

PREPARAÇÃO PARA HTML E CSS

MODULO 01

VETORES (ARRAYS)

Os dados visto até agora só podiam ser guardados um único valor por variável. Mesmo sendo várias operações ou várias concatenações, ao final teriamos apenas um único valor.

```
operacao = (5 + 3) * (9 - 4)
# a variável 'operacao' tem um dado só, o inteiro 40

string = "Eu " + "gosto " + "de " + "panquecas"
# a variável 'string' tem um único dado, a string "Eu gosto de panquecas"
```

Para guardar mais de um valor em uma variável utilizamos os **vetores** ou **Arrays**

Declarar um Array em Python

```
lista_numeros = []
Declaração de um vetor vazio

lista_numeros = [3, 19, 54, -2]
Declaração de um vetor com dados

lista_floats = [0.3, 3.9, 89.15, 123.45]
lista_strings = ["Eu", "gosto", "de", "panquecas"]
lista_booleanos = [False, True, True, False]

lista_listas = [[1, 2, 3], ['a', 'b', 'c']]
ARRAYS DE ARRAYS
```

JavaScript e Python **permitem Arrays de tipo de dados diferentes**. Outras linguagens não permitem.

```
lista_mista = [10, 'dez', True]
```

Imprimir os valores de um Array

```
lista_frutas = ['maçã', 'banana', 'pera']
print(lista_frutas)
```

```
# Imprimirá: ['maçã', 'banana', 'pera']
```

Para acessar os itens individuais de um Array é utilizado o valores numericos dos índices.

```
lista_frutas = ['maçã', 'banana', 'pera']

# 'maçã' tem o índice 0
# 'banana' tem o índice 1
# 'pera' tem o índice 2

# Imprimindo um valor específico em um Array

lista_frutas = ['maçã', 'banana', 'pera']
print(lista_frutas[0])

# Chamará o primeiro valor e Imprimirá: 'maçã'
```

Acessar um item do Array não altera ou exclui seu conteúdo.

FUNÇÃO len()

Para sabermos o tamanho de um Array utilizamos a função **len()**, que significa length(cumprimento)

```
lista_frutas = ['maçã', 'banana', 'pera']
quantidade_frutas = len(lista_frutas)

print(quantidade_frutas)

# Imprimirá o número 3, pois lista_frutas tem três elementos
```

Também é possível passar a função **len()** como argumento da função **print()**, sem guardar previamente numa variável.

Elemento != índices No exemplo acima há 3 elementos e que seu índice é 2. Uma vez que se inicia em zero.

PERCORRER ARRAYS (Python)

```
#uma forma de percorrer Arrays, porém pouco efetivo quando se tem milhares de dados.

lista_frutas = ['maçã', 'banana', 'pera']

print(lista_frutas[0])
print(lista_frutas[1])
```

```
print(lista_frutas[2])

# Imprimirá:
# maçã
# banana
# pera
```

Utilizamos uma estrutura de repetição percorrer todos os dados de um Array

```
lista_frutas = ['maçã', 'banana', 'pera']

for i in range(3):
    print(lista_frutas[i])

# Imprimirá:
# maçã
# banana
# pera
```

Lembre-se:

que a estrutura For, utilizando a Função Range() ao declararmos o laço e não passamos mais argumentos, a função range() atribui um valor padrão para o valor inicial e o incremento do contador. Estes valores por padrão são 0 e +1, respectivamente.

Podemos percorrer um Array utilizando a função len(). Independentemente da quantidade de elementos que um vetor tiver, conseguiremos percorrer todos eles usando: uma estrutura de repetição, e a função len()

```
lista_num = [2, 45, 65, 78, 126, 987, 457, 345, 679, 107, 2345, 452, 3, 34, 560]

for i in range(len(lista_num)):
    print(lista_num[i])
```

ALTERAR DADOS DO ARRAY USANDO ÍNDICES

```
lista_frutas = ['maçã', 'banana', 'pera']
# Não gostamos de maçãs

lista_frutas[0] = 'melancia' # atribuindo novo valor ao **índice 0**
print(lista_frutas)
# Imprimirá: ['melancia', 'banana', 'pera']

# Abaixo alteremos mais de um valor do Array
lista_frutas[1], lista_frutas[2] = 'morango', 'abacaxi'
print(lista_frutas)
# Imprimirá: ['melancia', 'morango', 'abacaxi']
```

```
#Podemos atribuir um valor de um índice a outro
lista_frutas[1] = lista_frutas[0]
print(lista_frutas)
# Imprimirá: ['melancia', 'melancia', 'abacaxi']
```

LEMBRESE:

Atribuição =

Quem fica do lado esquerdo é que recebe a informação, o da direita é o dado que queremos salvar.

