

1 Introdução

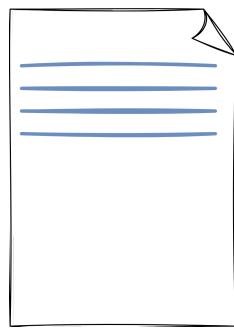
Neste material, estudaremos sobre as principais funcionalidades oferecidas pelo sistema de controle de versão chamado “Git”. Para tal, construiremos um sistema simples utilizando a linguagem de programação Python.

2 Desenvolvimento

2.1 Intuição: controle de versão manual de um documento Considere que você está escrevendo um documento como uma dissertação de mestrado, uma tese de doutorado, um programa de computador ou mesmo uma notícia. Estes são alguns exemplos de documentos cuja elaboração pode levar alguns dias ou meses e cujo **conteúdo evolui ao longo do tempo**.

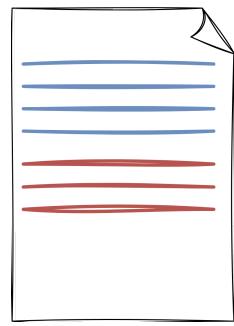
Após criar uma primeira **versão** do documento, para que possa continuar trabalhando nele posteriormente, você vai, obviamente, armazená-lo em meio persistente. Veja a Figura 2.1.1.

Figura 2.1.1



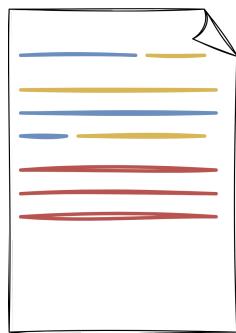
Depois disso, num próximo dia de trabalho, você edita o arquivo um pouco mais, obtendo o resultado exibido pela Figura 2.1.2.

Figura 2.1.2



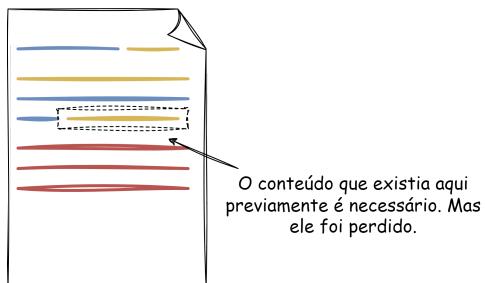
No próximo dia de trabalho, você deseja fazer algumas correções no conteúdo já existente. Você abre o arquivo, trabalha nele e depois armazena as alterações em meio persistente. Veja a Figura 2.1.3.

Figura 2.1.3



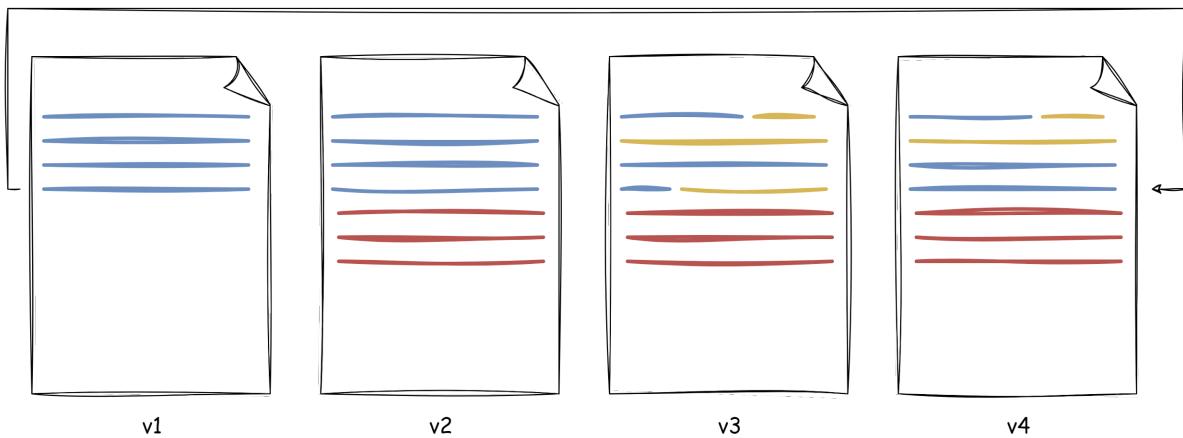
No próximo dia, **você percebe que partes do conteúdo removido são desejáveis novamente**. Como você já armazenou as alterações e fechou o editor de texto, elas não podem ser mais recuperadas. Veja a Figura 2.1.4.

Figura 2.1.4



A partir deste momento, você decide fazer uma espécie de **controle de versão manual**. No final de cada dia de trabalho, você armazena todo o conteúdo em um novo arquivo, mantendo todos os outros criados previamente. Assim, se necessário, você pode recuperar conteúdo antigo. Veja a Figura 2.1.5.

Figura 2.1.5
Agora o conteúdo pode ser facilmente recuperado de uma versão antiga.



Embora funcione, ao longo do tempo, este **controle de versão manual** pode ser um tanto **difícil de manter**. Especialmente se muitas pessoas forem trabalhar simultaneamente com o arquivo. Veja algumas dificuldades.

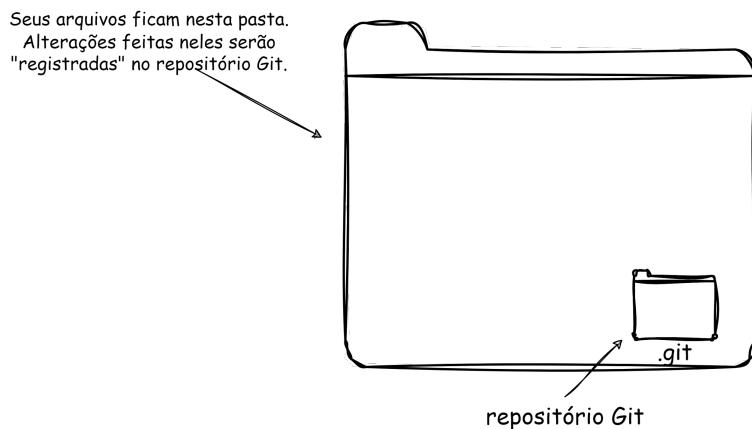
- mesclar conteúdo produzido por pessoas diferentes em uma única versão
- esquecer de criar uma nova versão do arquivo antes de editar conteúdo existente
- perder conteúdo ao sobrescrever um arquivo sem intenção

É aí que entram os sistemas de controle de versão. Há diversos deles. Para o desenvolvimento de software, o **Git** é o mais utilizado. Veja algumas das vantagens que sistemas de controle de versão como o Git oferecem.

- voltar arquivos para versões anteriores, recuperando conteúdo sem perder o atual
- comparar alterações realizadas ao longo do tempo
- verificar quem foi o responsável por uma alteração que pode estar causando algum problema
- recuperar arquivos “perdidos” (se o sistema for utilizado direitinho)

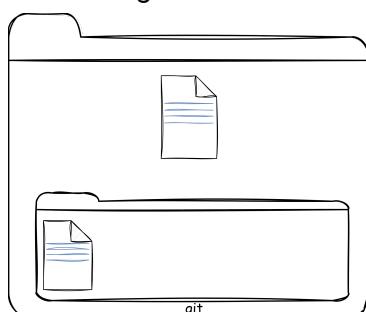
2.2 Controle de versão de um documento usando o Git O primeiro passo é criar uma pasta que servirá de abrigo para os arquivos de interesse. Ela conterá uma sub pasta oculta chamada **.git**. Ela é o **repositório Git** e ali ficam registradas as informações sobre o controle de versão realizado pelo Git. Veja a Figura 2.2.1.

Figura 2.2.1



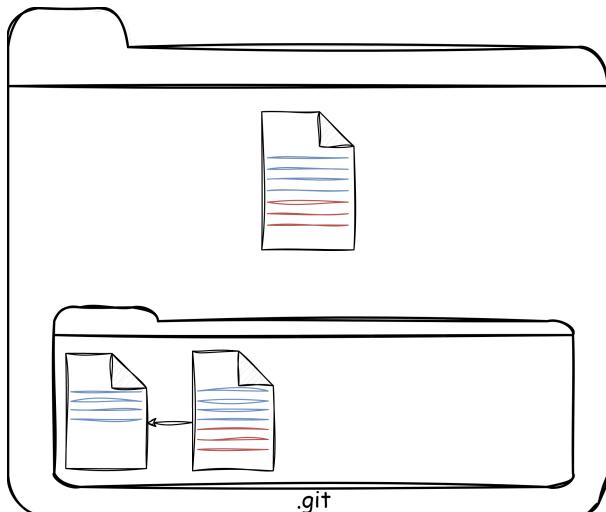
A seguir, digamos que você crie um primeiro arquivo. Ao final do primeiro dia de trabalho, você **torna as suas alterações permanentes**, fazendo uma operação do Git denominada **commit**. Uma vez que o faça, você pode fazer novas edições no arquivo, sabendo que o conteúdo atual está armazenado no repositório Git e pode ser recuperado a qualquer momento. Veja a Figura 2.2.2.

