

# 实验一 Git和Markdown基础

班级： 21计科02

学号： B20210302209

姓名： 章丽媛

Github地址：

[https://github.com/shixiaoxiya/py\\_course\\_zly\\_/blob/main/experiments\\_pdf/experiment1.pdf](https://github.com/shixiaoxiya/py_course_zly_/blob/main/experiments_pdf/experiment1.pdf)

## 实验目的

1. Git基础，使用Git进行版本控制
2. Markdown基础，使用Markdown进行文档编辑

## 实验环境

1. Git
2. VSCode
3. VSCode插件

## 实验内容和步骤

### 第一部分 实验环境的安装

1. 安装git，从git官网下载后直接点击可以安装：[git官网地址](#)
2. 从Github克隆课程的仓库：[课程的仓库地址](#)，运行git bash应用（该应用包含在git安装包内），在命令行输入下面的命令（命令运行成功后，课程仓库会默认存放在Windows的用户文件夹下）

```
git clone https://github.com/zhoujing204/python_course.git
```

如果你在使用 `git clone` 命令时遇到SSL错误，请运行下面的git命令(这里假设你的Git使用了默认安装目录)：

```
git config --global http.sslCAInfo C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
```

该仓库的课程材料后续会有更新，如果需要更新课程材料，可以在本地课程仓库的目录下运行下面的命令：

```
git pull
```

3. 注册Github账号，创建一个新的仓库，用于存放实验报告和实验代码。

4. 安装VScode，下载地址：[Visual Studio Code](#)

5. 安装下列VScode插件

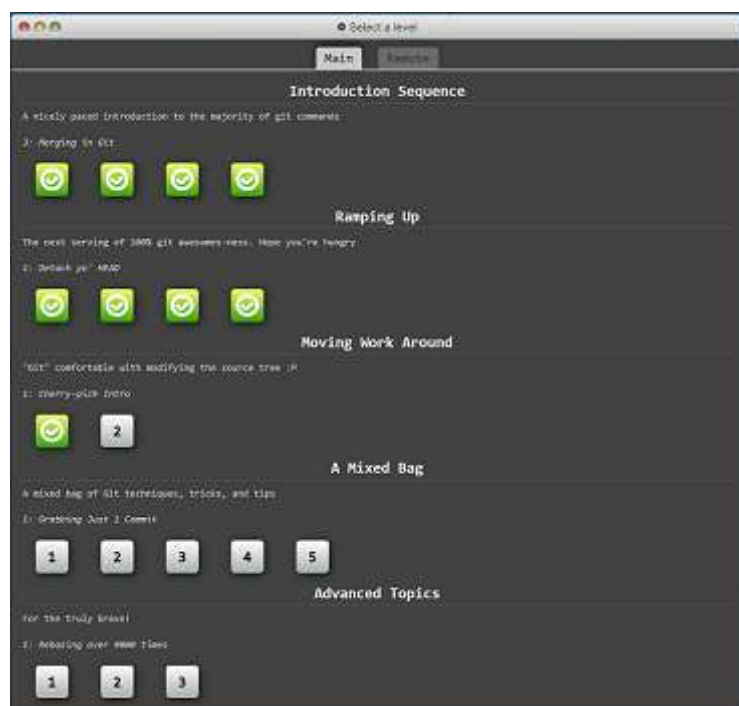
- GitLens
- Git Graph
- Git History
- Markdown All in One
- Markdown Preview Enhanced
- Markdown PDF
- Auto-Open Markdown Preview
- Paste Image
- markdownlint

## 第二部分 Git基础

教材《Python编程从入门到实践》P436附录D：使用Git进行版本控制，按照教材的步骤，完成Git基础的学习。

## 第三部分 [learngitbranching.js.org](https://learngitbranching.js.org)

访问[learngitbranching.js.org](https://learngitbranching.js.org)，如下图所示完成Main部分的Introduction Sequence和Ramping Up两个小节的学习。



上面你学习到的git命令基本上可以应付百分之九十以上的日常使用，如果你想继续深入学习git，可以：

- 继续学习[learngitbranching.js.org](https://learngitbranching.js.org)后面的几个小节（包括Main和Remote）
- 在日常的开发中使用git来管理你的代码和文档，用得越多，记得越牢
- 在git使用过程中，如果遇到任何问题，例如：错误删除了某个分支、从错误的分支拉取了内容等等，请查询[git-flight-rules](https://git-flight-rules.com)

