Aula: Git e GitHub

- Primeiros passos com o Git
  - O que é controle de versão?
  - **O que é Git?**
  - Instalando o Git
  - Configurando o Git
  - Criando repositório Git
  - Estados de arquivos no Git
  - O primeiro *commit:* `git commit`
    - Exercício: criando um repositório do Git
  - Primeiras mudanças: `git diff`
  - Voltando no tempo com `git checkout`
  - Ramificações (*branches*): `git branch`
  - Resolvendo conflitos
    - Sem fast-forward, sem conflitos (arquivos diferentes)
    - Mudanças com conflitos
    - Exercício: Git branching
  - Resumo do Git
- GitHub e Repositórios Remotos
  - Repositórios locais vs remotos
  - Criando um repositório no GitHub
  - Clonando um repositório do GitHub e fazendo `push`
    - Exercício: clonando um repositório e fazendo `push`
  - Upload de repositório local para o GitHub
  - Sincronizando o trabalho com `pull` e `push`
- Colaboração
  - Pull Requests
    - Exercício: criando um Pull Request
  - Issues
    - Exercício: criando uma Issue e um Pull Request associado
  - Fork
    - Mantendo o *fork* atualizado
    - Exercício: fazendo um fork do repositório do curso
  - Permissões
    - Exercício: adicionando colaboradores
  - *Gists*

Contribuindo com projetos *Open-Source*
    Minhas contribuições para o Flask:
Dicas
    Git Config
    VSCode
Referências

Primeiros passos com o Git

O que é controle de versão?

> 💡 Como **você faria** o controle de versão de um arquivo? E de vários arquivos? Quais os problemas?

O que é Git?

É um **programa** para gerenciar versões de projetos (focado em *código-fonte*, mas também pode manter outros arquivos – imagens, binários, etc.).

Instalando o Git

https://git-scm.com/downloads

> 💡 Git é um **programa** que pode ser utilizado via *linha de comando* ou via *GUI* (Interface Gráfica)

Após instalar o Git, você deve ser capaz de executar o seguinte comando no seu terminal (ou no Git Bash, caso instalado):

```
$ git --version
git version 2.32.0 (Apple Git-132)
```

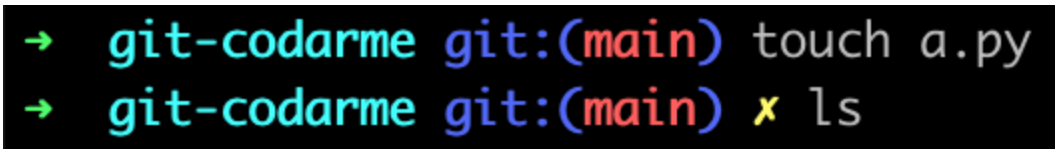
Configurando o Git

```
$ git config --global user.name "Seu Nome"
$ git config --global user.email seuemail@example.com
# Para configurar o editor de texto a ser utilizado com o Git
$ git config --global core.editor "code --wait"
```

Criando repositório Git

```
$ mkdir git-cordame
$ cd git-codarme
$ git init
$ ls -A
.git  # Pasta .git criada, indicando que a pasta raíz é um repositório git
$ git status
On branch main...
```

💡 Dependendo do programa de linha de comando que você usa, podem existir algumas dicas visuais, como por exemplo, em qual *branch* você está, ou indicando se existem arquivos a serem adicionados ou não.



Exemplo: branch “main", arquivo não rastreado no *working directory*

Estados de arquivos no Git

- *Modified*. Arquivos que fazem parte da pasta do projeto versionado. Ao ambiente que contém esses arquivos chamamos de ***working directory***.
- *Staged:* Arquivos/alterações marcadas para serem incluídas no próximo *commit* (persistidas no banco de dados do Git). Ambiente: **staging area**.
- *Commited:* Alterações efetivamente versionadas e persistidas no banco de dados do Git. Ambiente: o próprio banco de dados (.git).

O primeiro *commit:* `git commit`

```
$ touch README.md  # cria arquivo REAMDE.md, adicionado ao working directory

$ git status
...
Untracked files: README.md  # Não-rastreado pelo git ainda (untracked)

$ git add README.md  # Adiciona à staging area

$ git status  # Exibe status do repositório
...
Changes to be commited:
    new file: README.md

$ git commit  # Vai abrir um editor de texto (possivelmente vim)
# Após salvar, mudanças são commitadas (persistidas) no git database
# Outra opção: git commit -m "Mensagem"

$ git log  # Exibe o log dos últimos commits
commit 69def055fcbee6d7520810a7884db6fccada48e8 (HEAD -> main)
Author: John Doe <john@doe.com>
Date:   Mon Mar 7 11:04:18 2022 -0300

    Mensagem aqui!
```

Exercício: criando um repositório do Git

Crie uma pasta e inicialize um repositório do Git dentro dela.

Adicione um arquivo README.md e escreva alguma coisa nesse arquivo.

Faça um *commit* contendo a adição desse arquivo.

Visualize o *status* do repositório.

Visualize o *log* do repositório.

Primeiras mudanças: `git diff`

```
# Adicionar algum texto ao arquivo README.md
$ echo "# Aula de Git e GitHub" >> README.md

$ git status
...
Changes not staged for commit: ...

$ git diff
diff --git a/README.md b/README.md
index e69de29..8032b55 100644
```

```
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+# Aula de Git e GitHub

$ git add README.md  # Ou "git add ." para adicionar todas mudanças no diretório
$ git status  # Verificar mudanças staged
...

$ git commit -m "Adiciona título"

$ git log
commit 75ab4cb9c0c4c20f708e412e7f57bba3d2b532d6 (HEAD -> main)
    ...
    Adiciona título

commit 69def055fcbee6d7520810a7884db6fccada48e8
    ...
    Arquivo inicial
```

Voltando no tempo com `git checkout`

```
$ git checkout 69def055fcbee6d7520810a7884db6fccada48e8
...
HEAD is now at 69def05 Arquivo inicial
# "main" continua apontando para o último commit

$ git status
...
HEAD deatched at 69def05

$ git checkout main  # move HEAD para main novamente
```

