-learn context.
8. Use Google style for docstrings.
9. Format of TODO and FIXME: `# TODO(mygithubuser): blah-blah-blah`.
10. Add Type hinting when possible.
11. Use standard argparse for CLI interactions. Click is also allowed when it improves the readability and maintainability of the code.

2.8.3 Useful tips

1. **Each function should do only one thing**. If you find yourself writing a long function that does a lot of stuff, consider splitting it into different functions.
2. **Give variables a meaningful name**. If names became too long, use abbreviations. This abbreviations should be explained in comments when defining the variable for the first time.
3. Keep in mind that coding is creating abstractions that hide complexity. This means that you should be able to get an idea of what a function does just by reading its documentation.
4. **Avoid meaningless comments**. Assume the person who is reading your code already know how to code in python, and take advantage of the syntax of the language to avoid using comments. For example, a comment is welcome when it can save you reading several lines of code that do stuff which is difficult to understand.
5. **Document the functions**, and make sure that it is easy to understand what all the parameters are. When working with tensors and vectors, specify its dimensions when they are not obvious.
6. **Follow the Zen of Python**, it is your best friend.
7. A well documented function lets you know what it does and how to use it without having to take a look at its code. **Document all the functions**! It is a pain in the ass but it pays off.

2.8.4 Code formatting

- We use `black` for formatting the code. Run `black .` before committing to automatically format the code in a consistent way.

2.8.4.1 Separating blocks of code with blank lines

Although using blank lines to separate code blocks may seem like a good idea, it has the following drawbacks:

- It does not offer any information regarding how and why you are defining different blocks of code.
- It makes code reviews more difficult:
  - It forces the reviewer to make assumptions about why you decided to create the different blocks.
  - It removes context when showing possible suggestions about changes in the code.
  - Sparse code makes adds unnecessary scrolling time when reading the code.
  - Sparse code makes the code diffs less reliable.

If you want to separate different code blocks inside the same function there are better alternatives:

- **Write a comment** explaining what the code block you are defining with a blank line matters:
  - It helps the reviewer understand why you are separating different code blocks.
  - If the comment is meaningless you'll realize that it was an unnecessary line break. For example, imagine you are defining a fancy neural network as a `pytorch.nn.Module`:

```
super().__init__()

self.device = device

self.layer_1 = torch.nn.Linear(in_dim, out_dim)
self.layer_2 = torch.nn.Linear(out_dim, out_dim)
```

(continues on next page)

(continued from previous page)

```
self.layer_3 = torch.nn.Linear(in_dim, out_dim)
self.layer_4 = torch.nn.Linear(out_dim, in_dim)
```

To understand if the blank lines you wrote to separate different code blocks are useful, you could write a comment to separate the different blocks:

```
super().__init__()
# Device definition
self.device = device
# Encoder layers of my fancy DNN model
self.layer_1 = torch.nn.Linear(in_dim, out_dim)
self.layer_2 = torch.nn.Linear(out_dim, out_dim)
# Decoder layers of my fancy DNN model
self.layer_3 = torch.nn.Linear(in_dim, out_dim)
self.layer_4 = torch.nn.Linear(out_dim, out_dim)
```

When you do that you will realize that you almost spent more time reading the comments than the code blocks that they separate, and that the "device" comment you wrote is extremely obvious for anyone that is remotely familiar with `pytorch`. In that case, deleting taht blank line improves the readability of your code.

The comments about the blocks of your fancy neural network are indeed adding some useful information, but there may be a better alternative. Those comments are useful because they give information about what each layer is doing, but this is something that could be improved by finding meaningful names to the defined layers.

```
super().__init__()
self.device = device
self.encoder_in = torch.nn.Linear(in_dim, out_dim)
self.encoder_out = torch.nn.Linear(out_dim, out_dim)
self.decoder_in = torch.nn.Linear(in_dim, out_dim)
self.decoder_out = torch.nn.Linear(out_dim, out_dim)
```

