O Mínimo Essencial sobre Git e GitHub

Este é um guia rápido sobre Git e GitHub com o objetivo de fornecer o conhecimento essencial para organizar códigos de projetos pessoais e facilitar a colaboração em projetos com poucos colegas.

Se você não quer perder uma versão do seu código e não gostaria de lidar com arquivos como `projeto_final.py`, `projeto_final_revisado.py`, `projeto_final_final_mesmo.py`... saber Git e GitHub pode ajudar bastante!

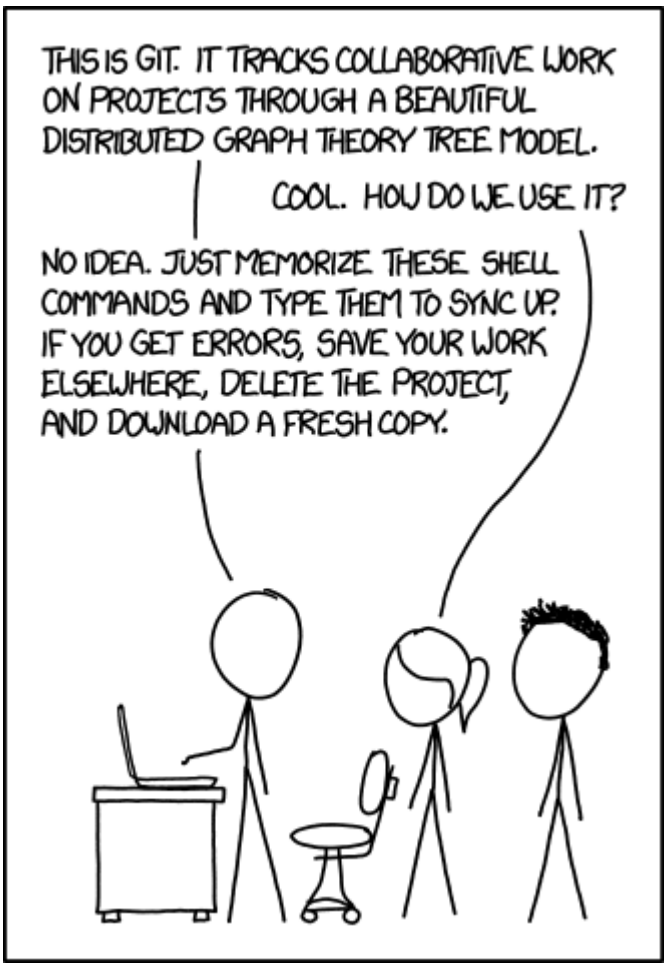
1. O que é Git?

É um software local de controle de versão, o que significa que ele registra todas as mudanças que você faz em seus projetos (códigos, arquivos, diretórios, etc.) ao longo do tempo.

Por que isso é útil?

- Mantém um **histórico das mudanças**, quem mudou e por quê (se bem documentado);
- Faz com que seja possível **voltar atrás facilmente** para uma versão do código que funciona, caso algo dê errado;
- Permite que várias pessoas **colaborem** no mesmo projeto ao mesmo tempo sem sobrescrever o trabalho uma das outras.

2. Comandos Básicos de Git



Estes são os comandos mais comuns de Git:

- `git init`: inicia um novo **repositório Git** em um diretório. A partir daqui o Git consegue rastrear as mudanças nesse diretório.
- `git add <nome_do_arquivo>`: informa ao Git quais arquivos novos/modificados você quer que ele "prepare" para serem salvos na próxima versão.
  - adiciona os arquivos na área de preparação ("staging")
  - `git add .`: adiciona todos os arquivos modificados.
- `git commit -m "uma_mensagem_bonita"`: salva as mudanças "preparadas" como uma nova versão (checkpoint) no histórico do projeto.
  - A mensagem é importante como documentação do que foi feito. Exemplo: `git commit -m "Implementação do sistema linear"`.
- `git status`: mostra o estado atual do repositório Git: quais arquivos foram modificados desde a última versão (commit), quais estão "preparados" para o próximo commit, etc.
- `git log`: mostra o histórico de todos os commits.
  - `git log --all --oneline --graph`: adicionar essas opções faz com que o histórico seja mostrado de forma mais compacta e com uma visualização da estrutura dos commits, considerando todas as branches (vistas na seção 7.).

Observação: Na primeira vez que você instalar o Git, use estes comandos para adicionar sua identificação, que será associada aos seus commits:

```
git config --global user.name "Nome Sobrenome"
git config --global user.email seuemail@examplo.com
```

3. O que é GitHub?

O Git é um software de controle de versão local, enquanto o GitHub é uma **nuvem** onde você pode armazenar e compartilhar seus respositórios Git.