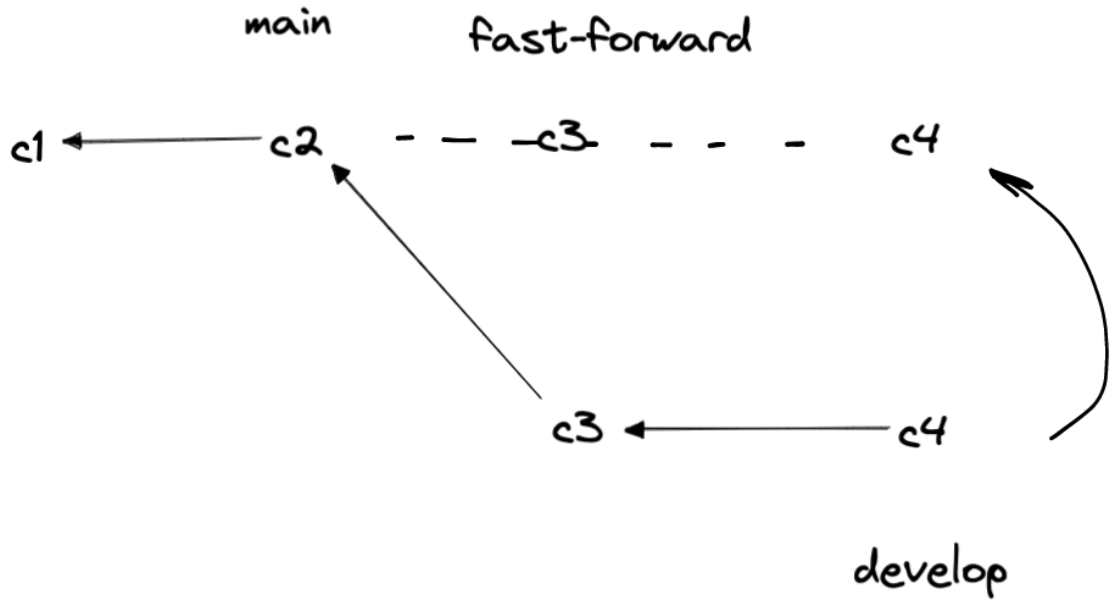
Ramificações (*branches*): `git branch`

```
$ git branch develop  # Cria uma nova branch "develop" a partir do commit atual
$ git branch
develop
*main  # atual marcada com *
```

Normalmente utilizamos *branches* para gerenciar um **trabalho em progresso**. Quando o trabalho está pronto, juntamos às mudanças na *branch principal* (*main*).

```
$ git checkout develop  # Muda o HEAD para branch develop
$ touch app.py  # Cria novo arquivo
$ echo "## Nova seção no arquivo README.md" >> README.md
$ git add app.py README.md  # Adiciona 2 arquivos
$ git commit
$ git log  # Observar o log
$ git checkout main  # Volta para branch main
$ git log  # Commits em develop NÃO inclusos
$ git merge develop  # Junta commits develop => main
```

💡 *Fast-forward* é quando o *git* consegue resolver automaticamente o *histórico* do repositório.

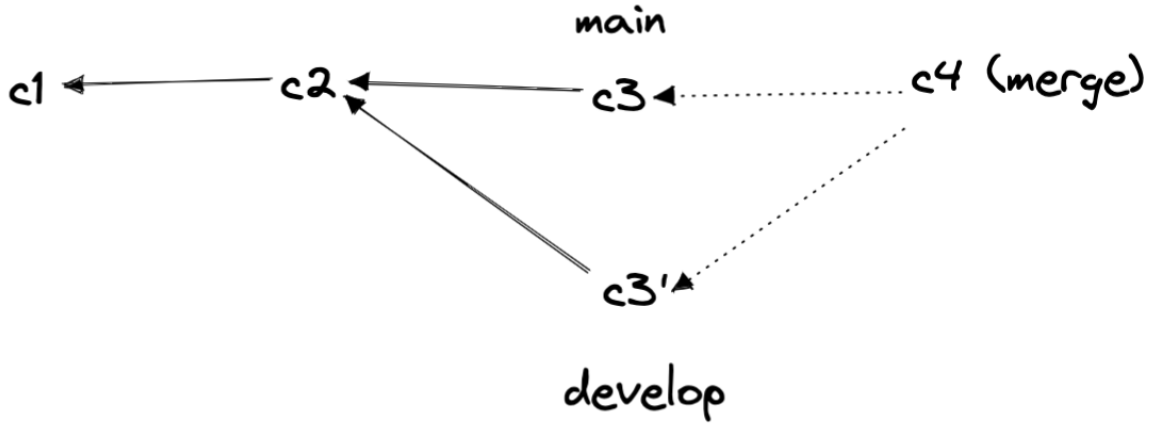


Resolvendo conflitos

Poderíamos ter tido mudanças em *main* antes de tentar juntar as mudanças de *develop*.

Sem fast-forward, sem conflitos (arquivos diferentes)

```
$ git checkout develop
$ touch outro_arquivo.py
$ echo "print(123)" >> outro_arquivo.py
... add commit ...
$ git checkout main
$ echo "Mais coisas" >> README.md  # Alterado apenas em main
$ git merge develop
... Necessário gerar um commit de merge
$ git log --graph
# Verificar mudanças que vieram de develop, de main, e merge commit
```



▼ `git log —-graph` (**merge**)

```
*   commit a83eb9368f0c2e8d614ce2027033d1f1fa70a91b (HEAD -> main)
|\  Merge: 419e0b5 bad763d
| | Author: Gabriel Saldanha <gabrielcrsaldanha@gmail.com>
| | Date:   Mon Mar 7 11:56:08 2022 -0300
| |
| |     Merge branch 'develop'
| |
| * commit bad763d679b0120df138634ea0ba630b53c96b8e (develop)
| | Author: Gabriel Saldanha <gabrielcrsaldanha@gmail.com>
| | Date:   Mon Mar 7 11:55:34 2022 -0300
| |
| |     Mais mudanças em README
| |
* | commit 419e0b51febb32f2e11f27d6015977ea7968a2b5
|/  Author: Gabriel Saldanha <gabrielcrsaldanha@gmail.com>
```

```
|   Date:   Mon Mar 7 11:55:59 2022 -0300
|
|       Adiciona outro_arquivo.py
|
* commit 72751d7ba339214e850f9ee0ffa4cda642845c7d
| Author: Gabriel Saldanha <gabrielcrsaldanha@gmail.com>
| Date:   Mon Mar 7 11:45:40 2022 -0300
|
|     Adiciona função soma
|
* commit d3e231ff0c28712386f6ab66a548edfcffd2f50c
| Author: Gabriel Saldanha <gabrielcrsaldanha@gmail.com>
| Date:   Mon Mar 7 11:32:03 2022 -0300
|
|     Adiciona título
|
* commit 69def055fcbee6d7520810a7884db6fccada48e8
  Author: Gabriel Saldanha <gabrielcrsaldanha@gmail.com>
  Date:   Mon Mar 7 11:04:18 2022 -0300

      Arquivo inicial
```