## 第四部分 Markdown基础

查看[Markdown cheat-sheet](#)，学习Markdown的基础语法

使用Markdown编辑器（例如VScode）编写本次实验的实验报告，包括[实验过程与结果](#)、[实验考查](#)和[实验总结](#)，并将其导出为 **PDF格式** 来提交。

## 实验过程与结果

### 第一部分 实验环境的安装

顺利完成所有实验环境的安装。

### 第二部分 Git基础

#### D.1 配置git

Git命令：

```
git config --global user.name "shixiaoxiya"
git config --global user.email "shixiaoxiya@example.com"
```

#### D.2 创建项目

创建一个要进行版本控制的项目。在系统中创建一个文件夹，将其命名为git\_practice.在这个文件中，创建一个简单的Python程序：

```
print("Hello Git world")
```

#### D.3 忽略文件

创建一个名为.gitignore的特殊文件，并在 其中添加一行内容

```
_pycache_/
```

## D.4 初始化仓库

打开一个终端窗口，切换到git\_practice,并执行如下命令：

```
git init
```

运行结果，表明Git在git\_practice中初始化来了一个空仓库。

```
Initialized empty Git repository in C:/Users/章丽媛/Desktop/git_practice/.git/
```

```
章丽媛@LAPTOP-G6MUIKPK MINGW64 ~/Desktop/git_practice (master)
```

## D5 检查状态

执行其他操作前，看一下项目的状态：

```
git status
```

运行结果

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
.gitignore
```

```
prac.py
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

```
章丽媛@LAPTOP-G6MUIKPK MINGW64 ~/Desktop/git_practice (master)
```

## D6 将文件加入仓库

下面将这两个文件加入仓库，再次检查状态：

```
git add .
```

```
git status
```

运行结果

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: .gitignore

new file: prac.py

章丽媛@LAPTOP-G6MUIKPK MINGW64 ~/Desktop/git\_practice (master)

## D7 执行提交

下面来执行第一次提交：

```
git commit -m "Started project"
```

运行结果

```
[master (root-commit) 63c1ff8] Started project
2 files changed, 2 insertions(+)
create mode 100644 .gitignore
create mode 100644 prac.py
```

编写的Git代码

```
git status
```

运行结果

On branch master  
nothing to commit, working tree clean

章丽媛@LAPTOP-G6MUIKPK MINGW64 ~/Desktop/git\_practice (master)

## D8 查看历史提交记录

Git记录所有的项目提交。下面看一下提交历史：

```
git log
```

## 运行结果

```
commit 63c1ff860a791e0b286570863c9b985d2a0e46c4 (HEAD -> master)
Author: shixiaoxiya <shixiaoxiya@qq.com>
Date:   Mon Sep 18 14:54:47 2023 +0800
```

Started project

章丽媛@LAPTOP-G6MUIKPK MINGW64 ~/Desktop/git\_practice (master)

## D9 第二次提交

为了展示版本控制的强大威力，我们需要修改项目并提交所做的修改。在hello\_git.py中再添加一行代码：

```
print("Hello everyone.")
```

如果现在查看项目的状态，将发现Git这个文件发生了变化：

```
git status
git commit -am "Extended greeting"
git status
git log --pretty=online
```

## D10 放弃修改

下面来看看如何放弃所做的修改，恢复到上一个可行状态。首先在hell\_git.py中再添加一行代码：

```
print("Oh no,I broke the project!")
```

如果要放弃刚刚的修改，则可以终端执行如下命令：

```
git restore .
git status
```

## 运行结果

```
On branch master
nothing to commit, working tree clean
```

章丽媛@LAPTOP-G6MUIKPK MINGW64 ~/Desktop/git\_practice (master)

## D11 检出以前的提交

编写的Git代码

```
git log --pretty=oneline
git checkout cea13d
git switch
```

## D12 删除仓库

编写的Git代码

```
git status
rm -rf .git/
git init
git status
git add .
git commit -m "Starting over."
git status
```

## 第三部分 [learngitbranching.js.org](https://learngitbranching.js.org)

完成结果



### 一、基础篇

循序渐进地介绍Git主要命令：git commit; git branch; git merge; git rebase

#### 1. Git Commit

Git仓库中的提交记录保存（git commit）的是你的目录下所有文件的快照，就像是把整个目录复制，然后再粘贴一样，但比复制粘贴优雅许多。提交记录可以看作是项目的快照，它非常轻量，可以快速地在这些提交记录之间切换。