Now that the names are meaningful and there are no comments, we find that the different line length between `self.device` and `self.encoder_in = torch.nn.Linear(in_dim, out_dim)` makes it easier to differentiate those two blocks. On the other hand `encoder` and `decoder` have the same length, and that makes it difficult to spot quickly when the definition of the decoder starts. Adding a comment between the two of them will improve the readability of the code:

```
super().__init__()
self.device = device
self.encoder_in = torch.nn.Linear(in_dim, out_dim)
self.encoder_out = torch.nn.Linear(out_dim, out_dim)
# Decoder layers of my fancy DNN model
self.decoder_in = torch.nn.Linear(in_dim, out_dim)
self.decoder_out = torch.nn.Linear(out_dim, out_dim)
```

Keeping the comment about de encoder is not worth it in this case because its commenting a block of only two lines of code, but if the encoder had more layers it could be useful to make the code more readable. Inside the `__init__` function, try to not add more than 1 comment per 5 lines of code.

- **Write a new function** to encapsulate the new code block:
  - It will reduce the cognitive load of the reviewer by abstracting the complexity of the code.
  - The function call will remove the need of additional comments.
  - It will make testing easier.

- **Use reserved words** in variable definitions and function calls to separate code blocks: Take advantage of the different color highlighting that reserved words have to make an implicit separation of blocks. For example, instead of:

```
my_var = do_one_thing(param_1=val_1, param_2=val_2, param_3=value_3)

another_var = do_another_thing(my_var)
return another_var
```

  You could do:

```
my_var = my_object.do_one_thing(param_1=val_1, param_2=val_2, param_3=value_3)
return do_another_thing(my_var)
```

2.8.5 Pycharm tips

1. Using PyCharm as an IDE will help you highlight the most common mistakes amd help you enforce PEP8.

2.8.6 Example codebase

Reading well-written Python code is also a way to improve your skills. Please avoid copying anything that has been written by a researcher; it will likely be a compendium of bad practices. Instead, take a look at any of the following projects:

- Django
- Flask
- Jinja 2

When it comes to Reinforcement Learning, please avoid at any cost using OpenAI baselines as an example.

2.8.7 Resources

- **The Zen of Python in examples**:
  - Post in Medium
  - Quora post
  - Code example
- **Idiomatic Python**:
  - Blog post about Idiomatic Python
  - More Python Idioms

2.9 Introduction to Python

According to Python *webpage*, "*Python is a programming language that lets you work quickly and integrate systems more effectively.* [...] *Python is powerful... and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open*". Reading these lines, one can get an immediate and accurate understanding of the key guidelines that drove the development of this language.

To be precise, Python is a high-level, interpreted, general-purpose, and dynamic programming language that supports object-oriented approach. Its modularity and easiness to be customized to the developer's needs with external libraries have made Python the first choice used by many programmers, and broadly adopted in multiple environments. Python's key strengths are:

1. Open source language: Python can be downloaded without any cost at its official webpage. It is developed under an open source license, *making it freely usable and distributable*.
2. Friendly language: Compared with other programming languages, Python is very intuitive and natural. One only needs a few lines of code to write a program, thus helping the debugging process.
3. Libraries and packages: Python has a vast library of modules and functions ready to enhance its capabilities. This modularity allows the application of Python in diverse fields, such as Machine Learning, Mathematics and Science, or Web and Internet development.
4. Extensible and portable: Python allows you to use libraries written in other languages. Moreover, it is platform-independent, which means that Python code can be executed on any platform (Linux, Windows, macOS).
5. Dynamical data-structure: Data-type in Python is assigned dynamically, which means that one does not have to declare the data-structure of each variable in advance (as occurs in *Fortran*, for instance), but it is assigned automatically while declaring the variable. For example, when we set the variable `x = 5.0`, Python assigns systematically a float value (`5.0`) to a float variable (`x`), avoiding wastage of memory.
6. Object-oriented approach: Python supports an object-oriented environment (properties and behaviors are bounded into individuals objects), providing multiple functions and tools to developed applications with an object-oriented approach in mind. We will discuss this point in greater depth later.