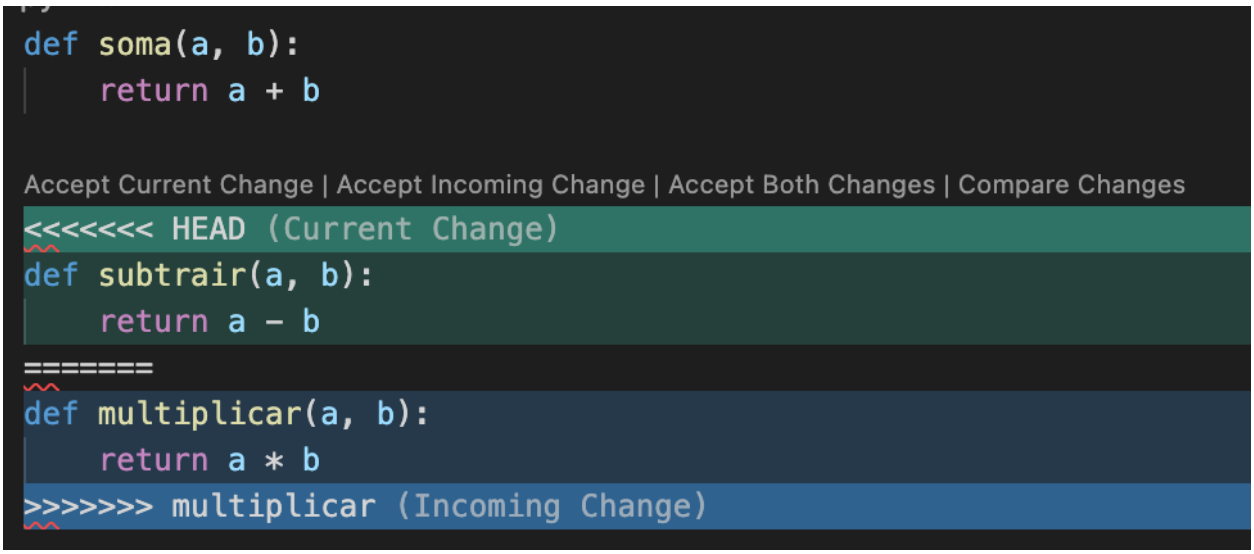
Mudanças com conflitos

E se tivéssemos alterado o mesmo arquivo?

- Adicionar método subtrair em app.py
- Adicionar método multiplicar em app.py

```
$ git checkout -b subtrair  # checkout -b <branch>: cria e vai para a branch
... faz modificações, add, commit

# outro desenvolvedor
$ git checkout -b multiplicar
... faz modificações, add, commit

$ git checkout main
$ git merge subtrair
... fast-forward

$ git merge multiplicar
Auto-merging app.py
CONFLICT (content): Merge conflict in app.py
Automatic merge failed; fix conflicts and then commit the result.

$ git status
... both modified: app.py
```

> 💡 Precisamos corrigir os conflitos! **current**: branch atual, **incoming**: branch sendo *mergeada*.



- Agora é um bom momento para utilizar o *VSCode* para resolver os conflitos 😉

Após resolver os conflitos:

```
$ git add .
$ git commit
$ git log --graph
```

Exercício: Git branching

Crie um arquivo `app.py`, adicione este arquivo à *staging area* e faça um *commit* na *branch principal.*

Crie 3 branches (`git branch <nome-branch>`) chamadas: *fast-forward*, *merge-commit, merge-conflict*.

Crie commits nessas branches de modo que:

1. Ao fazer o *merge de fast-forward → main*, não haja conflitos e o histórico seja gerado pela estratégia de *fast-forward*.
2. Ao fazer *merge* de *merge-commit → main*, não haja conflitos e seja criado um *merge commit*.

3. Ao fazer *merge* de *merge-conflict* → *main*, tenha conflitos e seja criado um *merge commit* com os conflitos resolvidos.