Figura 2.2.2



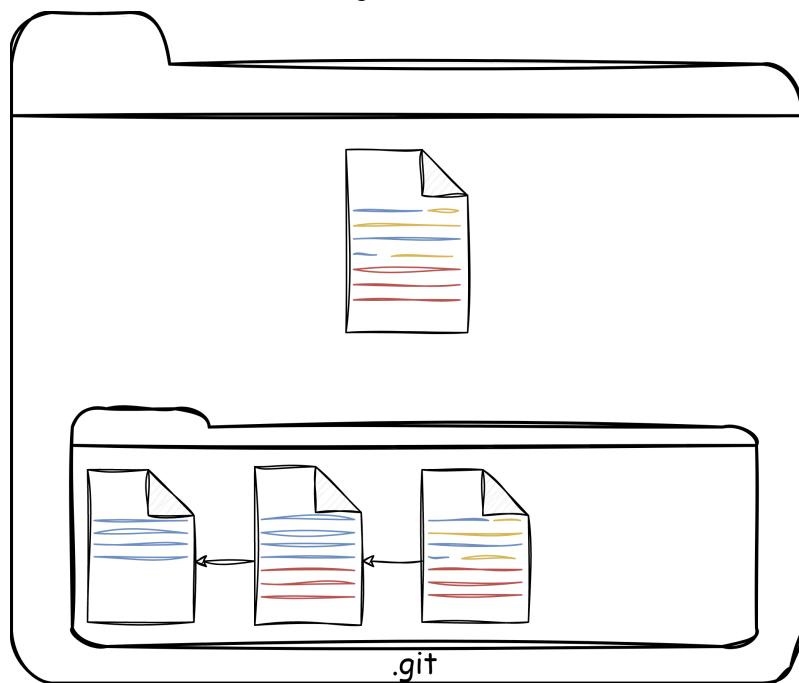
No segundo dia de trabalho, você faz novas edições no arquivo, tornando-as permanentes com mais uma operação **commit**. O seu diretório atual conterá somente o arquivo editado. Entretanto, o repositório Git conterá as duas versões. Observe como ele armazena um "ponteiro" para o conteúdo anterior, permitindo a navegação entre versões, se necessário. Veja a Figura 2.2.3.

Figura 2.2.3



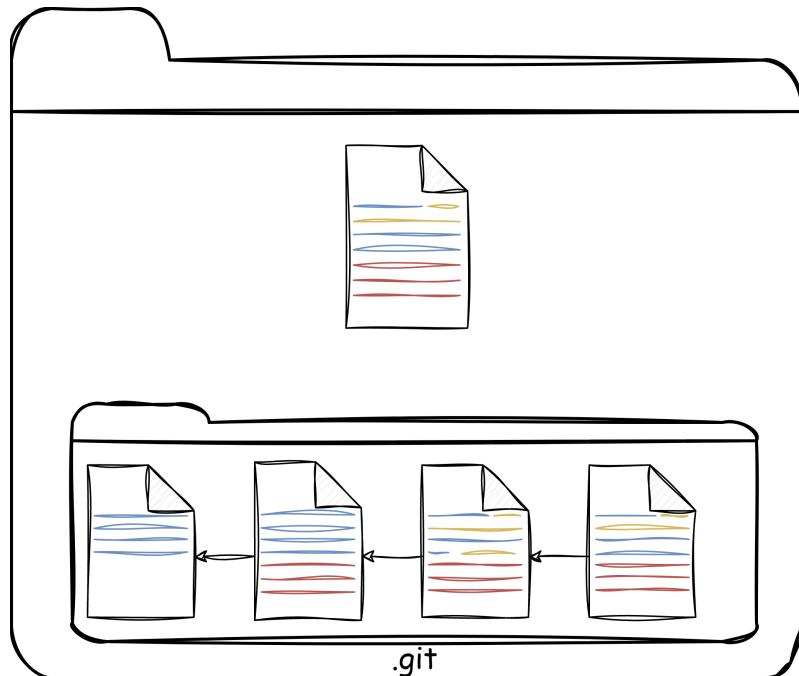
Em mais um dia de trabalho, você faz as edições desejadas e as torna permanentes, fazendo um novo **commit**. O resultado é aquele exibido pela Figura 2.2.4.

Figura 2.2.4



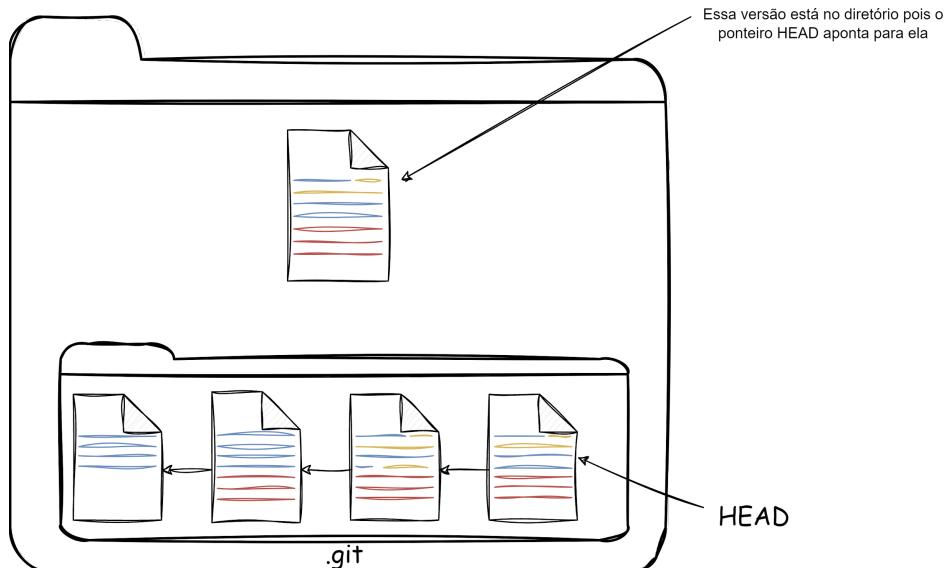
Agora, observe como o conteúdo mantido pelo Git em seu repositório torna simples a recuperação de conteúdos de versões antigas. Veja a Figura 2.2.5.