Resources

You will be able to find more about Python's core strengths at:

- *Python webpage*.
- *Python Tutorial*.
- *Features of Python*.
- Discussion on why *Python is so popular*.

2.9.1 Data-structures in Python

Python uses ***Data Structures*** as the method to handle and organize the data. They are the fundamental building blocks around which one develops a Python program. Data Structures allow the coder to store data, perform different operations and establish relationships among them. Python provides multiple types of data structures with unique features to organize and store the information efficiently according to the developer's needs.

External libraries

Raw Python offers different data structures that can easily be enhanced by using standard and external libraries. In this chapter, we will focus only on the fundamental data types built into core Python. We will give a small glimpse of enhanced data structures at the end of this section.

Data Structures are classified into two main categories: *Primitive* structures, which represent the simplest blocks of data storage; and *Non-Primitive* structures, which consist of more advanced elements used in more specific and complex scenarios.

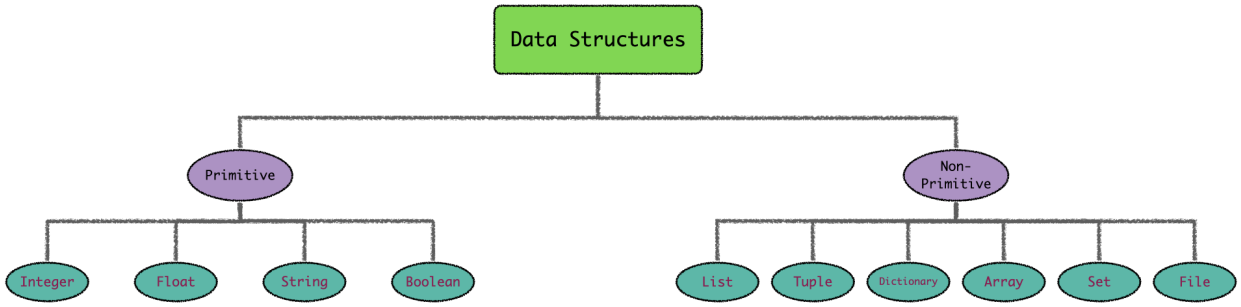

Fig. 8: **Python Data Structures**. These can be classified into two main groups: *Primitive* Data Structures, core building elements in Python, and *Non-primitive* Data Structures, collections of Primitive Data Structures.

2.9.1.1 Primitive Data Structures

Primitive Data Structures represent the most elementary building elements for storing pure and plain data values in Python. They are defined in conjunction with a predefined set of operations, allowing the manipulation of the stored data. Python has four types of Primitive Data Structures:

1. **Integer Data Structures** are used to represent the integer numeric data. The maximum integer number one is able to reach is constrained by the memory of the system.
2. **Float Data Structures** are used for designating rational numbers. Float numbers in Python are represented as 64-bit (double-precision) values. The maximum value a float value can have is 1.8×10^{308}, while the minimum number one can represent is 5.8×10^{-324}. One should notice that Integer Structures are *promoted* to float numbers when operating with Float Data Structures. In this way, the operation `3 + 4.0` yields `7.0`; the integer number `3` is implicitly converted to the float number `3.0` to operate. This type of transformation is called *Implicit Data Type Conversion*.
3. **String Data Structures** are sequences of character data. Python uses String Structures to store textual data information as an immutable series of characters (string variables cannot be updated once they are defined). String chains are delimited by either single or double-quotes. Surprisingly, string variables are iterative objects; they represent recursive structures where each character is also a string object. One can find more information about String Data Structures *along these lines* (we strongly recommend reading the section concerning *Suppressing Special Character*).
4. **Boolean Data Structures** is a built-in data type that represents the true value of an expression. It has only two possible values, `True (1)` and `False (0)`. Python considers Boolean Data Structures as numeric elements, which means that one can apply arithmetic operations to boolean variables. This fact turns very useful when counting the number of `True` values, for instance. Read *this page* to get more information about Boolean Data Structures.