Resumo do Git

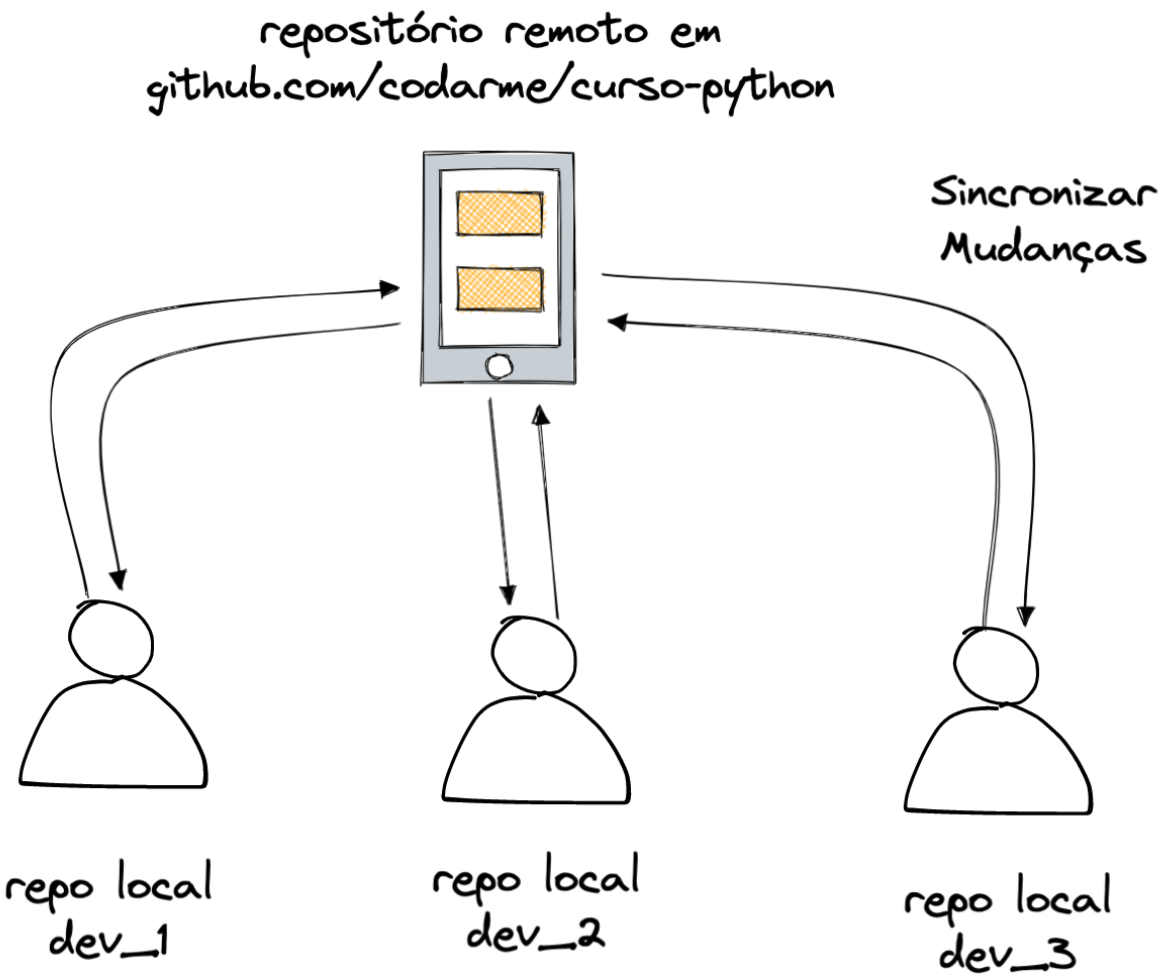
- Adicionar arquivos na *staging area*
- Fazer *commit* no banco de dados (git)
- Criar *branches* para fazer alterações sem modificar a *branch principal* (main)
- Fazer junção (*merge*) de *branches*
  - Fast-forward
  - Merge commit
  - Conflitos

> 💡 Se você entender bem esses conceitos, o resto é prática e procurar no Google. "Como removo arquivos adicionados à staging area?", "Como desfazer commits?", etc.

GitHub e Repositórios Remotos

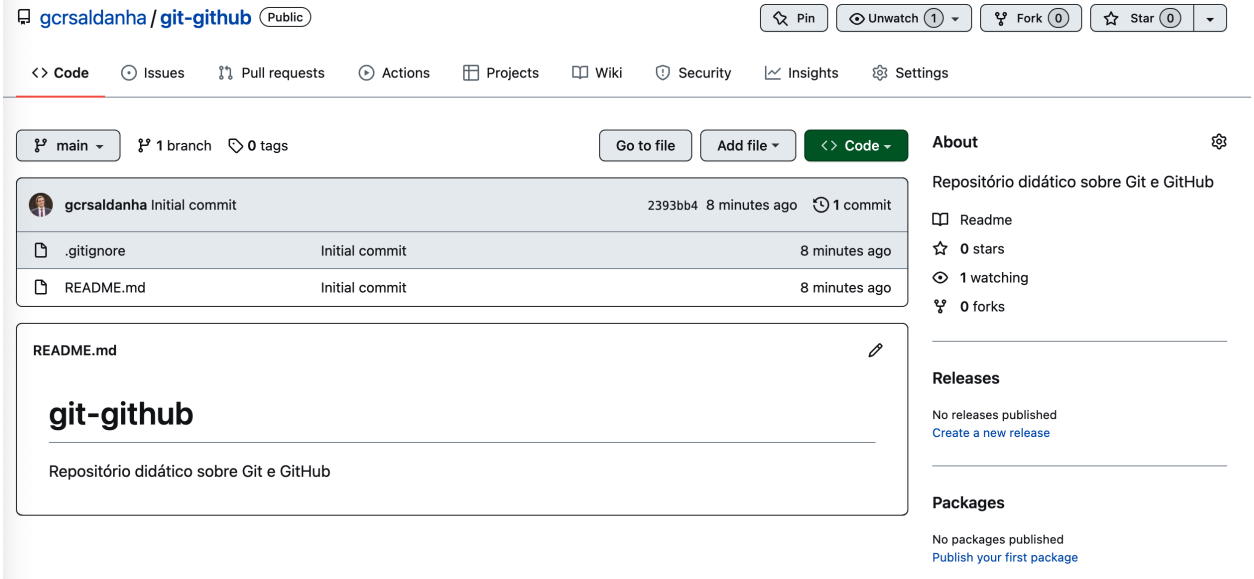
Repositórios locais vs remotos

- O *Git* se diferencia de alguns sistemas de controle de versão pelo fato de permitir desenvolvimento distribuído – repositórios locais e repositório remoto.



Criando um repositório no GitHub

1. Acessar https://github.com/ e criar uma nova conta com um usuário e endereço de e-mail.
2. Selecionar "novo repositório" (+ no canto direito superior)
   a. Add README
   b. Add `.gitignore` para Python



Clonando um repositório do GitHub e fazendo `push`

Normalmente nós vamos fazer o *download* desse repositório para o nosso computador para trabalhar nele, fazer nossas mudanças, e depois *sincronizá-las* (local ↔ remoto).

Chamamos de *clonar* o ato de criar uma cópia local de um repositório remoto.

1. Faça um `clone` repositório no seu computador
    a. HTTPS
    b. SSH: precisamos criar um **par de chaves** para autenticação.
        i. https://docs.github.com/pt/authentication/connecting-to-github-with-ssh
        ii. https://github.com/settings/keys
2. Entre no repositório local e observe os arquivos.
3. Digite `git remote -v`
    a. Exibe o *destino* do repositório remoto para fazer sincronização (*fetch*/*push*).