ADICIONAR E REMOVER ELEMENTOS NO ARRAY

- Adicionar elementos **ao final** em um Array Função **append()**
- Remover o **último elemento** de um Array Função **pop()**

```
lista_frutas = ['melancia', 'morango', 'abacaxi']
print(lista_frutas)
# Imprimirá: ['melancia', 'morango', 'abacaxi']

lista_frutas.append('kiwi')
print(lista_frutas)
# Imprimirá: ['melancia', 'morango', 'abacaxi', 'kiwi']
```

Leitura Complementar

- Official documentation for Python 3.10.8. Disponível em: <https://docs.python.org/pt-br/3.7/library/array.html>. Acesso em: 12 de out. 2022
- KITAMURA, Celso. O Que É Array Em Python? Disponível em: https://www.youtube.com/watch?v=1QIG_XiED0g. Acesso em 12 de out. 2022
- SHERRILL, Derrick. Python Algorithm Series. Disponível em: https://www.youtube.com/playlist?list=PLc_Ps3DdrcTsizjAG5uMhpoDfhDmxpOzv Acesso em 12 de out. 2022
- FelixTechTips. Merge Sort In Python Explained (With Example And Code). Disponível em: <https://www.youtube.com/watch?v=cVZMah9kEjI>. Acesso em 12 de out. 2022

MODULO 02

GIT

Git

Conceito: São ferramentas extremamente úteis, mas por mais que sejam semelhantes são totalmente diferentes. Lembre-se Git e Github são coisa muito diferentes.

Terminal Git Bash

Graphic User Interface-**GUI**: Interface gráfica do Usuário (explorador de arquivos, navegador).

Terminal: São conhecidos como **consoles, linhas de comando, shells** de uma forma ampla, pois cada um tem suas especificidades.

O que é um terminal?

Os terminais nos permitem interagir com nosso computador **usando apenas o teclado**. Nós digitamos algum comando, e o terminal executa a tarefa relacionada com esse comando, por exemplo, abrir uma pasta, criar um arquivo, mudar o nome do arquivo, abrir o arquivo com um determinado programa, etc.

Cada sistema operacional vem com um ou mais terminais instalados por padrão.

O GitBash é um **aplicativo que emula a experiência de linha de comando** pensada especificamente para o uso do Git. Da mesma forma que com o PowerShell, podemos apertar a tecla de Windows, escrever "git bash", e apertar a tecla Enter para abrir o Git Bash.

Comandos de navegação

Comando	Significado	Usado para
<code>pwd</code>	<i>Print Working Directory</i>	"Imprimir" a rota ou caminho da pasta atual
<code>cd</code>	<i>Change Directory</i>	- Acessar uma pasta, ou "descer um nível" - Voltar à pasta de início por padrão
<code>cd ..</code>		Voltar uma pasta, ou "subir um nível"
<code>ls</code>	<i>List Files</i>	Mostrar todas as pastas e arquivos na pasta atual
<code>mkdir "NOME"</code>	<i>Make Directory</i>	Criar uma pasta
<code>rmdir "NOME"</code>	<i>Remove Directory</i>	Remover, ou deletar, uma pasta
<code>code.</code>		Abrir a pasta atual no VSCode

Comando	Significado	Usado para
<code>pwd</code>	Print Working Directory	"imprimir" a rota ou caminho da pasta atual
<code>cd</code>	Change Directory	1. Acessar uma pasta ou "descer um nível" 2. Voltar à pasta de início por padrão
<code>cd ..</code>		Voltar uma pasta, ou "subir um nível". Atenção, há espaço entre o cd e o dois pontos
<code>ls</code>	List Files	Mostrar todas as pastas e arquivos na pasta atual
<code>mkdir "NOME"</code>	Make Directory	Criar uma pasta
<code>RMDIR "NOME"</code>	Remove Directory	Remover, ou deletar, uma pasta

Comando	Significado	Usado para
<code>code</code>		Abrir a pasta atual no VSCode

Acessar pastas

`ls`, listará os arquivos e pasta do local atual.