Data Type Conversion

One can check the type of any defined variable using the built-in Python function `type()`. Besides, coders can modify and play with the *nature* of the defined variables. There are two ways to change the type of a variable:

- **Implicit Data Type Conversion**: Python automatically converts the data type of the objects for the user. This action arises when operating variables with different data types, as we saw in the previous example.
- **Explicit Data Type Conversion**: Python has built-in functions that allow developers to explicitly change certain variables' data types at will. Nevertheless, some defined data structures might not be modified.

2.9.1.2 Non-primitive Data Structures

In addition to the building elements used to construct the most elementary objects, Python provides a series of structures that work at a higher level of complexity than the former. Instead of storing new definitions, these new objects store a **collection** of Primitive Data Structures. Called *Non-primitive Data Structures*, they are further classified into different categories depending on how the primary elements are stored:

Strings

Strings can also be viewed as non-primitive Data Structures, since they are an *immutable collection of unicode characters* of length 1.

1. **Arrays** are a fundamental structure included in the Python standard library. They consist of fixed-size data objects where each element is saved in adjoining memory blocks. Each memory block is identified by a unique number, allowing the interpreter to efficiently locate the stored objects based on this index. However, arrays can only hold data structures of the same type and allow neither new entries nor modifications of existing elements (arrays are immutable). These restrictions make arrays incredible efficient containers, being extremely fast to find any variable contained inside a block. The `array` module should be imported before using them, allowing the user to define array-type objects by applying the `array` function.

```
>>> import array as arr
>>> new_array = arr.array("f", [3.4, 5.7, 2.1])
>>> print(new_array)
array('I', [2, 4, 6])
```

2. **Lists** are built-in Python structures. They are implemented as *dynamic arrays*, meaning that one is able to remove or introduce new elements inside a list dynamically (the list object will automatically reset the memory size of the blocks). Lists can store different data structures, which allows mixing arbitrary kinds of data into a single list. Lists objects are defined by using square brackets `[ ]`.

```
>>> list = ["a", "b", "c"]
>>> print(list)
['a', 'b', 'c']
>>> list[1]
'b'
```

3. **Tuples**, like lists, are part of Python core language. Tuples are immutable objects (tuple entries cannot be modified or removed dynamically), but they can store multiple data structures. One defines tuple-type objects using parenthesis `( )`. They are broadly adopted in scenarios where we need to pass a data set to other users without them being able to modify the original collection.

```
>>> tuple = ("one", "two", "three")
>>> print(tuple)
('one', 'two', 'three')
>>> # tuples are immutable
>>> arr[1] = "hello"
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

4. **Dictionaries** represent a fundamental Data Structure in Python, built into its core language. Dictionaries are used to store an arbitrary number of primitive, **indexed** objects, meaning that each element is identified by a unique label or *key*. This indexation allows developers to quickly locate the data associated with the given dictionary key. An illustrative analogy to understand Python's dictionaries is to compare them to a phone book, where each phone number (the data) is associated with a name (the key). As far as dictionary keys are concerned, not every object can be used as a valid key. These must be *hashable* objects, elements that do not change during their lifetime and can be compared to other objects (strings and numbers are usually adopted as dictionary keys).

```
>>> my_dict = {
               "alice" : 1,
               "bob" : 2,
               "charlie" : 3
               }
>>> print(my_dict["alice"])
1
```

5. **Sets** represent an unordered collection of primitive, **non-repeating** objects. The latter restriction is, precisely, their principal trait: duplicate elements not allowed in sets. Sets are mutable objects with dynamical insertion and deletion of entries.