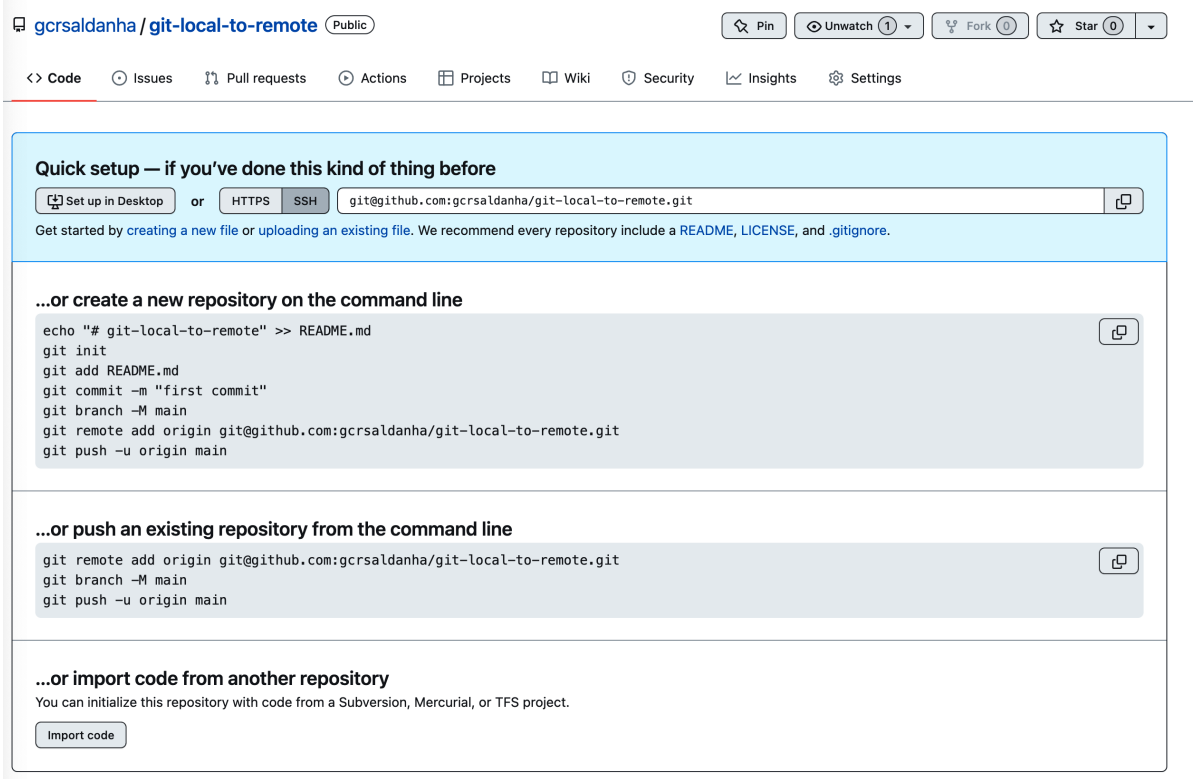
Exercício: clonando um repositório e fazendo `push`

Crie um repositório no GitHub, faça o *clone* dele localmente, faça alguns *commits*. Por fim, *sincronize suas mudanças* com o repositório remoto utilizando o comando `git push`.

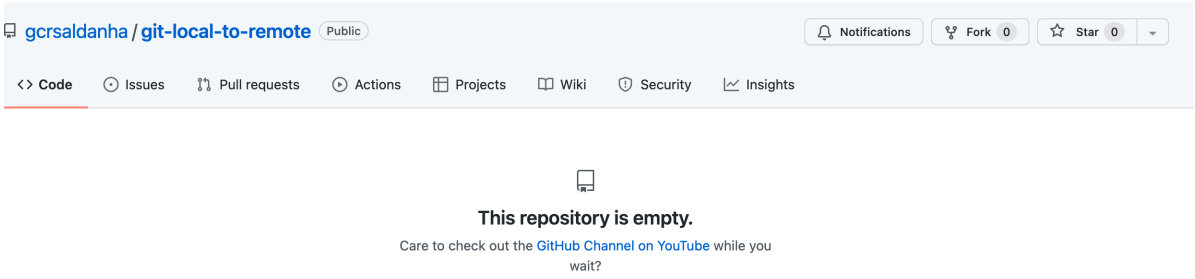
Upload de repositório local para o GitHub

Também podemos fazer o *upload* de um repositório local para um repositório remoto no GitHub.

1. Acesse seu repositório local
2. Digite `git remote` e verifique que **ele não possui um remote**
3. Crie um novo repositório no GitHub
    a. **Não marque as opções de criar arquivos README, gitignore, etc.**
    b. Assim o GitHub entende que você quer *importar* um repositório existente e vai exibir a tela abaixo:



Caso você **não seja o dono do repositório:**



4. Agora basta seguir as instruções

```
# Adicionar um remote (origin é o nome padrão)
git remote add origin git@github.com:gcrsaldanha/git-local-to-remote.git
git branch -M main  # Renomeia branch atual para main
git push  # erro!
fatal: The current branch main has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin main

git push --set-upstream origin main  # sincroniza local:main com origin:main
# lembrando que origin é o apelido do remote que adicionamos
git remote -v  # para visualizar o remote
```

Sincronizando o trabalho com `pull` e `push`

Utilizamos o comando `push` para enviar mudanças locais para o repositório remoto.

Utilizamos o comando `pull` para *trazer* mudanças do repositório remoto para o repositório local.

> 💡 Às vezes podemos ter conflitos, merge-commits, fast-forward assim como tivemos quando estávamos trabalhando com múltiplas branches. Isso porque estamos tentando mesclar mudanças de dois lugares diferentes: `local:branch` e `remote:branch`. Sempre leia o erro que aparecer no *git console* e as sugestões para resolver o problema.

Colaboração

Pull Requests

O GitHub criou o conceito de *Pull Request*, que é basicamente uma proposta de alterações que um desenvolvedor envia para um determinado projeto.

Exemplo de Pull Request adicionando material de apoio no repositório do curso de Python: https://github.com/CodarMe/curso-python/pull/2

A vantagem dessa abordagem é que fica mais fácil de gerenciar e permitir mudanças feitas no código, além de podermos definir regras como *aprovadores necessários*.

Também fica muito mais fácil de receber *feedback* de outras pessoas.

> 💡 A partir de agora, sempre que você quiser feedback sobre alguma mudança de código, crie um Pull Request com a mudança e envie o link do Pull Request no Discord!

P.S.: Apenas colaboradores podem ser *revisores* de Pull Requests.

P.S.2: Você não pode ser o *revisor* de seu próprio Pull Request.