Figura 2.2.5



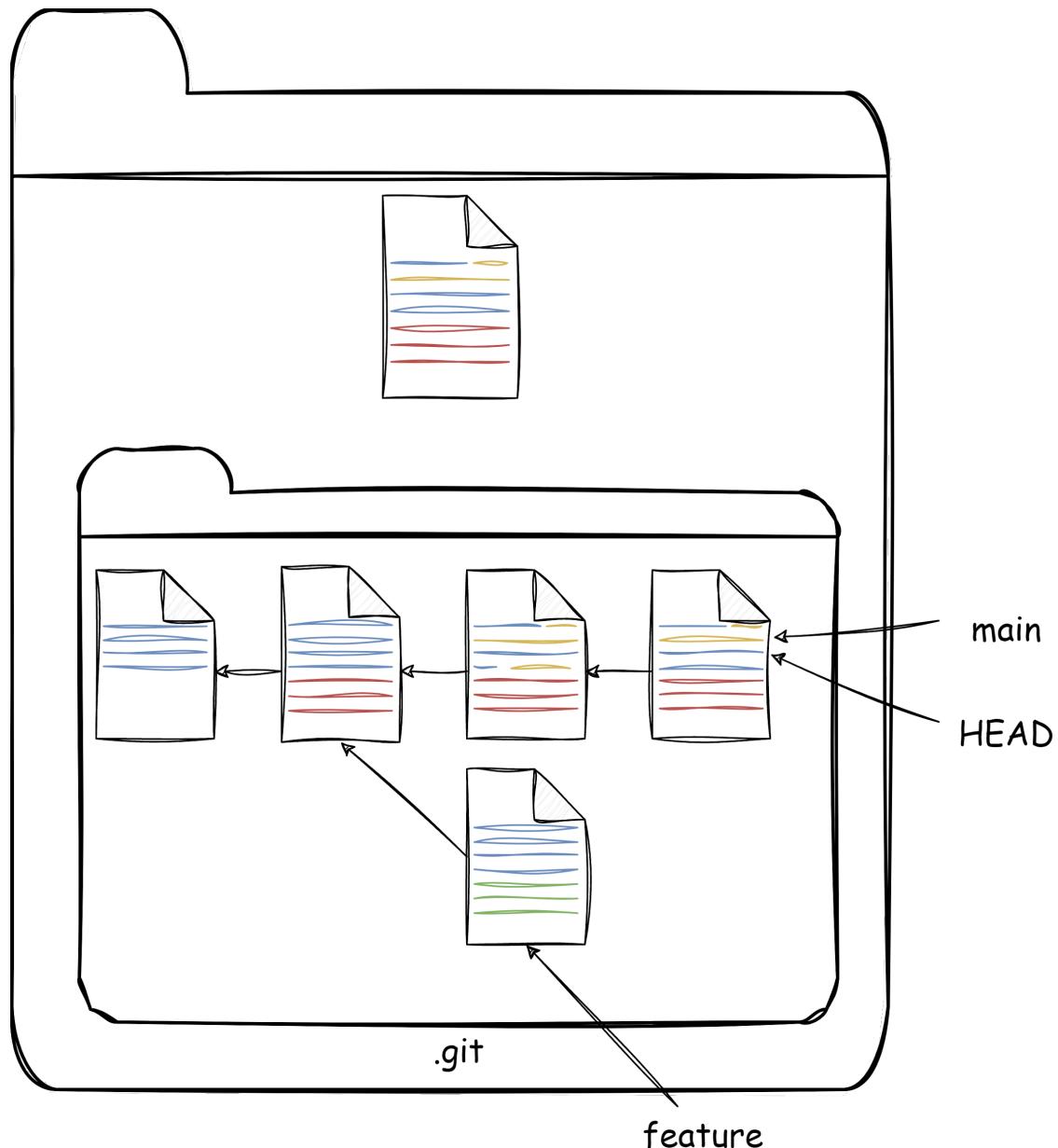
2.3 Branches e o ponteiro HEAD Como o git decide a versão do arquivo a ser mantida no diretório de trabalho? Há um ponteiro chamado HEAD que faz referência à versão atual. Veja a Figura 2.3.1.

Figura 2.3.1



Neste contexto, há também o conceito de **branch**. Uma branch também é um simples ponteiro que referencia um determinado “commit”. Um único repositório Git pode possuir múltiplas branches e esse é um cenário muito comum no desenvolvimento de software. A Figura 2.3.2 mostra um cenário em que há duas branches: a principal (main) e uma outra, chamada (feature). A ideia é ilustrar que, a partir de uma determinada versão, dois rumos diferentes podem ter sido seguidos, dando origem a versões diferentes do arquivo. Em algum momento no futuro, essas versões podem ser mescladas com a ajuda do Git.

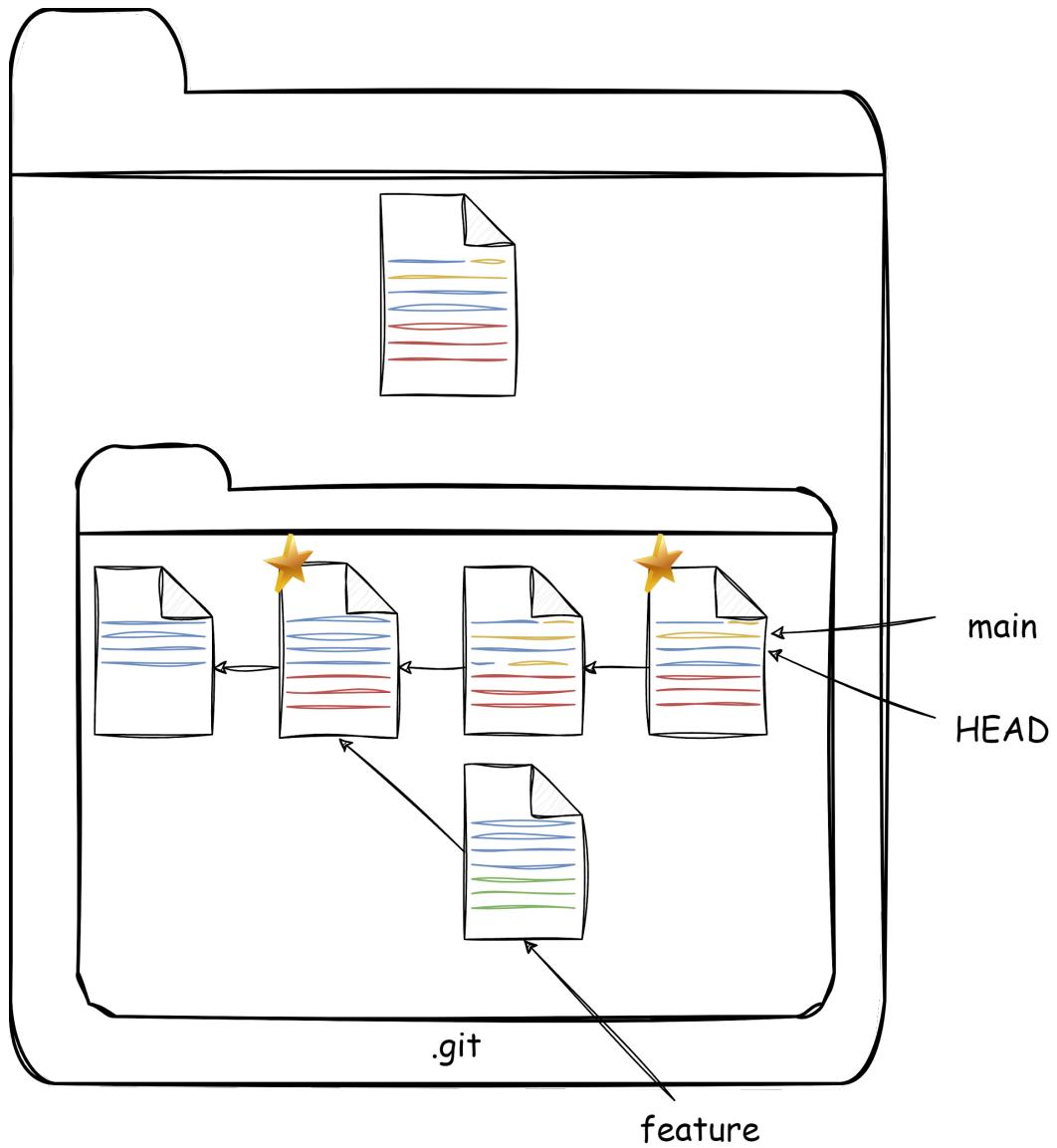
Figura 2.3.2



2.4 Tags Ao longo do desenvolvimento, é natural que tenhamos uma versão final “entregável”. Dependendo do tipo do projeto, é possível também que tenhamos entregas parciais. O Git oferece um mecanismo denominado “tag” que nos permite destacar um commit, mostrando que ele representa algo notável. Ou seja, é um mecanismo muito

interessante para fazer a liberação de versões de interesse. Veja a Figura 2.4.1. As versões com uma “estrela” foram marcadas com uma tag do Git, indicando que elas são importantes.