通关代码：

```
git commit //生成C2节点
git commit //生成C3节点
```

## 2. Git Branch

Git的分支也非常轻量。它们只是简单地指向某个提交记录——仅此而已。所以许多Git爱好者传颂：早建分支！多用分支！

这是因为即使创建再多的分支也不会造成储存或内存上的开销，并且按逻辑分解工作到不同的分支要比维护那些特别臃肿的分支简单多了。

在将分支和提交记录结合起来后，我们会看到两者如何协作。现在只要记住使用分支其实就相当于在说：“我想基于这个提交以及它所有的父提交进行新的工作。”

通关代码：

```
git branch bugFix //生成分支bugFix
git checkout bugFix //切换到bugFix分支上
```

## 3. Git Merge

将两个分支合并到一起的第一种方法

```
git checkout -b bugFix //创建并切换分支bugFix
git commit //把bugFix分支增加C2节点
git checkout main //切换到main上
git commit //在main分支增加C3节点
git merge bugFix //将bugFix分支的内容合并到main上
```

## 4. Git Rebase

将两个分支合并到一起的第二种方法

通关代码：

```
git checkout -b bugFix //创建并切换分支bugFix
git commit //在bugFix上增加C2节点
git checkout main //切换到main上
git commit //在main上增加C3节点
git checkout bugFix //切换到bugFix分支上
git rebase main //将bugFix分支上的内容合并到main上
```

# 二、高级篇

## 1. 分离HEAD

HEAD是一个对当前检出记录的符号引用——也就是指向正在其基础上工作的提交记录。

HEAD总是指向当前分支上最近一次提交记录。大多数修改提交书的Git命令都是从改变HEAD指向



开始的。HEAD通常情况下是指向分支名的（如bugFix）。在提交时，改变了bugFix的状态，这一变化通过HEAD变得可见。

通关代码：

```
git checkout c4
```

## 2. 相对引用 (^)

通过指定提交记录哈希值的方式在Git中移动不太方便。在实际应用中，并没有像本程序这么漂亮的可视化提交树供你参考，所以你就不得不用git log来查看提交记录的哈希值。并且哈希值在真实的Git世界中也会更长（基于SHA-1，共40位）。比较令人欣慰的是，Git对哈希的处理很智能，秩序提供唯一标识提交记录的前几个字符即可，因此可以仅输入fed2而不是上面的一长串字符。由于通过哈希值指定提交记录很不方便，所以Git引入了相对引用。这个就很厉害了。使用相对引用的话，可以从一个易于记忆的地方（比如bugFix分支或者HEAD）开始计算。

通关代码：

```
git checkout bugFix^  
或  
git checkout bugFix  
git checkout HEAD^
```

## 3. 相对引用2 (~)

如果想在提交树中向上移动很多步的话，敲很多^貌似也挺烦人的，git也考虑到了这一点，于是又引入了操作符~。该操作符后面可以跟一个数字（可选，不跟数字时与^相同，向上移动一次），指定向上移动多少次

通关代码：

```
git branch -f master c6 //分支master直接指向C6  
git branch -f bugFix c0  
git checkout c1 //HEAD指向C1
```

## 4. 撤销变更

在git里撤销变更的方法很多，和提交一样，撤销变更由底层部分（暂存区的独立文件或者片段）和上层部门（变更到底是通过哪种方式被撤销的）组成。我们这个应用主要关注的是后者。主要有两种方法用来撤销变更——一是git reset，还有就是git revert

通关代码：

```
git reset HEAD^ //撤销本地分支用reset  
git checkout pushed  
git revert HEAD //撤销远程分支用revert
```

# 第四部分 Markdown基础

查看[Markdown cheat-sheet](#)，学习Markdown的基础语法

使用Markdown编辑器（例如VScode）编写本次实验的实验报告，包括[实验过程与结果](#)、[实验考查](#)和[实验总结](#)，并将其导出为 **PDF格式** 来提交。

## 实验考查

请使用自己的语言回答下面的问题，这些问题将在实验检查时用于提问和答辩，并要求进行实际操作。

### 1. 什么是版本控制？使用Git作为版本控制软件有什么优点？

- 版本控制：

- 版本控制是一种追踪和管理文件、代码或项目变化的方法。在软件开发中，版本控制是一种非常重要的管理工具，可以帮助开发者追踪代码的更改和历史，以确保代码的质量和可维护性。它有助于记录文件的不同版本，并提供了一种协作工作的方式，多个人可以在同一个项目上同时工作而不会干扰彼此的工作。版本控制系统还允许您回溯到先前的项目状态，以便查找错误、比较不同版本以及管理项目的演化历史。