```
>>> my_set = {"alice", "bob", "charlie"}
>>> print(my_set)
{'alice', 'charlie', 'bob'}
```

6. **Files** Data structures are used to store and later recover large sets of data and information. Python offers multiple tools to write and read files.

Resources

All the information concerning Python's Data Structures can be found here. These pages contain additional information explaining sophisticated structures with more advanced features than the basic Data Structures we have covered:

- *Common Python Data Structures*.
- *Tutorial about Python's Data Structures*.

2.9.1.3 Some remarks on *enhanced* Data Structures

As pointed before, Python's standard library, in addition to all user-created libraries, hugely expand the regular possibilities offered by Python and the properties of its fundamental Data Structures. An exhaustive review of these improved Data Structures is out of the scope of this section. Instead, we will highlight those that we find more interesting for the user (if one is interested in learning more about non-standard Data Structures, this *tutorial* covers the most common modules used to store data in Python).

- `types.MappingProxyType` modifies regular *dictionary* lists, making **immutable** versions of the original dictionary's data (`MappingProxyType` module is a dictionary *decorator*). Like tuples, `MappingProxyType` objects are useful when defining dictionary views that do not allow further modifications.

```
>>> from types import MappingProxyType
>>> my_dict = {
              "Alice": "spoon",
              "Bob": "fork",
              "Charlie": "knife"
              }
>>> view_dict = MappingProxyType(my_dict)

>>> view_dict["Charlie"] = "peeler"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
```

- `collections.namedtuple` increases the functionalities offered by regular tuples. This module allows you to define *field names* for each record entry. In this way, the developer ensures that the correct variable or data structure enters into each of the tuple slots. `namedtuple` objects are implemented as Python *classes* and are immutable sequences. Information stored inside `namedtuple` is accessed via the field name used to identify the entry (similarly to Python's dictionaries).

```
>>> from collections import namedtuple
>>> Family = namedtuple("Family", "mother father sister brother")
>>> my_family = Family("Marge", "Homer", "Lisa", "Bart")

>>> my_family
Family(mother='Marge', father='Homer', sister='Lisa', brother='Bart')

>>> my_family.sister
'Lisa'
```

- `collections.Counter` is a built-in class of the Python standard library. It defines a set object in which multiple occurrences of the set elements are allowed. This fact turns handy when one is interested in counting the number of appearances of the components included in the set.

```
>>> from collections import Counter
>>> shopping_list = Counter()

>>> food = {"apple": 4, "bread": 2, "milk": 3}
>>> shopping_list.update(food)
>>> shopping_list
Counter({'apple': 4, 'milk': 3, 'bread': 2})

>>> more_food = {"apple": 7}
>>> shopping_list.update(more_food)
>>> shopping_list
```

(continues on next page)

(continued from previous page)

```
Counter({'apple': 11, 'milk': 3, 'bread': 2})
```

- `collections.deque` is an optimized list that allows one to efficiently add and remove elements from both the top and the bottom of the sequence. `deque` collections are doubly-linked lists, which makes them excellent containers to support *LIFO* and *FIFO* semantics.

```
>>> from collections import deque
>>> party_list = deque()

>>> party_list.append("Lisa")
>>> party_list.append("George")
>>> party_list.append("Alfred")
>>> party_list.append("Miller")

>>> party_list.pop()
'Miller'