Por que isso é útil?

- Hospeda seus repositórios Git online, permitindo que você os recupere de qualquer lugar
- Facilita a colaboração
- Oferece uma interface visual para gerenciar os repositórios
- Pode ser usado como um portfólio

4. Como utilizar o GitHub?

4.1 Subindo Projeto Local para o GitHub

1. **Crie um repositório no GitHub**
   - Botão `+` e clicar em `New repository`
   - Dar um nome ao repositóio (ex.: `dissertation-experiments`)
   - Existem opções para adicionar um arquivo `README`, `.gitignore` e escolher uma licença
     - Em princípio, não precisam ser marcados

2. **Conecte seu projeto local ao GitHub**

   Criado o repoisitório, ele mostrará alguns comandos, vamos usar estes:

   ```
   git remote add origin https://github.com/seu-usuario/seu-repositorio.git
   git branch -M main
   git push -u origin main
   ```

   No repositório local (o diretório em que você fez `git init`), copie e cole esses comandos.

   O que eles fazem?

   - `git remote add origin <link_para_o_repositorio_remoto>`: diz ao seu Git local onde está o repositório remoto no GitHub e o chama de `origin` (nome usual mas poderia ser qualquer outro).
   - `git branch -M main`: renomeia sua branch principal para `main` (esse nome é `master` por padrão, mas `main` passou a ser adotado).
   - `git push -u origin main`: envia (push) seus commits locais para o GitHub pela primeira vez (a opção `-u` pareia a branch `main` local e a `main` do repositório remoto, de forma que das próximas vezes você só precisa escrever `git push`).

   Pronto! Seu projeto agora está no GitHub.

4.2 Colaborando em um Projeto do GitHub

Para trabalhar em um projeto que já está no GitHub, alguns comandos são úteis:

1. `git clone <link_do_repositório>`: baixa uma cópia completa do repositório remoto.
2. `git pull`: atualiza seu projeto local com as últimas mudanças que foram enviadas para o GitHub.
3. `git push`: envia as mudanças locais (commits) para o repositório remoto

Ciclo de trabalho comum:

1. `git pull` (para ter as últimas mudanças)
2. Fazer alterações nos arquivos
3. `git add .`
4. `git commit -m "sua mensagem sobre as alterações"`
5. `git push`

5. [Bônus] Trabalhando em projetos de terceiros

Você consegue trabalhar normalmente com repositórios do GitHub que você criou ou que você tenha permissão para editar. Caso você queira colaborar em um projeto público ou que você não tenha permissão, você têm algumas opções:

- **Opção 1: Criar uma cópia independente do repositório:**

Se você quer trabalhar no projeto de forma totalmente independente a partir do projeto original:

1. `git clone <link_do_repositório>`
2. Dentro da pasta do projeto que você clonou, "desconecte" o repositório local do repositório remoto:

```
git remote remove origin
```

Para verificar se ele foi removido, liste os repositórios remotos associados ao seu repositório local (deve estar vazio): `git remote -v`.

3. Crie um novo repositório vazio no GitHub (conforme 4.1-1)
4. Adicione o seu novo repositório vazio como o novo "remote" `origin` e envie (`push`) todo o conteúdo local para o seu novo repositório no GitHub (conforme 4.1-2).
5. Siga seu ciclo de trabalho normalmente, as mudanças serão enviadas para o seu novo repositório, sem qualquer ligação direta com o projeto original.

- **Opção 2: Criar um "fork" do repositório:**

Um "fork" também cria uma cópia pessoal de um projeto de terceiro:

1. Vá até a página do repositório no GitHub e clique no botão "Fork"
2. Agora que você tem sua própria cópia (seu "fork") na sua conta do GitHub, use o `git clone` para baixá-la: `git clone <link_do_seu_repositorio_forkeado>`
3. A partir daí, você pode seguir seu ciclo de trabalho normalmente com o novo repositório. As mudanças serão salvas no seu repositório "forkeado".

Qual a diferença principal entre as duas opções?

- A **Opção 1** é para quando você quer uma cópia e transformá-la em um projeto **totalmente seu e independente**, sem se preocupar com o projeto original. **Exemplo:** o projeto original foi de um trabalho anterior que não recebe mais atualizações de outras pessoas.
- A **Opção 2 (Fork)** é para quando você quer uma cópia, mas **mantendo uma ponte com o projeto original**. Muito utilizado em desenvolvimento de código aberto pois permite que você:
  - Sugira suas mudanças de volta para o projeto original, permitindo que o mantenedor revise e incorpore seu trabalho (chamado de "Pull Request" - veja a seção "Creating a Pull Request" do Pro Git para mais informações).
  - Atualize facilmente sua cópia com as últimas mudanças que o projeto original possa receber através da interface do GitHub.