- 优点：

Git是一种广泛使用的版本控制系统，它允许开发团队成员在本地存储和管理代码库的更改，并与其他人协作开发项目。以下是使用Git作为版本控制软件的一些优点：

- 分布式版本控制：Git是分布式的，这意味着它不需要中央服务器来存储版本历史，而是使用分散在计算机网络上的多个存储库来存储数据。这种分布式特性使得用户可以随时随地访问和更新代码库，无需等待其他人的批准或中央服务器的响应。
- 强大的分支功能：Git的分支功能非常强大，用户可以轻松地创建、切换和使用不同的分支。这使得用户可以自由地组织和管理代码库，并行处理多个更改，而无需等待其他人的批准或中央服务器的响应。
- 高效性和可靠性：Git通过使用高效的存储和索引机制来提高开发效率，并确保数据可靠性。它可以在本地计算机上并行处理多个更改，并自动进行冲突检测和解决。这可以大大加快开发速度，并减少错误。
- 广泛使用和支持：Git是一种广泛使用的版本控制系统，因此有很多资源和文档可供参考，也有很多现成的工具和插件可以使用。此外，Git还有强大的社区和用户支持，可以帮助用户解决问题和学习新技能。
- 可扩展性和灵活性：Git具有高度的可扩展性和灵活性，可以适应不同的项目需求。它支持多种协议（如HTTP、SSH、FTP等）和存储后端（如磁盘、NFS、S3等），可以轻松地与其他工具和技术集成。
- 跨平台性：Git可以在不同的操作系统上运行，包括Windows、macOS和Linux。

### 2. 如何使用Git撤销还没有Commit的修改？如何使用Git检出（Checkout）已经以前的Commit？（实际操作）

- 撤销未commit的修改

- 查看未提交的修改：首先，可以使用以下命令查看尚未提交的修改，以确保您知道要撤销的内容。

```
git status
```

- 撤销单个文件的修改：如果您只想撤销单个文件的修改，可以使用以下命令：

```
git checkout -- 文件名
```

- 撤销所有尚未提交的修改：如果您想撤销所有尚未提交的修改，可以使用以下命令：

```
git reset --hard HEAD
```

- 对于已经Commit的修改，可以使用以下方法检出（Checkout）到以前的Commit：
  - 找到想要检出的Commit的ID，例如，想要检出到HEAD^，即上一次的Commit，可以使用git reset --hard HEAD^命令。

注意：git reset --hard HEAD^命令会删除所有位于该提交之上的更改，因此如果您想要保留这些更改，可能需要使用另一种方式来撤销这些更改。

### 3. Git中的HEAD是什么？如何让HEAD处于detached HEAD状态？（实际操作）

- HEAD是什么：

在Git中，HEAD是一个指向当前检出的分支的引用。当你执行git clone或git checkout操作时，Git会将HEAD设置为指向你正在检出的分支的最新提交。

- 如何让 HEAD 进入分离头指针状态：

有时你可能想要查看或编辑一个特定的提交，而不影响你当前检出的分支。在这种情况下，你可以将HEAD置于“detached HEAD”状态。在“detached HEAD”状态下，HEAD指向一个特定的提交，而不是一个分支。这意味着你可以自由地尝试提交，而不必担心会改变任何现有的分支。以下是如何将HEAD置于“detached HEAD”状态的步骤：

首先，找到你想要查看或编辑的提交的哈希值。你可以通过git log命令查看提交历史，并找到你感兴趣的提交的哈希值。

然后，使用git checkout命令并传入该哈希值，如下所示：

```
git checkout <commit-hash>
```

在“detached HEAD”状态下，你可以自由地查看和编辑代码。如果你想创建一个新的分支，基于当前的HEAD位置，你可以使用git branch命令，如下所示：

```
git branch <new-branch-name>
```

如果你想返回到一个特定的分支，你可以使用git checkout命令并传入分支名称，如下所示：

```
git checkout <branch-name>
```

### 1. 什么是分支（Branch）？如何创建分支？如何切换分支？（实际操作）

在版本控制系统中，分支是一种将开发线分离的方式，以防止在一个开发线上进行的更改影响到其他开发线。在Git中，分支是一种轻量级、独立的线，它包含提交和其他开发线的引用。