>>> party_list.popleft()
'Lisa'
```

2.10 Bibliography

2.10.1 Python

2.10.1.1 Introduction to Python

Articles and tutorials recommended for a soft introduction to Python.

2.10.1.2 Data-Structures

Online tutorials explaining the fundamental concepts behind Python Data Structures.

2.10.1.2.1 Other references

Within the Introductory chapter of Python, one could find references related to specific features or objects of Python. Along these lines you will find the corresponding sources referencing these objects.

2.11 My topic (Tool, process, technique. . .)

2.11.1 Overview

The overview section provides a starting point to understand the topic described in the template.

2.11.1.1 Introduction

- Short description of the topic assuming the reader has no prior knowledge of the topic
- Estimated time to understand the topic
- Estimated level of difficulty (Introductory, Easy, Medium, Advanced, Pro)
- Brief description of the core concepts explained in the current topic

2.11.1.2 TLRD

- Super quick summary of the contents of `topic.md`.

2.11.1.3 Goals and use cases

- Description of why the topic is useful
- Description of what you can achieve when understanding the topic

2.11.1.4 Pros, Cons & alternatives

- Description of the main benefits of the topic
- Description of the main drawbacks of the topic
- Trade-offs & implications of using the topic

2.11.1.5 Requirements and background

- Recommended prerequisites to understand the topic and a classification of its difficulty.
- Tools used to utilize and understand the topic.
- Other related topics
- Alternatives to the topic, if any.

2.11.1.6 Learning

- Advice on how to learn about the topic.
- Description of external resources to learn about the topic.
- Introductory tutorials

2.11.1.7 Reference

- Advice on resources that come in handy when using the topic for people familiar with it.

2.12 Topic

This document describes the topic of the current template. There are no restrictions about its format.

2.13 How to

This section includes all the tutorials for using or applying the template's topic.

The format of the tutorials can vary: code snippets, markdown, notebooks, presentations... All of them are valid as long as they help to solve an issue related to the topic.

2.14 FAQ

This document answers questions related to the topic such as:

- Noob questions to understand better the fundamentals
- Common problems faced by learning the topic.
- Features and use cases of the topic
- Answers to recurring problems related to the topic

2.15 Debugging

This document contains a compendium of common problems related to the topic, and how to solve them.

For example

2.15.1 Error when loading file

- **Stacktrace**

```
----> 1 with open("non_existent_file", "rb") as f:
      2     f.read()
      3