Exercício: criando um Pull Request

Crie uma nova branch no seu repositório local e faça pelo menos um commit com alguma mudança.

Faça o `push` da branch com o novo commit para o repositório remoto.

Acesse o GitHub e crie um novo *Pull Request* da sua branch recém-criada para a branch principal*:* https://github.com/gcrsaldanha/git-github/pulls

Envie o link para o Pull Request no chat.

Issues

Issues representam "problemas" que alguém encontrou em um repositório. Normalmente, qualquer usuário do GitHub pode criar *issues* em repositórios públicos. O interessante é que ao criar um Pull Request, podemos associar o *Pull Request à uma Issue* criada.

Exercício: criando uma Issue e um Pull Request associado

1. Crie uma nova issue no repositório de exemplo
2. Crie um novo Pull Request (a partir de uma nova branch)
3. Adicione na descrição do Pull Request: "Resolves #2" (onde 2 é o número da issue)
    a. Isso faz com que esse Pull Request automaticamente finalize a issue #2 ao ser *mergeado*.
4. Faça o merge do PR e verifique se a Issue está como "closed".

Fork

Quando *clonamos* um repositório, simplesmente criamos a versão *local* daquele repositório, mantendo todas as permissões e configurações de acordo com o que foi definido pelo *dono do repositório*. Por exemplo: eu posso clonar um repositório mas não necessariamente posso fazer um *push* diretamente para uma branch desse repositório – pois não tenho *acesso de escrita à ele*.

A operação *fork* do GitHub cria uma *nova cópia completa* do repositório original e define o usuário que fez o *fork* como o *dono daquele repositório.* Ou seja, a partir de agora, esse *fork* pode existir independentemente do repositório original.

Normalmente para projetos *open-source*, você precisa criar um *fork* do projeto original, fazer suas alterações (escrita) nesse novo repositório, e depois criar um *Pull Request* partindo de `fork:branch` para `original:branch`.

Mantendo o *fork* atualizado

Para manter o *fork* atualizado com o repositório original (também chamado de `upstream`), você precisa adicionar o repositório *upstream* como um *remote*:

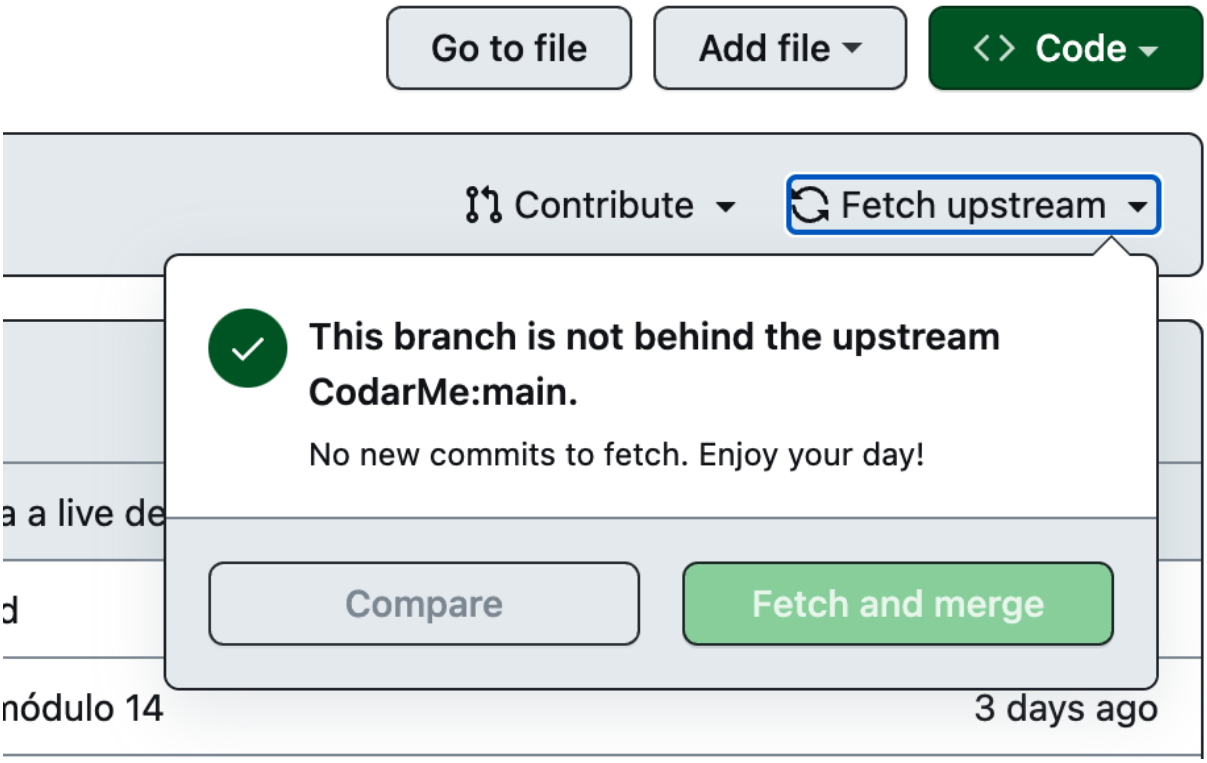
```
$ git remote add upstream https://github.com/CodarMe/curso-python.git
$ git remote -v
origin  git@github.com:gcrsaldanha/curso-python.git (fetch)  # Meu repo
origin  git@github.com:gcrsaldanha/curso-python.git (push)
upstream  https://github.com/CodarMe/curso-python.git (fetch)  # Repo da codarme
upstream  https://github.com/CodarMe/curso-python.git (push)
```

A partir daí, você pode fazer o `pull` das mudanças de `upstream/main` e mergear em `origin/main`

```
$ git pull upstream main  # Faz o pull (fetch e merge) das mudanças
$ git push origin main
```

💡 Mais sobre como manter o *fork* sincronizado: https://docs.github.com/pt/pull-requests/collaborating-with-pull-requests/working-with-forks/syncing-a-fork

Uma opção mais simples é utilizando o "fetch upstream" do próprio GitHub:

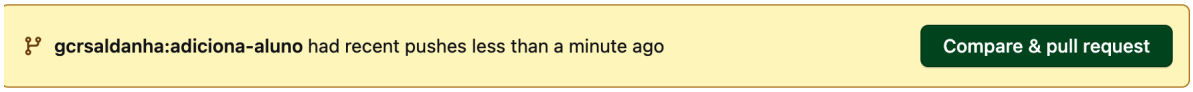


Exercício: fazendo um fork do repositório do curso

1. Acesse https://github.com/codarme/curso-python e faça um *fork*.
2. Faça clone do repositório recém-criado para sua máquina local.
3. Crie uma nova branch.
4. Adicione seu nome e usuário do GitHub no arquivo `lives/alunos.md`
5. Abra um Pull Request a partir da sua branch para o repositório principal (upstream).

a. Repositório principal: https://github.com/CodarMe/curso-python

b. Exemplo de Pull Request: https://github.com/CodarMe/curso-python/pull/3



6. Envie o link no chat ou no canal de exercícios do Discord 😄

> 💡 Resumindo: se você tem acesso de escrita ao repositório, basta *clonar* e fazer o *push* em uma branch com suas mudanças. Caso não possua acesso de escrita, é necessário fazer o *fork, criar sua branch* e abrir um Pull Request a partir do `seu-repo:sua-branch` para `upstream-repo:upstream-branch`.

Permissões

Para gerenciar as permissões e colaboradores de um determinado repositório, acesse: https://github.com/gcrsaldanha/git-local-to-remote/settings/access

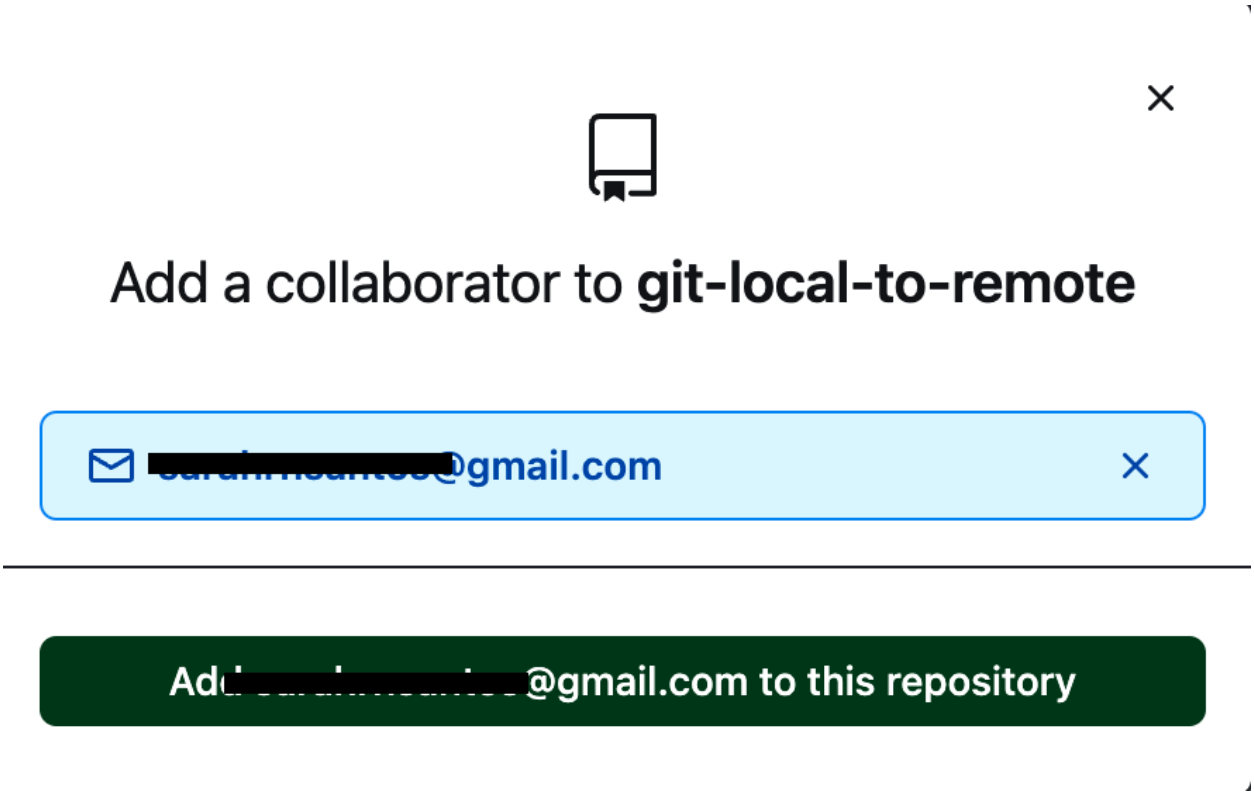
> 💡 Repositórios públicos podem ser **visualizados por qualquer pessoa da internet** mesmo que não tenha uma conta no GitHub!!! **Muito cuidado** com valores de tokens, chaves de acesso, variáveis de ambiente, etc.

Exercício: adicionando colaboradores

> 💡 Durante a aula, o instrutor vai selecionar algum aluno para ser o colaborador do repositório.

Adicione algum aluno como colaborador do seu repositório.

Tente adicioná-lo como *revisor* de um Pull Request.



Gists

Gist é o termo do GitHub para um trecho de código que pode ser facilmente compartilhado. Tipo um *snippet.*

Para criar um novo gist basta clicar no + ou acessar gist.github.com e criar a partir de lá.

Todo gist possui um link único par ser acessado e outros usuários podem copiar (fork) ou comentar no gist. Um gist pode possuir múltiplos arquivos de diferentes extensões.

Você pode criar gists públicos ou privados.

- Públicos: podem ser vistos por qualquer pessoa que visitar seu perfil no GitHub e são indexados por mecanismos de pesquisa.
- Privados: não são listados no seu perfil ou indexados por mecanismos de pesquisa. Acessados somente através do link.

> 💡 Gists não possuem nenhum tipo de autenticação/autorização e podem ser acessados por qualquer pessoa com o link!