创建分支的步骤如下：

首先，确定当前所在的分支。你可以使用以下命令查看当前分支：

```
git branch
```

然后，使用git branch命令创建一个新的分支，如下所示：

```
git branch <new-branch-name>
```

可以使用git checkout命令切换到你新创建的分支，如下所示：

```
git checkout <new-branch-name>
```

现在，已经在新的分支上工作了。可以在这个分支上进行更改并提交，而不会影响到你当前所在的分支。当准备好将更改合并回主分支（或其他分支）时，可以使用git merge命令。

## 2. 如何合并分支？git merge和git rebase的区别在哪里？（实际操作）

在Git中，你可以使用git merge或git rebase命令来合并分支。这两个命令有一些重要的区别，具体使用哪一个取决于你的需求和团队的工作流程。

git merge命令将两个分支合并在一起。当你使用git merge时，Git会创建一个新的提交，这个提交包含了从一个分支到另一个分支的所有更改。这个新的提交包含了两个父提交：一个是当前分支的最新提交，另一个是你要合并的分支的最新提交。

git rebase命令将一个分支的更改应用到另一个分支上。当你使用git rebase时，Git会取出你要合并的分支上的每个提交，并将它们依次应用到当前分支上。这意味着，如果你要合并的分支上的提交与当前分支有冲突，你可能需要解决这些冲突。

这是一个简单的示例，说明如何使用这两个命令：

使用git merge合并分支：

```
git checkout <branch-to-merge-into>
git merge <branch-to-merge>
```

使用git rebase合并分支：

```
git checkout <branch-to-rebase>
git rebase <branch-to-rebase-onto>
```

重要提示：如果你在公共分支上使用git rebase，并且你的同事也在同一个分支上工作，这可能会导致问题。因为git rebase会改变提交的SHA哈希值，这可能导致你的同事在拉取更新后遇到冲突或问题。因此，对于公共分支，通常推荐使用git merge。

## 3. 如何在Markdown格式的文本中使用标题、数字列表、无序列表和超链接？（实际操作）

在Markdown格式的文本中，您可以使用以下方式创建标题、数字列表、无序列表和超链接：

标题：在Markdown中，标题是由井号(#)表示的。一个#代表一级标题，两个##代表二级标题，以此类推，直到六级标题。例如：

# 这是一级标题  
## 这是二级标题  
### 这是三级标题

数字列表：数字列表在Markdown中是通过在行首添加数字和一个英文句点创建的。数字的顺序不影响列表的排序。例如：

1. 这是第一项
2. 这是第二项
3. 这是第三项

无序列表：无序列表在Markdown中是通过在行首添加破折号或星号创建的。例如：

- 这是第一项
- 这是第二项
- 这是第三项

超链接：在Markdown中，超链接是通过方括号和圆括号创建的。方括号中包含链接的文本，圆括号中包含链接的URL。例如：

[点击这里访问GitHub](https://github.com/)

还可以为链接添加一个可选的标题，将标题放在引号中：

[点击这里访问GitHub](https://github.com/ "GitHub首页")

## 实验总结

总结一下这次实验你学习和使用到的知识，例如：编程工具的使用、数据结构、程序语言的语法、算法、编程技巧、编程思想。

- 编程工具的使用：这个实验主要使用了Git和Markdown。Git是一种版本控制系统，可以帮助程序员追踪和管理代码的变化。学会了如何在本地初始化Git仓库，以及如何配置Git全局设置。Markdown是一种轻量级的标记语言，用于编写格式化的文本。
- 数据结构：Markdown文件中的文本内容和格式信息可以视为一种数据结构。在这个实验中，你需要理解和操作这种数据结构，以便正确地编写和格式化文本。
- 程序语言的语法：Markdown有自己的语法规则，需要理解这些规则才能正确地编写Markdown文件。
- 算法：在这个实验中，需要理解和应用Git的工作流程，包括如何创建和合并分支，初始化仓库、添加文件、提交更改、查看历史记录、如何提交和推送代码等。
- 编程技巧：需要掌握一些编程技巧，例如如何安装和使用Git和Markdown编辑器，如何在编辑器中创建和编辑Markdown文件等。学习了如何与远程Git仓库进行交互，包括克隆远程仓库、推送

本地更改和拉取远程更改。

了解了协作开发时的常见工作流程，如合并请求（Pull Request）等。

- 编程思想：这个实验涉及到了版本控制和文档编写的思想。需要理解这些思想，以便正确地使用Git和Markdown来管理代码和文档。

总的来说，这次实验强调了版本控制、分支管理、冲突解决和文本排版等编程和文档处理的重要概念和技能。这些技能对于日后的软件开发和协作工作将非常有用。