`cd OneDrive`, Acessa a pasta OneDrive dentro da pasta que foi listada.

o final da linha em amarelo nos mostra onde estamos, no exemplo `~/OneDrive`

```

USER@ControleInterno-PMGP01 MINGW64 ~
$ cd OneDrive/

USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive
$ ls

USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive
$ ls
'CONTRATO DE CESSÃO E TRANSFERÊNCIA DE DIREITOS DE QUOTAS PARTES DA COOPERATIVA
MISTA DOS GARIMPEIROS DA CUTIA.docx'
'Cofre Pessoal.lnk'*
'Contabilidade Financeira'/
Documento.docx
Documentos/
'Email attachments'/
'IR Mercado Financeiro'/
Imagens/
IntelliJ/
'Lua de Mel'/
Música/
'Orçamento doméstico mensal.xlsx'
'PROPOSTA ANA MARIA ZANELATO PORTO 2022.pdf'
PROTOCOLO.xlsx
Protocolo_ADM/
Protocolo_ADM.url
'Traderplan manual'/
desktop.ini
'mat eletrico.xlsx'
'Area de Trabalho'/

USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive
$ cd Area\ de\ Trabalho/

```

Podemos ver a pasta Área de Trabalho, porém, se tentarmos executar o comando `cd Área de Trabalho` o terminal provavelmente disparará a mensagem de erro “bash: cd: too many arguments”. Isto acontece porque **pastas com nomes que contem palavras separadas por espaços não são interpretados como nomes de uma pasta**, mas cada palavra é interpretada como um novo comando.

Para resolver isto, sempre que tivermos o nome de uma pasta ou arquivo com duas ou mais palavras separadas por espaços, devemos **escreve-las entre aspas simples ou duplas, como se fosse uma string**

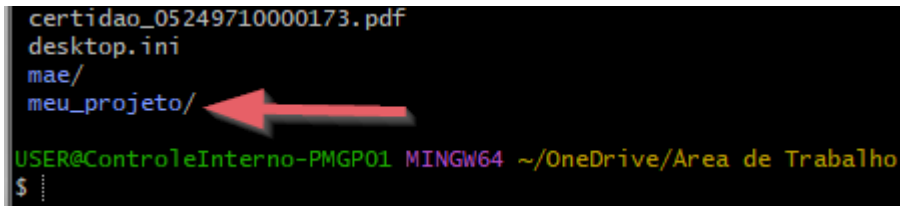
ATENÇÃO

mesmo que o acento não seja visível na linha de comando, não podemos esquecer dele, pois senão o terminal não reconhecerá o nome da pasta.

Uma boa prática entre programadores é escrever os nomes das pastas e arquivos em minúsculas e separando palavras com underlines (ex. “area_de_trabalho”).

Criar e deletar pastas

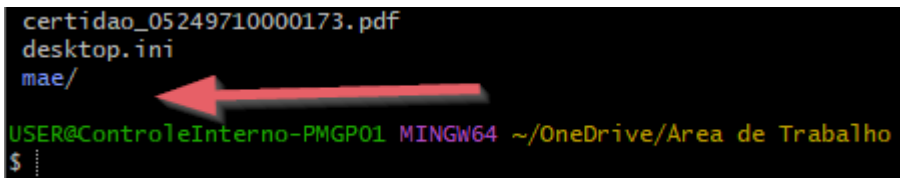
saremos o comando `mkdir meu_projeto` para criar uma nova pasta chamada `meu_projeto`. Na sequência, executemos o comando `ls` para verificar que ela foi criada.



```
certidao_05249710000173.pdf
desktop.ini
mae/
meu_projeto/
USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive/Area de Trabalho
$
```

A red arrow points to the newly created `meu_projeto/` directory in the listing.