Figura 2.4.1



2.5 Repatórios remotos (remotes) Observe que todo o controle de versão discutido até então é feito completamente localmente. O que ocorre se

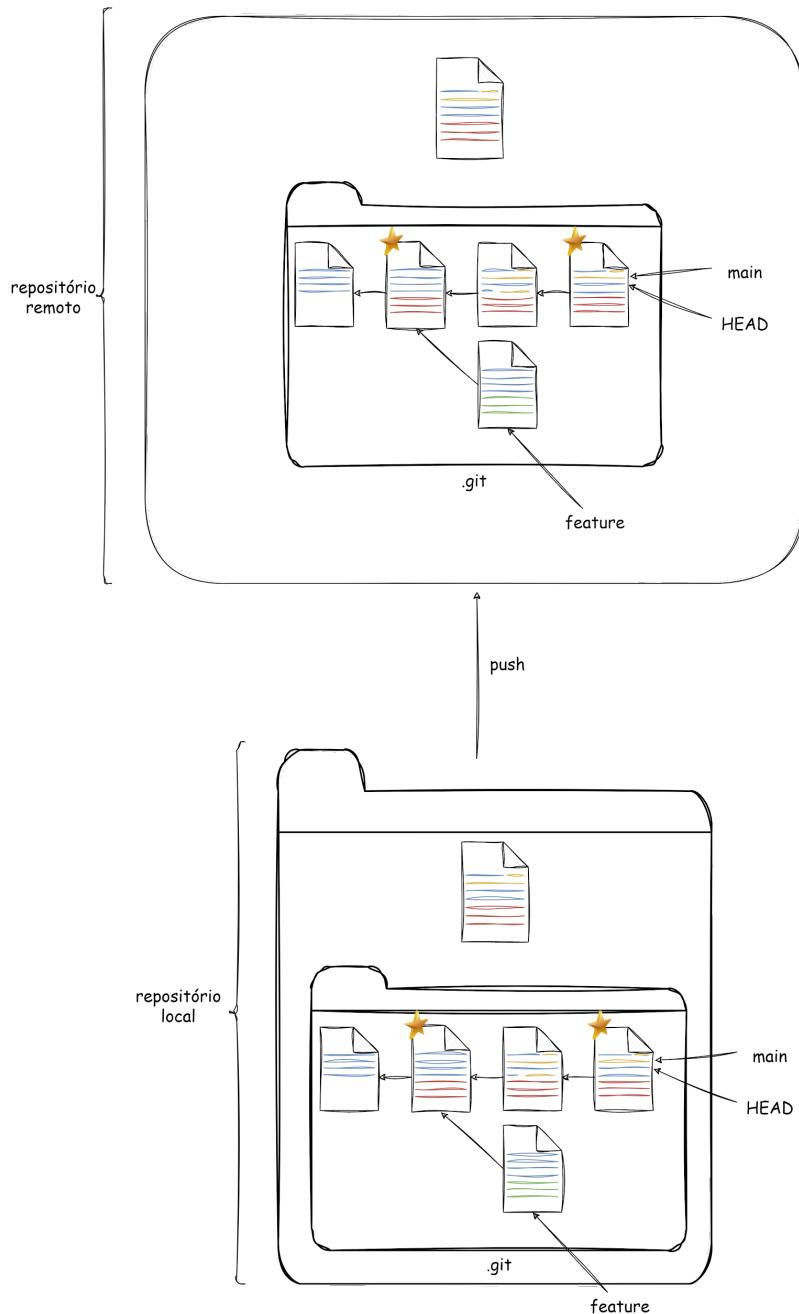
- desejarmos compartilhar o repositório com outras pessoas, para que possam trabalhar simultaneamente?
- desejarmos fazer um backup de todo o controle de versão?

Em geral, utilizamos um repositório remoto para isso. Há diferentes provedores de computação em nuvem que oferecem o Git como serviço, viabilizando o backup, o compartilhamento e muito mais coisas, incluindo estruturas próprias para Devops. Veja alguns exemplos:

- github.com
- gitlab.com
- bitbucket.org

A ideia é bastante simples. Criamos um repositório remoto e, utilizando comandos próprios do Git, fazemos “upload” do conteúdo de nosso repositório local. Esta é uma operação chamada “**push**”. Veja a Figura 2.5.1.

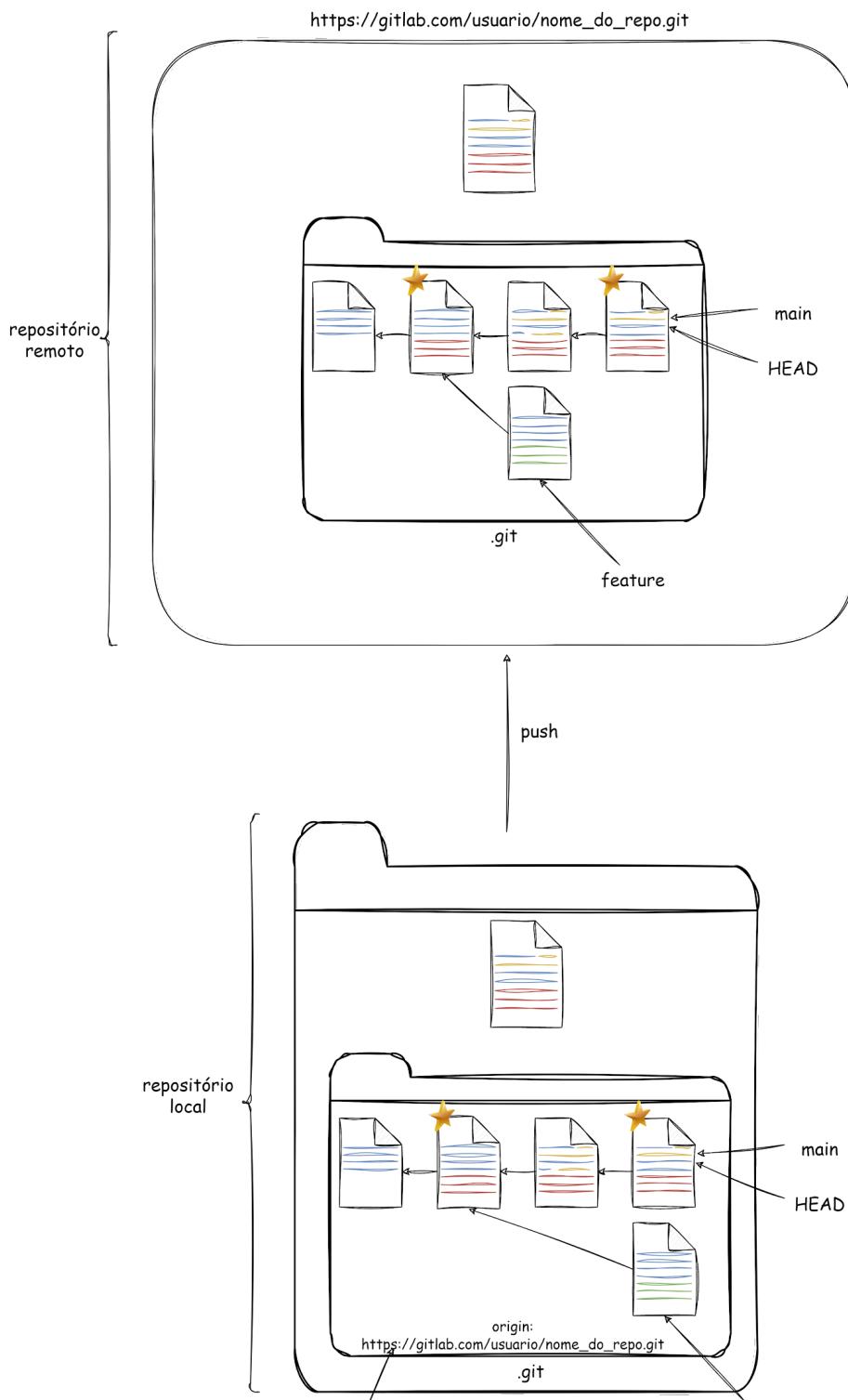
Figura 2.5.1



Quando criamos um repositório remoto, ele tem um link de acesso. Claro, precisamos desse link para fazer as operações de upload(push). Ocorre que seria muito pouco prático ter de digitar o link todas as vezes que fossemos fazer um push. É possível armazenar o link de um repositório remoto associado a um simples nome no repositório local e, a partir daí, usar apenas o nome escolhido no momento de fazer push. Veja a Figura 2.5.2.

Figura 2.5.2

Link do repositório remoto. Neste exemplo utilizamos o Gitlab. Porém, a ideia é a mesma para qualquer outro.



O link do repositório remoto fica armazenado no repositório local, associado a um nome que escolhemos. Aqui, usamos "origin". Mas pode ser qualquer outro nome.

Nota. O nome **origin** vem de “originally”. A ideia é que podemos fazer o clone de um repositório remoto criando uma cópia local e o nome indica que ela foi “originalmente” clonada de tal repositório. Seu uso é bastante comum. Mas você pode usar qualquer outro, como “gitlab”, “github” etc.

O Git é um sistema de controle de versão muito poderoso e ele oferece inúmeras outras funcionalidades que estão fora do escopo deste documento.