FileNotFoundError: [Errno 2] No such file or directory:
```

- **Why it's happening**

This error is caused by trying to open a file that cannot be found.

- **How to solve it**

  Make sure that the path to your target file is correct, and the file already exists.

2.16 Bibliography

This section contains all external links that have been referenced in the other documents of this template.

All the references will be added with the following format:

1. Reference description [Reference name] TLDR (few sentences) description of the reference. [PAPER CITATION]

If the reference is to scientific paper, cite the paper following academia standards after the description.

search

BIBLIOGRAPHY

[git_vadim_presentation] A set of slides that explain the main advantages of using version control and the basic concepts needed to understand git. It is an introductory tutorial illustrating the importance of using Git in a collaborative environment. It incorporates a summary of the essential commands, and helpful links to Git bibliography and interactive courses.

http://vmarkovtsev.github.io/mipt_web_2015/02_git/index.html

[git_handbook] An article containing a nice introduction to Git. It describes the basic commands and its most used features.

https://guides.github.com/introduction/git-handbook/

[learn_git_resources] A collection of resources to learn about Git.

https://try.github.io/

[git_lab_fundamentals] Interactive lab to learn the fundamentals of git.

https://gitimmersion.com/lab_01.html

[git_lab_branching] Interactive lab to learn how git branching works.

https://learngitbranching.js.org/

[course_using_git] Free online Git course. After completing it you will understand Staging, cloning, branching, and collaborating with Git.

https://www.pluralsight.com/courses/code-school-git-real

[saving-changes] Introductory tutorial to the saving process in Git.

https://www.atlassian.com/git/tutorials/saving-changes

[recording-changes] Page containing a complete description about the three-stage process in Git.

https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

[three-trees] Page containing an introduction to Git's internal state management system and reset command.

https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

[branch] Explains the basics about branching in Git.

https://www.atlassian.com/git/tutorials/using-branches

[merge-basic] Gentle tutorial explaining the concepts and mechanisms behind *git merge*.

https://www.atlassian.com/git/tutorials/using-branches/git-merge

[rebase-basic] Tutorial explaining the concepts and mechanisms behind *git rebase*.

https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase

[rebase-vs-merge] A clear overview over the two merging strategies.

https://www.atlassian.com/git/tutorials/merging-vs-rebasing

[git_cheat_sheet] Cheat sheet containing the most common Git commands. Recommended as a quick reference guide for solving common issues or remembering how to write a command.

https://education.github.com/git-cheat-sheet-education.pdf

[pro_git_book] Contains everything there is to know about Git. It's pretty hardcore, so we recommend using it as a last resort.

https://git-scm.com/book/en/v2

[interactive_rebase] Interactive rebase with Gitkraken.

https://support.gitkraken.com/working-with-repositories/interactive-rebase/

[rewriting_git_history] Explains how to rewrite git history.

https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

[rewriting_git_history2] Tutorial listing the rewriting options Git offers.

https://www.atlassian.com/git/tutorials/rewriting-history#git-rebase-i

[sync_remote] Explains how to work with remote repositories. Description of *git push* and *git pull* commands.

https://www.atlassian.com/git/tutorials/syncing

[push] *Git push* documentation page.

https://git-scm.com/docs/git-push

[pull] *Git pull* documentation page.

https://git-scm.com/docs/git-pull

[git-stash] Explains how to stash uncommitted changes with *git stash*.

https://www.atlassian.com/git/tutorials/saving-changes/git-stash

[log] Page listing the options offered by *git log*. https://www.atlassian.com/git/tutorials/inspecting-a-repository

[about_pull_requests] Explains the basics about Pull requests.

https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

[creating_a_pull_requests] Tutorial about how to open a new pull request.

https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-a-pull-request

[commenting_on_pull_requests] Explains how to use the GitHub interface to comment on a pull request.

https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/commenting-on-a-pull-request

[referencing_issue_and_pr] Explains how to reference issues from PRs and vice-versa.

https://docs.github.com/en/github/writing-on-github/autolinked-references-and-urls#issues-and-pull-requests

[issues-tools] List of available features on Issues page.

https://guides.github.com/features/issues/

[write-issue] Good practices for writing a proper GitHub issue.
https://medium.com/nyc-planning-digital/writing-a-proper-github-issue-97427d62a20f

[intro_code_reviews] Introductory article to code reviews containing useful tips.
https://www.evoketechnologies.com/blog/simple-effective-code-review-tips/

[code_review_checklist_1] Blog post containing a check list for performing more efficient code reviews.
https://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/

[code_review_checklist_2] Blog post containing a check list describing different principles of code reviews.
https://dev.to/codemouse92/10-principles-of-a-good-code-review-2eg

[nicer_code_reviews] Blog post containing tips for encouraging contributions and being nice during a code review.
https://developers.redhat.com/blog/2019/07/08/10-tips-for-reviewing-code-you-dont-like/

[new_ssh_key] Explains how to generate a new ssh key so you can log in to GitHub with it.
https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

[adding_ssh_key] Explains how to add a new ssh key to your GitHub account.
https://docs.github.com/en/github/authenticating-to-github/adding-a-new-ssh-key-to-your-github-account

[generating_gh_tokens] Explains how to use authentication tokens to avoid 2FA errors.
https://medium.com/@ginnyfahs/github-error-authentication-failed-from-command-line-3a545bfd0ca8

[github_docs] GitHub.com official documentation.
https://docs.github.com/en/github

[python_web] Python official web page. It contains the official Python documentation along with multiple resources.
https://www.python.org/about/

[python_tutorial] Complete article describing Python and its main features.
https://www.tutorialandexample.com/python-tutorial/

[python_features] In-depth article explaining the core blueprints on which Python language was developed.
https://www.tutorialandexample.com/features-of-python/

[python_medium] Light article explaining the benefits of Python and why it is so broadly adopted.
https://medium.com/@trungluongquang/why-python-is-popular-despite-being-super-slow-83a8320412a9

[data_struct1] Introductory article about common Python Data Structures.
https://www.tutorialandexample.com/python-data-structures/

[data_struct2] Complete tutorial that covers Primitive and Non-primitive Python Data Structures.
https://realpython.com/python-data-structures/#toc

[string] https://realpython.com/python-data-types/#strings

[boolean] https://realpython.com/python-boolean/

[hash] https://docs.python.org/3/glossary.html#term-hashable

[ooa_wrapper] https://realpython.com/primer-on-python-decorators/#simple-decorators

[ooa_classes] https://realpython.com/primer-on-python-decorators/#first-class-objects

[lifo] https://realpython.com/python-data-structures/#stacks-lifos

[fifo] https://realpython.com/python-data-structures/#queues-fifos