Se percebemos que criamos a pasta no lugar errado? Simples, usamos o comando `rmdir meu_projeto` para deletar a pasta que acabamos de criar.



```
certidao_05249710000173.pdf
desktop.ini
mae/
USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive/Area de Trabalho
$
```

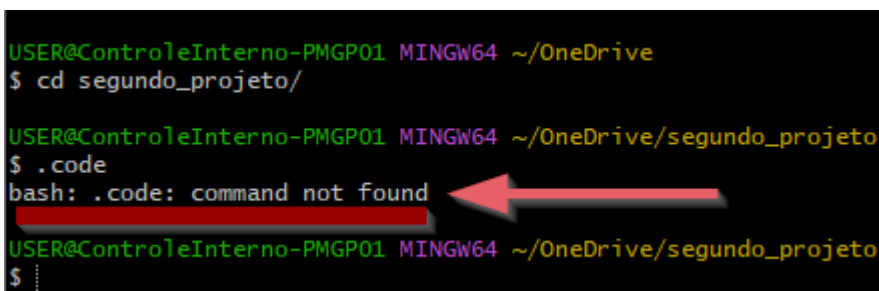
A red arrow points to the `mae/` directory, indicating that `meu_projeto` has been removed.

Usar o `cd ..` retorna uma pasta acima, um nível acima.

Abrir o VSCode através do Git

Se locomova através do Git até a pasta que deseja abrir no VSCode.

Uma vez dentro da pasta `segundo_projeto` podemos executar o comando `code`. (com o ponto no final mesmo). Se seguimos os passos de instalação do VSCode corretamente, este comando abrirá o VSCode na pasta atual. Contudo, caso isso não acontecer, não precisa se preocupar, pois temos outras formas de abrir um projeto no VSCode

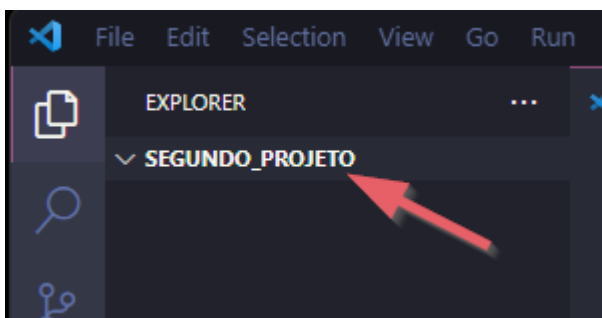


```
USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive
$ cd segundo_projeto/

USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive/segundo_projeto
$ .code
bash: .code: command not found
USER@ControleInterno-PMGP01 MINGW64 ~/OneDrive/segundo_projeto
$
```