6. Arquivos `.gitignore` e `README.md`

6.1 `README.md`

É um arquivo de documentação do projeto. Ele serve como "capa" do repositório no GitHub, sendo o primeiro arquivo que aparece. Ele é útil para apresentar o projeto, dar instruções de como instalar, configurar ou rodar o código, etc.

6.2 `.gitignore`

Um arquivo especial que diz ao Git quais arquivos/diretórios ele deve ignorar (não rastrear). Útil para:

- Arquivos temporários
- Arquivos com senhas/chaves
- Arquivos muito grandes
- Dados gerados automaticamente

Como criar?

1. Crie um arquivo chamado `.gitignore`
2. Adicione os padrões de arquivos a ignorar
    - Exemplo de conteúdo:

```
*.tmp
minhas_anotações.txt
data/raw/
.venv
```

3. `git add .gitignore` e `git commit -m "Adiciona .gitignore"`

7. Branches

Imagine que os commits formam uma árvore. A `main` é o tronco principal. Uma **branch** é uma cópia independente (ramificação) de um commit onde você pode fazer experimentos, desenvolver novas funcionalidades ou corrigir bugs sem afetar o funcionamento do tronco principal.

Por que usar **branches**?

- Testar ideias sem afetar o código principal
- Colaboração

Comandos relacionados à branches:

- `git branch`: lista todas as branches, destacando a que você está usando atualmente;
- `git branch <nome_da_nova_branch>`: cria uma nova branch.
- `git checkout <nome_da_branch>`: muda para uma branch existente.
- `git checkout -b <nome_da_nova_branch>`: cria e já muda para a nova branch (atalho).
- `git merge <nome_da_branch>`: combina as mudanças de uma branch para a branch atual. Normalmente, você volta para a `main` e mescla a sua branch de trabalho nela.

Ciclo de trabalho simples com branches:

1. `git pull`
2. `git checkout -b minha-nova-funcionalidade`
3. Faça alterações, `git add .`, `git commit -m "minhas mudanças"` (podem ser feitos mais de um commit, inclusive).
4. `git checkout main`
5. `git merge minha-nova-funcionalidade`
6. `git push`

8. Resolução de Conflito

Um conflito acontece quando duas pessoas (ou você e você do passado) modificam a mesma parte do código do mesmo arquivo de maneiras diferentes, e o Git não sabe qual versão manter.

Conflitos ocorrem quando você faz `git pull` ou `git merge`. O Git avisará que ocorreu o conflito e os arquivos com conflito passam a ter marcadores especiais desse tipo:

```
<<<<<<< HEAD
// A versão atual do trecho de código
=======
// Versão remota (ou de outra branch)
>>>>>>> nome-da-branch-de-conflito
```

Como resolver conflitos?

1. Abra o arquivo com conflito em um editor de texto
2. Decida qual versão você quer manter (ou combine as duas)
3. Remova os marcadores (o VSCode -- ou outras IDEs -- já permitem fazer isso automaticamente) e salve o arquivo.
4. `git commit -m "Resolve conflito no arquivo meu_codigo.py"`
5. `git push`

A melhor forma de evitar grandes conflitos é fazer `git pull` com frequência e evitar grandes mudanças a cada `git push` ou `merge`.

9. Considerações finais

Resta agora praticar! A melhor forma de aprender é fazendo no dia-a-dia. Não precisa ter medo de errar, o Git foi feito para experimentar e voltar atrás. Se algo der muito errado, você encontra muitos recursos online ou pode pedir ajuda.

Alguns links interessantes:

- Pro Git - livro com conteúdo completo disponível gratuitamente.
- Learn Git Branching - jogo no navegador que ajuda a visualizar o modelo de dados do Git.
- Introduction to Git in VS Code - o VS Code e outras IDEs fornecem uma interface para usar o Git sem precisar do terminal. Acho que é útil começar com os comandos para saber exatamente o que está acontecendo, mas usar a interface pode facilitar a vida.

- Tutoriais em vídeo:
  - Iniciante - freeCodeCamp - "Git and GitHub for Beginners - Crash Course"
  - Intermediário - freeCodeCamp - "Git for Professionals Tutorial - Tools & Concepts for Mastering Version Control with Git"
- How to explain git in simple words? - post de blog tentando explicar os conceitos de Git de forma simples.
- Oh Shit, Git!?! - ter problemas é comum...

Espero que tenha sido útil! =)