2.6 Um exemplo prático com Git e Python Neste exemplo, vamos implementar um programa em Python bastante simples. A ideia é a seguinte:

- O programa implementa as funcionalidades básicas de uma calculadora
- As versões que desejamos tornar permanentes são as seguintes. Observe o destaque naquelas que serão marcadas como especiais por serem entregáveis.
 - aquela que faz apenas a soma
 - aquela que testa a operação soma (entregável)
 - aquela que faz soma e subtração
 - aquela que testa as operações soma e subtração (entregável)
 - aquela que faz soma, subtração e multiplicação
 - aquela que testa as operações soma, subtração e multiplicação (entregável)
 - aquela que faz soma, subtração, multiplicação e divisão
 - aquela que testa as quatro operações (entregável)

Novo diretório, vínculo com o VS Code e terminal interno Comece criando um novo diretório. É importante que ele seja criado visando evitar os mecanismos de segurança do sistema operacional. Por isso, caso esteja usando o Windows, uma boa ideia pode ser utilizar o diretório Documents do usuário logado atualmente. Pode ser algo assim:

C:\Users\usuario\Documents\dev\introducao_git

Em sistemas Unix-like, você pode criar algo assim:

/home/usuario/dev/introducao_git

Uma vez que tenha criado o seu diretório, abra o VS Code e clique em **File >> Open Folder**. Navegue no sistema de arquivos e vincule o VS Code ao diretório. Veja as figuras 2.6.1 e 2.6.2.

Figura 2.6.1

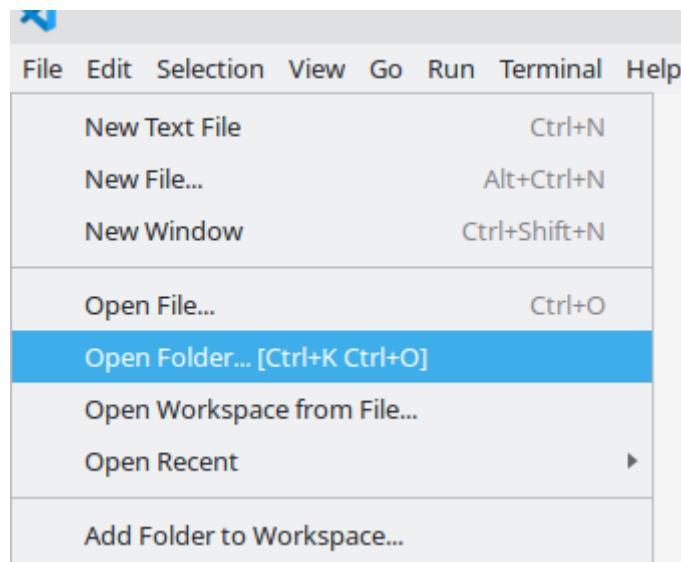
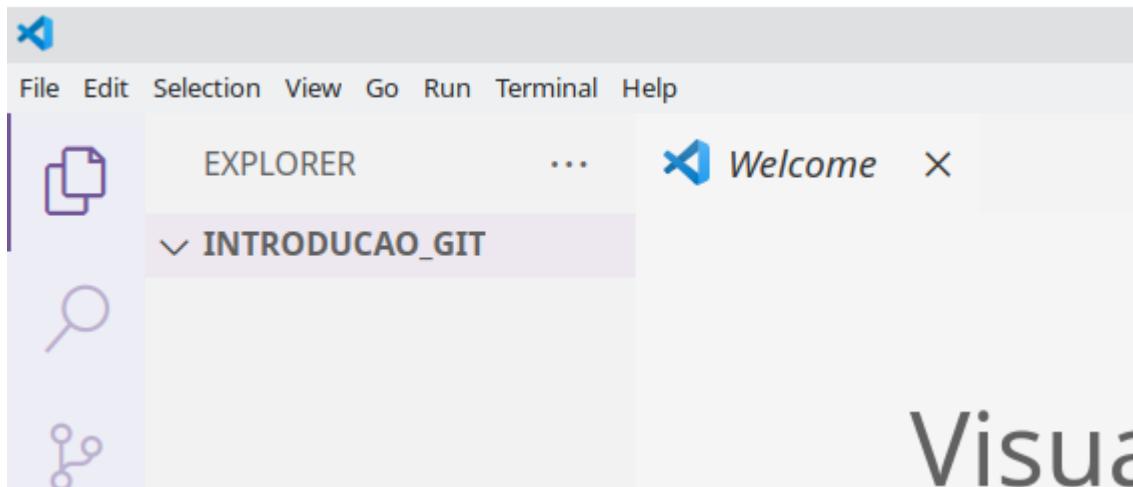
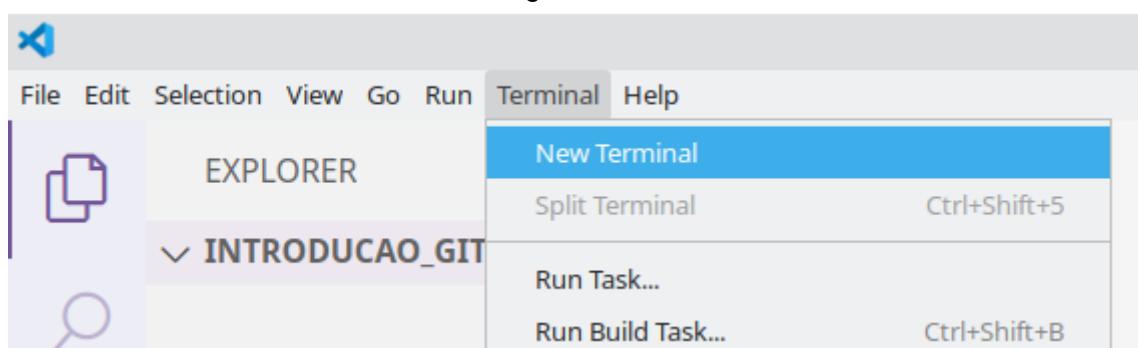


Figura 2.6.2



Observe como o nome da pasta deve aparecer no Explorer do VS Code. A seguir, clique **Terminal >> New Terminal**, como na Figura 2.6.3. Isso abrirá um terminal interno no VS Code, facilitando nossos trabalhos.

Figura 2.6.3



Criação do repositório Git (a pasta .git) A criação do repositório é basta simples. No terminal, use

```
git init
```

Algumas configurações Git Tudo aquilo que fazemos no Git deve ser identificado. Em geral, é preciso informar, pelo menos, o nome e o e-mail do responsável pela operação. Isso pode ser feito da seguinte forma

```
git config --local user.name "Rodrigo Bossini"  
git config --local user.email professorbossini@gmail.com
```

No futuro, quando fizermos nossos “commits” veremos que será necessário associar uma mensagem a cada um deles. Um pequeno pedaço de texto que descreve a razão de ser daquele commit. Isso pode ser feito com qualquer editor de texto e, utilizar o próprio VS Code é bastante simples. Para isso, vamos configurar o Git para utilizá-lo na edição de mensagens de commit.

```
git config --global core.editor "code --wait"
```

Nota. code é o comando para abrir uma aba do VS Code. O parâmetro wait indica que é preciso “esperar” até que o arquivo seja fechado para que o commit seja confirmado.

Quando fizermos nosso primeiro commit, criaremos também a nossa primeira branch. Toda branch tem um nome que podemos escolher. Há também um nome padrão, caso não escolhemos um. Historicamente, o nome **master** costumava ser utilizado. Entretanto, há uma iniciativa que sugere que o termo não seja mais utilizado, dando espaço para um nome mais inclusivo, evitando inclusive expressões comuns na computação como “master/slave”. Por isso, vamos configurar o Git para que ele use o nome **main** como padrão para as novas branches.

```
git config --global init.defaultBranch main
```