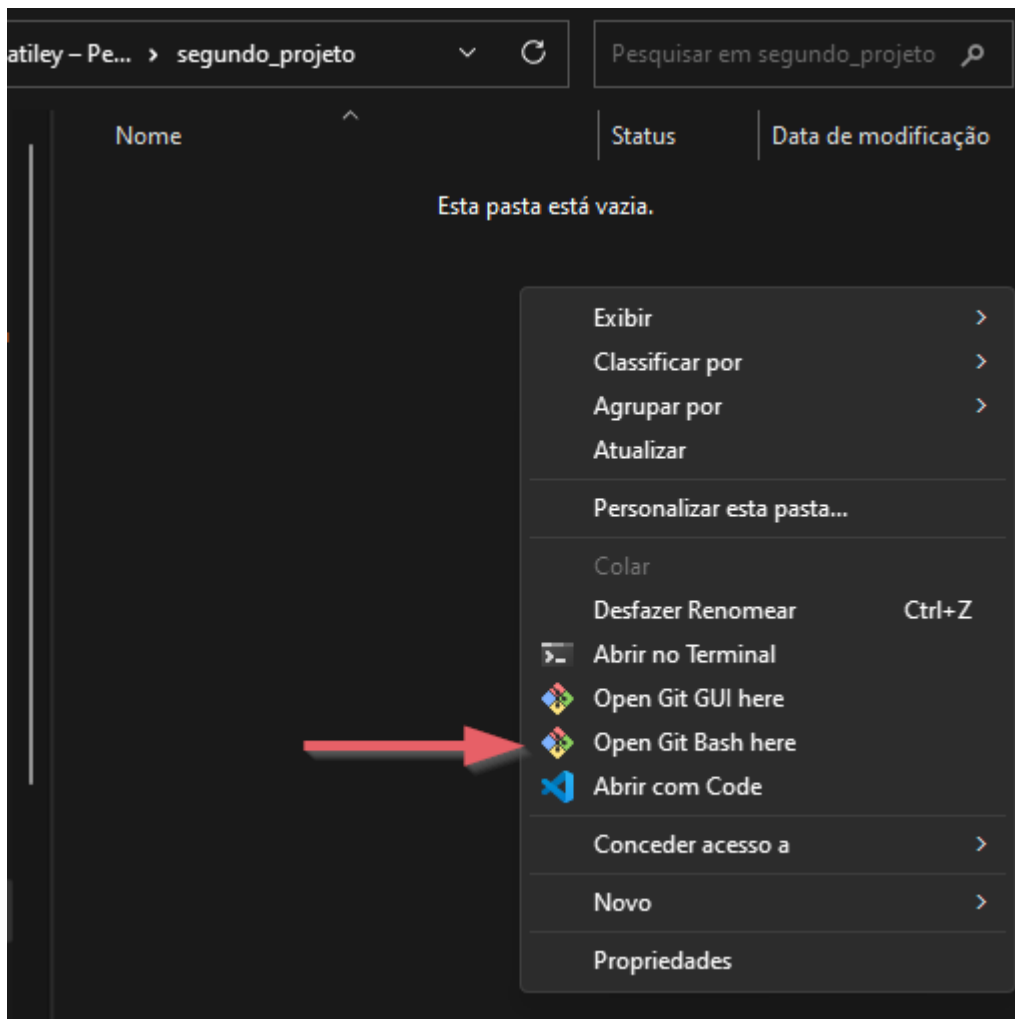
A red arrow points to the `bash: .code: command not found` error message.

Acima houve um erro, não foi dado espaço entre o ponto e a palavra `code`.



Alternativa via GUI

Navegue até a pasta que deseja abrir, clique com botão direito e selecione Git Bash Here



CONTROLE DE VERSÃO

Controle de versão

Em resumo, evitar que numa pasta contenha 20 arquivos sobre a mesma coisa, como por exemplo:

- Arquivo final
- Arquivo final_2
- Arquivo final_ok
- Arquivo final_ok_agoraVai
- Arquivo final_ok_Aprovado
- Arquivo final_ok_Aprovado_NovaVersão

Os sistemas de controle de versão, ou **VCS** (pelas siglas em inglês "Version Control System"), vêm para nos ajudar lidar com essas e outras situações, gerando um fluxo de trabalho mais organizado, que nos permite salvar um "histórico" das diferentes versões pelas quais nosso projeto passou.

Repositórios e Commits

Repositórios são lugares onde guardamos, ou arquivamos, alguma coisa. No mundo do desenvolvimento, os projetos que realizamos podem precisar de um enorme volume de pastas e arquivos. *No nosso computador, a*

pasta principal onde estão localizados todos esses arquivos e sub-pastas é o nosso **repositório local** (veremos depois outro tipo de repositório).

Commits Metáfora:

Imagine que está montando um móvel sem as instruções, pega as peças A e B e junta elas, deixa elas montadas do lado e junta agora as peças C e D que parece que vão juntas também. Contudo, chega uma hora que você fica na dúvida "Será que é assim mesmo? Acho que talvez era para as peças A, B e C irem juntas... mas se eu desmontar as peças C e D depois não vou lembrar como que elas iam juntas".

A solução que você acha é simples: tirar uma foto de como as peças estão montadas agora e escrever nela "A e B juntas, C e D juntas", assim se tentar encaixar as peças de outra forma mas se arrepender e resolver voltar à configuração original, basta dar uma olhada na foto que você tirou para ver como as peças estavam montadas antes.

O sistema parece funcionar, você vai tirando fotos ao longo da montagem e verificando as fotos anteriores para voltar um ou dois passos sempre que for necessário. É isso que são os commits! Porém, com uma vantagem, você não precisa "ajustar seu projeto" conferindo uma imagem de como ele estava antes, ao acessar um commit específico o projeto automaticamente se ajusta à forma como ele estava quando você "tirou a foto".