Contribuindo com projetos *Open-Source*

Fluxo comum:

- *fork*
- criar uma branch no *seu fork* (seguindo os *guidelines do repositório principal*)
- abrir um *pull request* a partir **do seu fork:branch** para o repo principal:branch
- *rodar os testes* (e escrever testes quando necessário)
- Aguardar aprovações 😄

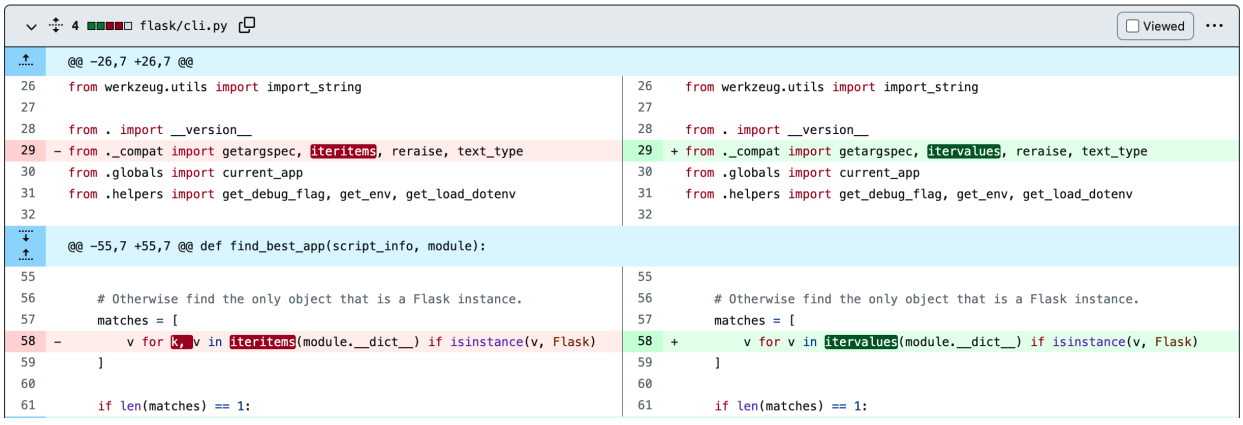
> 💡 Não é tão difícil quanto parece fazer contribuições quanto parece!

Minhas contribuições para o Flask:

🔥 **Remove code supposed to be removed at v1.0**

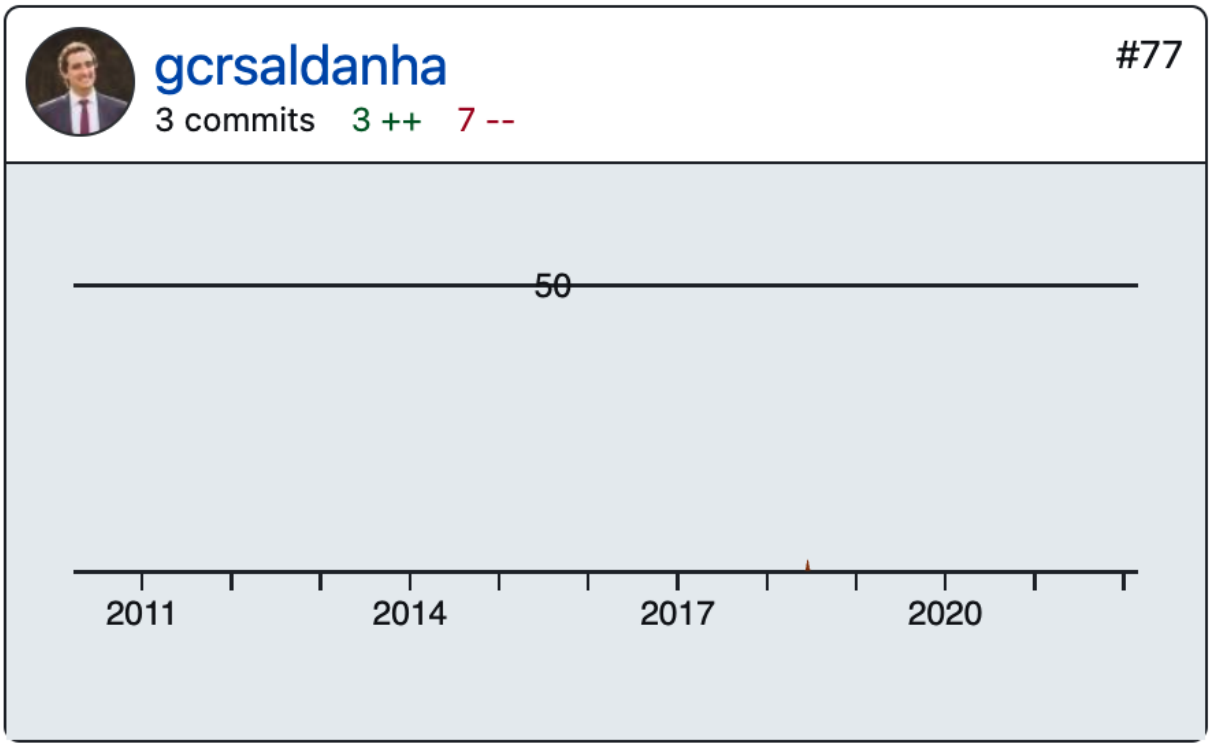
🎨 **Use set literal instead of set method in file uploading example**

🎨 Use `itervalues` instead of `iteritems` when key is not necessary



🔥 **Refactor `__init__.py` imports (REJEITADO 😂)**

Mesmo com essas contribuições simples (2018), eu sou o *contributor #77:* https://github.com/pallets/flask/graphs/contributors



Dicas

Git Config

Você pode configurar atalhos para o `git` utilizando `git config --global alias.<alias> <git command>`

Abrir `~/.gitconfig`

```
[user]
    name = Gabriel Saldanha
    email = gabriel@codar.me
[core]
    editor = vim
[alias]
    co = checkout
    br = branch
    ci = commit
    st = status
    unstage = reset HEAD --
    last = log -1 HEAD
```

VSCode

Você também pode utilizar o VSCode para realizar a maioria dos comandos git utilizando a GUI. Entretanto, eu recomendo que você se acostume a utilizar a linha de comando primeiro e utilize a GUI quando necessário para coisas como resolver conflitos

Referências

Gia rápido de comandos e fluxo git: https://www.instagram.com/p/B8du1S-FPpe/

Tutorial Git da Atlassian: https://www.atlassian.com/br/git/tutorials

GitHub's Cheatsheet: https://training.github.com/downloads/pt_BR/github-git-cheat-sheet.pdf

SSH: https://docs.github.com/pt/authentication/connecting-to-github-with-ssh

Acessando chaves GitHub: https://github.com/settings/keys

GitHub Quickstart: https://docs.github.com/pt/get-started/quickstart/hello-world