Se desejar visualizar as configurações realizadas, use

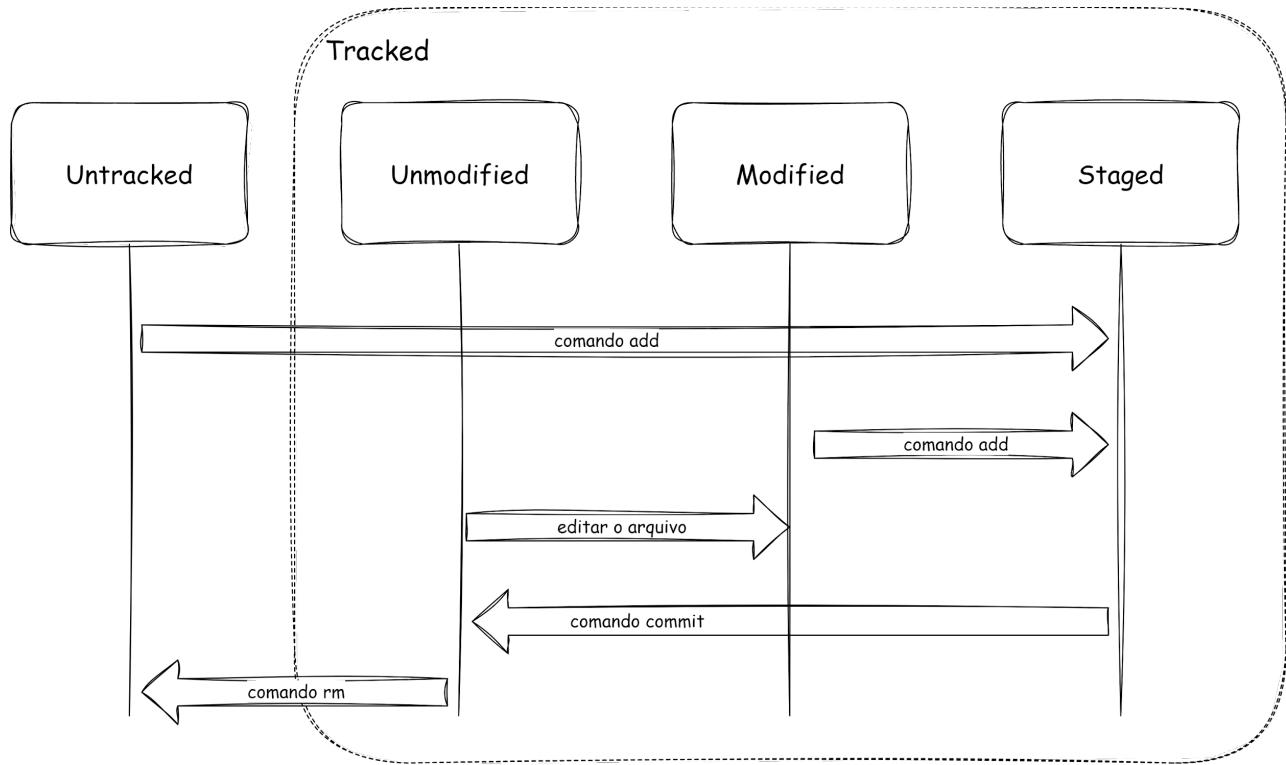
```
git config --list
```

ou ainda

```
git config --list --show-origins
```

2.7 Estado de um arquivo Cada arquivo sob controle de versão tem o seu “estado”. Para entender os comandos que utilizaremos a seguir, precisamos entendê-los. Veja a Figura 2.7.1.

Figura 2.7.1



Untracked: Arquivos neste estado se encontram no diretório de trabalho porém não estão sob controle de versão.

Tracked: Arquivos neste estado estão sob controle de versão. Observe que há três sub-estados deste.

Unmodified: A versão presente no diretório de trabalho é igual à última que foi tornada permanente.

Modified: A versão presente no diretório de trabalho está editada em relação à última que foi tornada permanente.

Staged: Arquivos neste estado estão prontos para participar do próximo commit. Somente arquivos neste estado são tornados permanentes quando um comando commit é executado.

2.7 Implementação da calculadora (operação de soma): No VS Code, crie um arquivo chamado **calculadora.py**. O Código 2.7.1 mostra a definição de uma função que faz a soma de dois valores recebidos como parâmetro.

Código 2.7.1

```
def somar (a, b):  
    return a + b
```

Observe que esta é a primeira versão a ser tornada permanente. Se desejar saber o estado do arquivo atual, use

git status

Também é possível verificar o estado do arquivo obtendo uma mensagem mais enxuta ou curta (short, daí a letra s)

git status -s

Agora precisamos fazer com que o arquivo passe a fazer parte do sistema de controle de versão.

git add calculadora.py

A seguir, consulte o seu estado novamente.

git status -s

Observe que ele está "Staged". Torne esta primeira versão permanente com

git commit

Você deverá ver uma aba do VS Code em que poderá digitar uma mensagem que descreve a razão de ser deste commit. Use algo como

apostila02(git-python): funcao somar implementada

Nota. Estamos utilizando uma convenção para a escrita de mensagens que descrevem commits. Leia mais sobre ela no Link 2.7.1.

Link 2.7.1

<https://github.com/angular/angular/blob/main/CONTRIBUTING.md#commit>

A seguir, salve o arquivo (ou clique File >> Auto Save) e feche a aba. Pronto, seu primeiro commit foi realizado!

Se perguntar sobre o estado novamente, você verá que o diretório de trabalho está “limpo”. Ou seja, não há mudanças que ainda não tenham sido tornadas permanentes.

git status -s

Se desejar, você pode visualizar a sua lista de commits.

git log

Se desejar ver o código contido (**patch**) em cada commit, use

git log -p

Teste também

git log --pretty=oneline

Visite o Link 2.7.2 para ver mais opções do comando git log.

Link 2.7.2

<https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>

2.8 Implementação da calculadora (teste da operação de soma e implementação da operação de subtração): Crie outro arquivo chamado **teste_calculadora.py**. Veja seu conteúdo inicial no Código 2.8.1.

Código 2.8.1

```
import calculadora  
print (calculadora.somar(1, 2))
```

Verifique o estado dos arquivos novamente

git status -s

Observe que cada arquivo tem o seu próprio estado.

Neste momento, seria interessante fazer um novo commit e torná-lo especial, marcando-o com uma tag. Entretanto, suponha que esqueçamos de fazê-lo e passemos a implementar a operação de subtração, no arquivo **calculadora.py**, como no Código 2.8.2.

Código 2.8.2

```
def somar (a, b):  
    return a + b  
  
def subtrair(a, b):  
    return a - b
```

Verifique o estado de cada arquivo.

```
git status -s
```

Observe que cada um tem o seu próprio estado. Como desejamos tornar permanente apenas o arquivo **teste_calculadora.py**, vamos transportá-lo para o estado staged.

```
git add teste_calculadora.py
```

Verifique o estado novamente.

```
git status -s
```

Agora podemos fazer um novo commit.

```
git commit
```

Use a seguinte mensagem.

apostila02(git-python): teste da função somar

Observe que agora apenas o arquivo **calculadora.py** está modificado. Verifique seu conteúdo modificado com

```
git diff
```

Veja que ele já possui a implementação da operação de subtração. Entretanto, antes de torná-lo permanente, vamos marcar o commit atual como importante, aplicando-lhe uma tag.

```
git tag -a v1.0.0 -m "operação de somar implementada e testada"
```

Nota. No Git, as tags podem ser **leves** ou **anotadas**. Uma tag leve é apenas um ponteiro temporário para algum commit. Uma tag anotada serve para destacar um commit como especial e ela inclui o nome do autor, uma mensagem, data etc. Leia mais no Link 2.8.1.

Link 2.8.1
<https://git-scm.com/book/en/v2/Git-Basics-Tagging>

Visualize a tag criada com

git tag --list

e também

git show v1.0.0

Verifique uma vez mais o estado dos arquivos.

git status -s

Observe que agora podemos tornar permanente a nova versão do arquivo **calculadora.py**. Se desejarmos, podemos “ignorar” o estado Staged se ele não for necessário. Veja.

git commit -a

Use a mensagem

apostila02(git-python): funcao subtrair implementada

Agora podemos implementar o teste da operação subtrair, tornar esta nova versão permanente e destacá-la com uma tag. Comece adicionando o teste, como no Código 2.8.3.

Código 2.8.3

```
import calculadora
print (calculadora.somar(1, 2))
print (calculadora.subtrair(1, 2))
```

Torne esta versão permanente e destaque-a como importante com

```
git add teste_calculadora.py
git commit -m "apostila02(git-python): teste da funcao subtrair"
git tag -a v1.0.1 -m "operação de subtrair implementada e testada"
```

Visualize as suas tags.

```
git tag
git show v1.0.0
git show v1.0.1
```

2.9 Implementação da calculadora (operação de multiplicação e teste): Comecemos pelo arquivo **calculadora.py**, implementando a operação de multiplicação, como no Código 2.9.1.

Código 2.9.1

```
def somar (a, b):  
    return a + b  
  
def subtrair(a, b):  
    return a - b  
  
def multiplicar(a, b):  
    return a * b
```

A seguir, implemente o teste no arquivo **teste_calculadora.py**, como no Código 2.9.2.