Arquivos que tinham sido modificados voltam aos seus formatos anteriores; pastas que você apagou aparecem novamente no seu repositório; um arquivo que você migrou da pasta home para a pasta usuario volta na home. **É essencialmente "viajar no tempo" dentro do nosso projeto!**

ATENÇÃO

Uma boa prática é salvar um commit cada vez que concluímos uma tarefa específica e começaremos uma nova.

Exemplo:



Branch e Merge

Branch

Branch, do inglês *galho* ou *ramificação*.

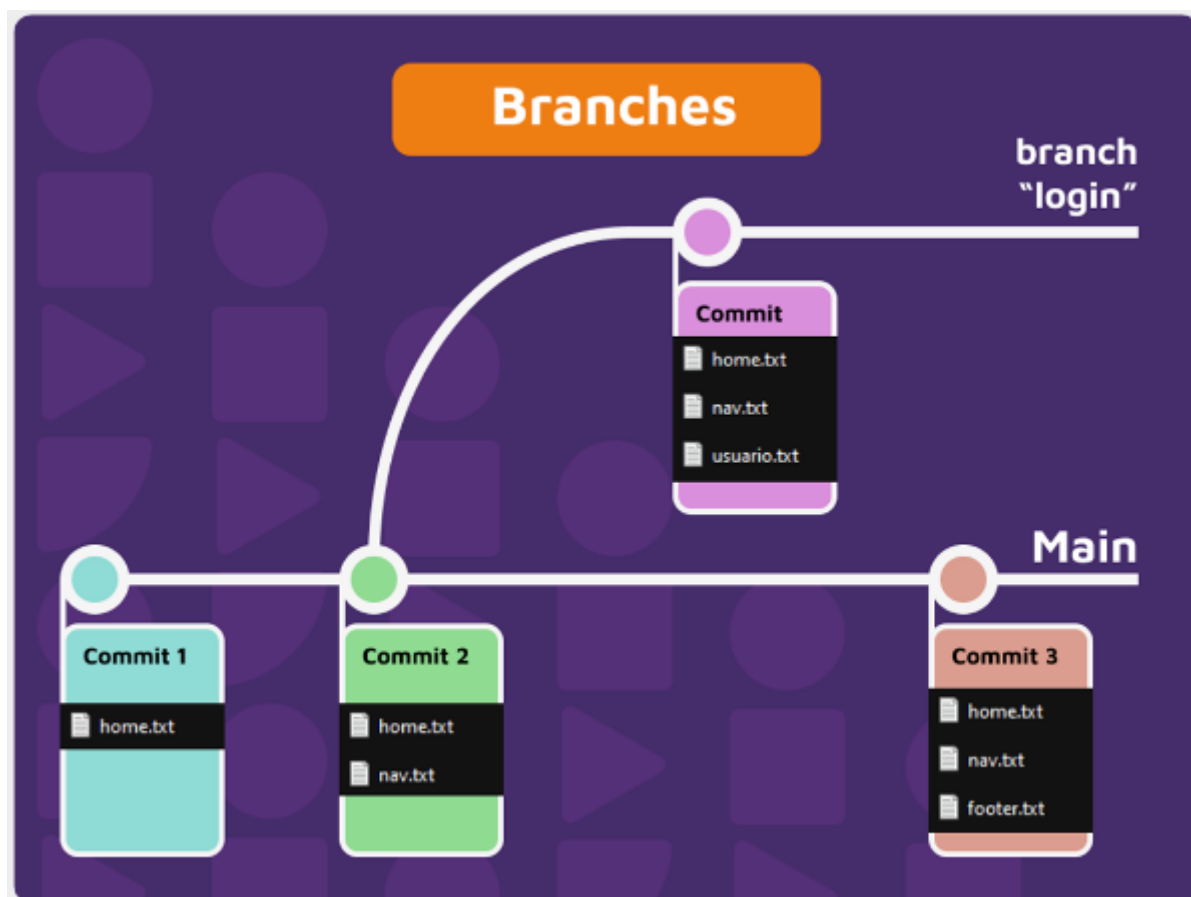
Nossa linha temporal principal é essa ramificação do fluxo principal de desenvolvimento.

DUAS BOAS PRÁTICAS SÃO NECESSÁRIAS

- Trabalho colaborativo
- Manutenção da "linha de tempo principal"

Duas pessoas escrevendo no word. Uma inclui um texto onde a outra queria reescrever. Esse é o **primeiro motivo** pelo qual usamos branches. Com elas, cada desenvolvedor consegue **trabalhar nas suas respectivas tarefas**, criando e alterando arquivos, **sem se preocupar nas mudanças que o resto da equipe faz em paralelo**

Para fazer isso, criamos uma "ramificação" a partir de um commit da "linha do tempo principal", que vira uma "linha do tempo secundária". Assim, as mudanças feitas no projeto nessa linha do tempo secundária não afetarão o projeto na linha do tempo principal.



Pensando num exemplo prático, observe a imagem acima. A main é a nossa linha de tempo principal e, se você observar, no segundo commit ela tem apenas dois arquivos: home.txt e nav.txt. A partir desse commit criamos uma linha do tempo secundária, a branch chamada login, e nessa branch criamos o arquivo usuario.txt.

Voltando agora para o terceiro commit na "main", vemos que também temos três arquivos, porém, o terceiro arquivo chama footer.txt, e não usuario. Isso acontece porque o arquivo usuario.txt foi criado numa outra branch ("login"), e as mudanças feitas nessa branch não afetam a "main". Da mesma forma, o arquivo

footer.txt também não consta na branch "login", pois criamos essa branch a partir do segundo commit, e nesse ponto do projeto só existiam os dois primeiros arquivos.

A **segunda motivo** que sustenta o uso de branches, é que a "main branch" **deve ser sempre uma versão, até certo ponto, completa e funcional do nosso sistema.**

Não queremos que alguém entre por exemplo num site(que é nosso projeto), e aperte um botão e ele não funcione, ou que o link não acesse a página requerida, que falte uma imagem, etc.

Por esse motivo, sempre que vamos implementar uma nova seção ao nosso site, ou um novo recurso no nosso sistema, é recomendável trabalhar ele numa branch separada, **e adicionar ela à main branch só depois que foi concluída e testada.**