Código 2.9.2

```
import calculadora  
print (calculadora.somar(1, 2))  
print (calculadora.subtrair(1, 2))  
print (calculadora.multiplicar(1, 2))
```

Observe que desejamos dois commits. O primeiro deve incluir apenas a implementação da operação de multiplicação. O segundo deve incluir o teste. Por isso, vamos usar o estado Staged.

Leve o arquivo **calculadora.py** ao estado Staged com

```
git add calculadora.py
```

E torne esta versão permanente.

```
git commit -m "apostila02(git-python): funcao multiplicar implementada"
```

A seguir, leve o arquivo **teste_calculadora.py** ao estado Staged com

```
git add teste_calculadora.py
```

E torne esta versão permanente.

```
git commit -m "apostila02(git-python): teste da funcao multiplicar"
```

Além disso, torne esta versão especial com uma nova tag.

```
git tag -a v1.0.2 -m "operação de multiplicar implementada e testada"
```

Visualize seus commits.

```
git log --pretty=oneline
```

E as suas tags.

```
git tag  
git tag show v1.0.0  
git tag show v1.0.1  
git tag show v1.0.2
```

Verifique também o estado dos arquivos. Eles deverão estar todos “unmodified”.

```
git status -s
```

2.10 Implementação da calculadora (operação de divisão e teste): Repetiremos os passos para implementar a divisão e seu respectivo teste. Ajuste o arquivo **calculadora.py** como no Código 2.10.1.

Código 2.10.1

```
def somar (a, b):  
    return a + b  
  
def subtrair(a, b):  
    return a - b  
  
def multiplicar(a, b):  
    return a * b  
  
def dividir (a, b):  
    return a / b
```

Torne esta versão permanente com

```
git commit -a
```

E a mensagem de commit

```
apostila02(git-python): função dividir implementada
```

Agora, implemente o teste, como no Código 2.10.2. Estamos no arquivo **teste_calculadora.py**.

Código 2.10.2

```
import calculadora
print (calculadora.somar(1, 2))
print (calculadora.subtrair(1, 2))
print (calculadora.multiplicar(1, 2))
print (calculadora.dividir(1, 2))
```

Torne esta versão permanente com

```
git commit -a -m "apostila02/git-python): teste da funcao dividir"
```

E não deixe de marcá-la como importante.

```
git tag -a v1.0.3 -m "operação de dividir implementada e testada"
```

2.11 (Repositórios remotos com Gitlab) O próximo passo consiste em criar um repositório remoto e fazer “upload” de nosso repositório local. Comece visitando o Link 2.11.1 e crie uma conta Gitlab caso ainda não possua uma.

Link 2.11.1
<https://gitlab.com/>

Na página inicial, clique em **New Project**. A seguir, clique em **Create blank Project**.

Nota. Um projeto no Gitlab é uma coleção de recursos, incluindo

- um repositório
- configuração de CI/CD
- issues
- colaboradores

entre outras coisas.

A seguir, escolha um nome para o seu repositório. Marque-o como público e desmarque as caixinhas para que ela nasça sem conteúdo algum. A seguir, clique em **Create Project**. Veja a Figura 2.11.1.

Figura 2.11.1

Your work / Projects / New project / Create blank project

Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name
20231_pessoal_maua_vgti_tutorial_git_python

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL
https://gitlab.com/ professorbossini

Project slug
20231_pessoal_maua_vgti_tutorial_git_python

Pick a group or namespace where you want to create this project.

Want to organize several dependent projects under the same namespace? [Create a group](#).

Project deployment target (optional)
Select the deployment target

Visibility Level
 Private
Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.
 Public
The project can be accessed without any authentication.

Project Configuration

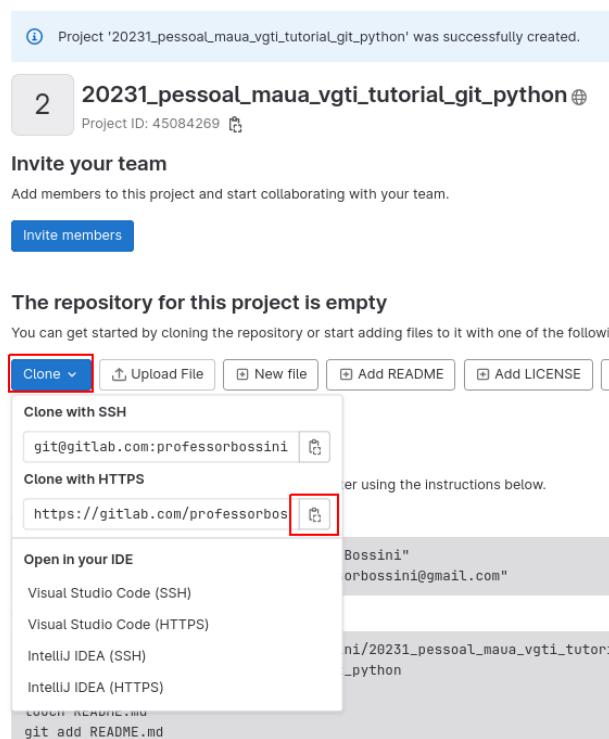
Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Enable Static Application Security Testing (SAST)
Analyze your source code for known security vulnerabilities. [Learn more](#)

Create project **Cancel**

Na tela seguinte, clique em **Clone >> Clone with HTTPS**, como na Figura 2.11.2. Observe que não estamos clonando coisa alguma. Estamos apenas obtendo o link do repositório remoto para que ele possa ser adicionado ao repositório local.

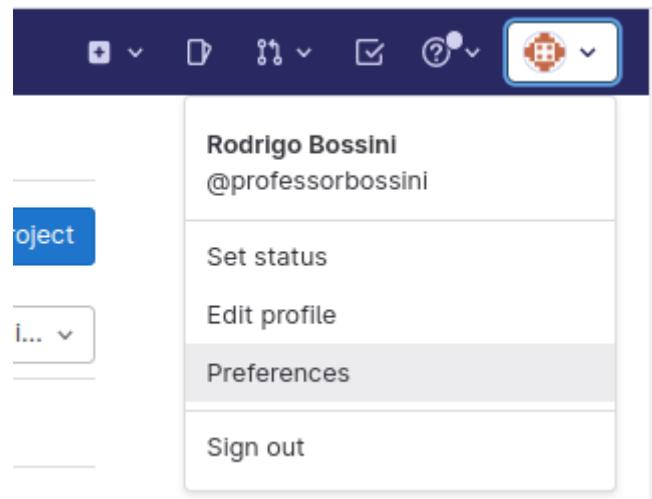
Figura 2.11.2



Uma vez que tenha copiado o link, volte ao VS Code e cole-o em um arquivo **temporário**. Vamos voltar ao Gitlab para gerar um token de autenticação, assim não será necessário digitar as credenciais para login a cada acesso.

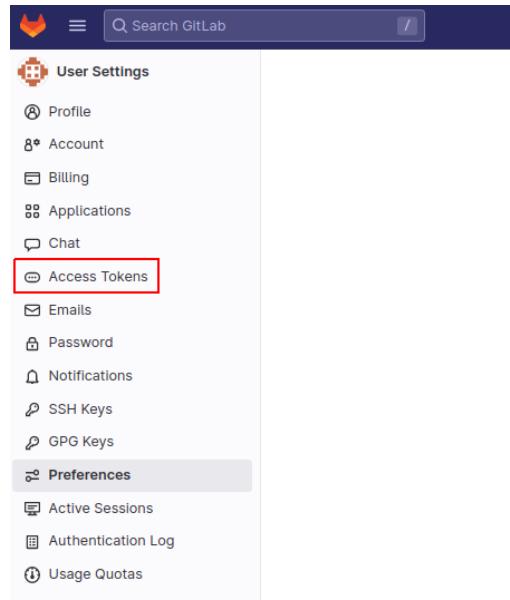
Na página principal do Gitlab, clique no **seu avatar no canto superior direito** e então clique em **Preferences**. Veja a Figura 2.11.3.

Figura 2.11.3



A seguir, à esquerda, clique em **Access Tokens**, como na Figura 2.11.4.

Figura 2.11.4



Dê uma descrição para o seu token, escolha uma data de expiração e marque os escopos **read_repository** e **write_repository**, como na Figura 2.11.5. A seguir, clique em **Create personal access token**.

Figura 2.11.5

Add a personal access token

Enter the name of your application, and we'll return a unique personal access token.

Token name

20231 - Pessoal2 - Maua - VGTI - Aula 01

For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Expiration date

2023-05-11

Select scopes

Scopes set the permission levels granted to the token. [Learn more](#).

api Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.

read_api Grants read access to the API, including all groups and projects, the container registry, and the package registry.

read_user Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public_email, and full_name. Also grants access to read-only API endpoints under /users.

read_repository Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.

write_repository Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

read_registry Grants read-only access to container registry images on private projects.

write_registry Grants write access to container registry images on private projects.

Create personal access token

Na tela seguinte, você deverá ser capaz de ver o seu token. Porém, ele ainda está oculto. Clique no botão destacado pela Figura 2.11.6 para visualizá-lo.

Figura 2.11.6

Your new personal access token has been created.

Search page

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Your new personal access token

Make sure you save it - you won't be able to access it again.

Add a personal access token

Enter the name of your application, and we'll return a unique personal access token.

Token name

For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Você deve copiar seu token agora. Se atualizar ou fechar a página, não poderá vê-lo novamente. Se isso acontecer, você precisará criar outro token.

Uma vez que tenha copiado seu token, volte ao VS Code para montar a URL, incluindo o token. O formato padrão é assim:

<https://oauth:token@gitlab.com/usuario/repositorio.git/>

Veja um exemplo completo, incluindo um token e um repositório válidos.

https://oauth:glpat-xJ8xh7spaiEDQkr3i5Bj@gitlab.com/professorbossini/20231_pessoal_m_aua_vgti Tutorial_git_python.git/

Depois de montar o seu link com o token, adicione-o ao repositório local com

git remote add origin

https://oauth:glpat-xJ8xh7spaiEDQkr3i5Bj@gitlab.com/professorbossini/20231_pessoal_m_aua_vgti Tutorial_git_python.git/

Aproveite e feche a aba temporária que você havia aberto no VS Code apenas para montar o link.

Agora já podemos fazer um push. Para tal, precisamos dizer o destino (o nome associado ao link do repositório remoto) e o objeto que será enviado (a branch main, neste caso). Se desejar, você pode verificar o nome do remote com

git remote

ou

git remote -v

para ver o link também.

E para ver o nome da branch, use

git branch

Para fazer o push, use

git push origin main

Observe que o seu token fica armazenado no seu repositório local e, qualquer pessoa que tiver acesso a ele poderá visualizá-lo. Se desejar, você pode remover o endereço de seu remote, incluindo o token, com

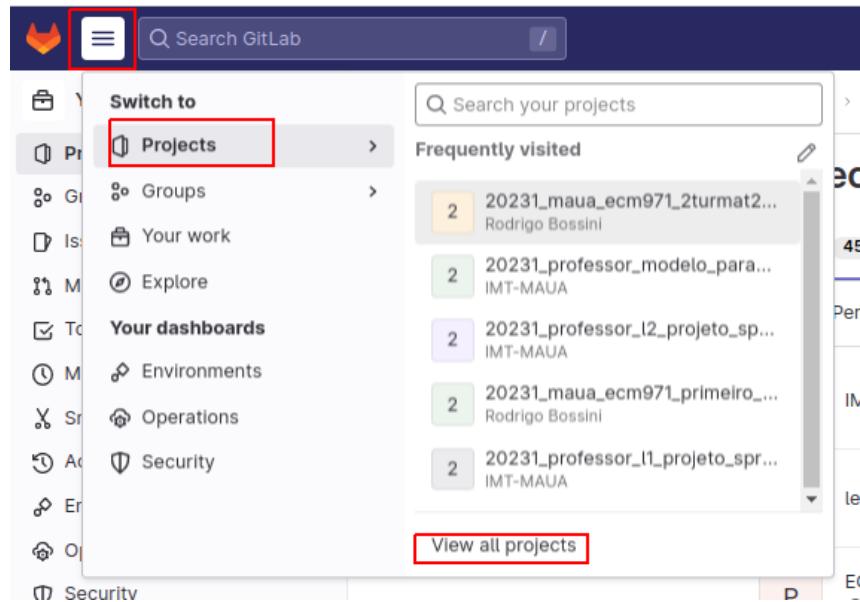
git remote remove origin

Certifique-se de que ele foi removido com

git remote -v

No site do Gitlab, visite a página do seu projeto. Se não conseguir encontrar, use os atalhos destacados na Figura 2.11.7. Vasculhe a lista de projetos a fim de encontrar o seu.

Figura 2.11.7



Observe que os commits estão por lá. No entanto, as tags ainda não. Veja a Figura 2.11.8.

Figura 2.11.8

The screenshot shows the project page for '20231_pessoal_maua_vgti_tutorial_git_python'. At the top, there is a message about GitLab Ultimate benefits. Below the message, the project title '20231_pessoal_maua_vgti_tutorial_git_python' is displayed with a count of 2 branches. A red box highlights the '8 Commits' link. Below the title, there are buttons for 'Star' (0), 'Fork' (0), and 'Clone'. Another red box highlights the '0 Tags' link. The main content area shows a commit history with one commit from 'apostila02(git-python)': 'teste da funcao dividir'. The commit was authored by 'Rodrigo Bossini' 1 hour ago. Below the commit history, there is a navigation bar with links like 'main', 'Find file', 'Web IDE', 'Clone', and buttons for 'Add README', 'Add LICENSE', etc. At the bottom, there is a table showing files with their last commits and update times.

Name	Last commit	Last update
calculadora.py	apostila02(git-python): funcao dividir implementada	1 hour ago
teste_calculadora.py	apostila02(git-python): teste da funcao dividir	1 hour ago

Faça o push da primeira tag com

git push origin v1.0.0

Verifique, no site do Gitlab, se a tag foi mesmo enviada, como na Figura 2.11.9. Não esqueça de atualizar a página no navegador.

Figura 2.11.9

Rodrigo Bossini > 20231_pessoal_maua_vgti_tutorial_git_python

Your project is no longer receiving GitLab Ultimate benefits as of 2022-07-01. As notified in-app previously, public open source projects on the Free tier can apply to the GitLab for Open Source Program to receive GitLab Ultimate benefits. Please refer to the [FAQ](#) for more details.

20231_pessoal_maua_vgti_tutorial_git_python

Project ID: 45084269

-o 8 Commits 1 Branch 1 Tag 0 Bytes Project Storage

apostila02(git-python): teste da funcao dividir
Rodrigo Bossini authored 1 hour ago f53dc9c

main v 20231_pessoal_maua_vgti_tutorial_git_python / + v Find file Web IDE Clone v Add README Add LICENSE Add CHANGELOG Add CONTRIBUTING Enable Auto DevOps Add Kubernetes cluster Set up CI/CD Add Wiki Configure Integrations

Name	Last commit	Last update
calculadora.py	apostila02(git-python): funcao dividir implementada	1 hour ago
teste_calculadora.py	apostila02(git-python): teste da funcao dividir	1 hour ago

Você também pode enviar todas as tags de uma vez com

git push origin --tags

Visite mais uma vez a página de seu repositório remoto e veja se todas as tags estão por lá, como na Figura 2.11.10. Não esqueça de atualizar a página.

Figura 2.11.10

Rodrigo Bossini > 20231_pessoal_maua_vgti_tutorial_git_python

Your project is no longer receiving GitLab Ultimate benefits as of 2022-07-01. As notified in-app previously, public open source projects on the Free tier can apply to the GitLab for Open Source Program to receive GitLab Ultimate benefits. Please refer to the [FAQ](#) for more details.

20231_pessoal_maua_vgti_tutorial_git_python

Project ID: 45084269

-o 8 Commits 1 Branch 4 Tags 0 Bytes Project Storage

apostila02(git-python): teste da funcao dividir
Rodrigo Bossini authored 1 hour ago f53dc9c

main v 20231_pessoal_maua_vgti_tutorial_git_python / + v Find file Web IDE Clone v Add README Add LICENSE Add CHANGELOG Add CONTRIBUTING Enable Auto DevOps Add Kubernetes cluster Set up CI/CD Add Wiki Configure Integrations

Name	Last commit	Last update
calculadora.py	apostila02(git-python): funcao dividir implementada	1 hour ago
teste_calculadora.py	apostila02(git-python): teste da funcao dividir	1 hour ago

[1] KIM, G.; HUMBLE, J.; DEBOIS, P; WILLIS, J.; FORSGREN, N. **The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations**. 2. ed. IT Revolution Press, 2021.

[2] CHACON, S; STRAUB, B. **Pro Git**. 2. ed. Apress, 2014.