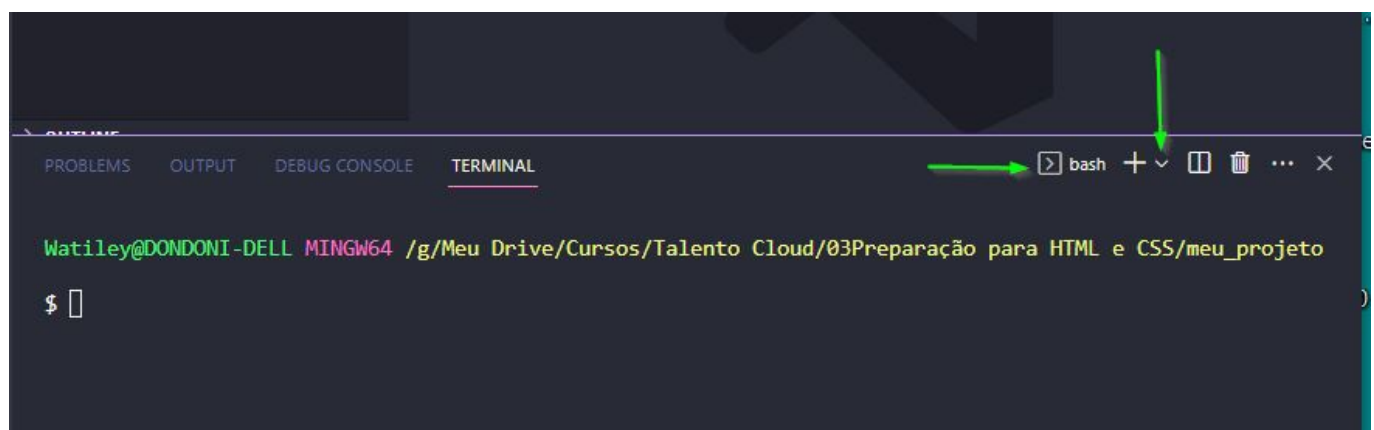
Merge

Finalmente, chamamos de **merge** ao ato de "juntar" branches. Como você pode imaginar, há situações nas quais fazer um merge é bastante simples, por exemplo, o caso anterior onde uma branch tem o arquivo usuario.txt e a outra não. Na hora de executar o merge o Git simplesmente adiciona o arquivo novo na main e problema resolvido! Contudo, e se ambas as branches tiverem o mesmo arquivo, porém, com informações diferentes? Isso é o chamamos de **conflito**, ou **merge conflict**. Para resolver esses conflitos, o Git nos pede para acessar os arquivos onde o conflito aconteceu para decidir quais linhas queremos manter, e quais queremos apagar.

Git Bash no VSCode

IDE e VCS são duas importantes ferramentas para o desenvolvedor. Podemos no VSCode abrir um terminal em New Terminal ("Novo Terminal" em português). Podemos também usar os atalhos de teclado Ctrl + ' ou Ctrl + J sempre que quisermos abrir e fechar um terminal no VSCode.

Por padrão, o VSCode abre o Shell do Windows como terminal



Git - Comandos I

Existem outros VCS, porém o Git é o mais conhecido.

Estrutura dos comandos



Palavra reservada - Todos os comandos do Git começam com a palavra reservada 'git'

Nome do comando - Ele, junto com a palavra reservada, são os únicos dois elementos imprescindíveis. Contudo, vale sinalizar que em certas situações precisamos escrever o nome de mais de um comando para realizar uma tarefa (ex. git remote add origin repo-url)

Esses dois comandos são os imprescindíveis.

Opção / Opções - Embora alguns comandos precisam apenas da palavra reservada e do nome do comando para serem executados (ex. git log), para outros podemos ou devemos especificar algumas opções do comando. Estas opções podem ser escritas com um hífen e uma letra (ex. -a, -m), ou dois hífen e uma palavra (ex. --all, --message)

Argumento - Alguns comandos precisam de um input nosso para "etiquetar" coisas, como a descrição de um commit, ou o nome de uma branch.

Inicializando um repositório local

Esse é um passo imprescindível, sem ele não há comits, branches, etc. Na pasta desejada, com o terminal aberto nessa pasta, digite **git init**, e a tecla enter. Sempre ao iniciar um projeto que terá o controle de versão será necessário inicializar o git.

Comandos de configuração

Comando	Opções	Função
git config user.name "NOME"		Definir o nome do usuário que usará Git neste projeto
git config user.name "NOME"	--global (após de config)	Definir o nome do usuário para todos os projetos neste computador
git config user.email "EMAIL"		Definir o EMAIL do usuário que usará Git neste projeto
git config user.email "EMAIL"	--global (após de config)	Definir o nome do usuário para todos os projetos neste computador

Atenção

Caso você for dividir seu computador com alguma pessoa que também usa o Git, é melhor não incluir este trecho do comando, contudo, será necessário definir o nome do usuário cada vez que iniciarmos um novo repositório (veremos como iniciar repositórios daqui a pouco).

O terminal não retorna "comando realizado com sucesso", porém se der erro, o Git sempre retornará um aviso.

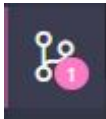
Utilize o mesmo nome e email da conta do GitHub. ATENÇÃO:

Lembrando que, se usamos a opção `--global` para estes comandos, **todos os próximos repositórios que inicializarmos nesse computador terão esse nome de usuário e email por padrão**, e não será necessário executar estes comandos novamente. Caso contrário (se não usou a opção `--global`), será necessário executar estes dois comandos após inicializar o repositório.

Comando	Opções/Argumentos	Função
git add	nome_do_arquivo.txt	Adiciona o arquivo especificado à área de staging
git add	-all ou -A	Adiciona todos os arquivos atuais (novos, atualizados e deletados)
git commit	-m "MENSAGEM"	Realizar um commit com uma mensagem que o identifique
git log		Mostra os últimos commits do repositório atual
git status		Mostrar os arquivos e pastas que estão na área de staging

Comandos para commits

Ao criar um arquivo num repositório do Git, com o VSCode aberto, percebe-se um aviso no ícono do Source



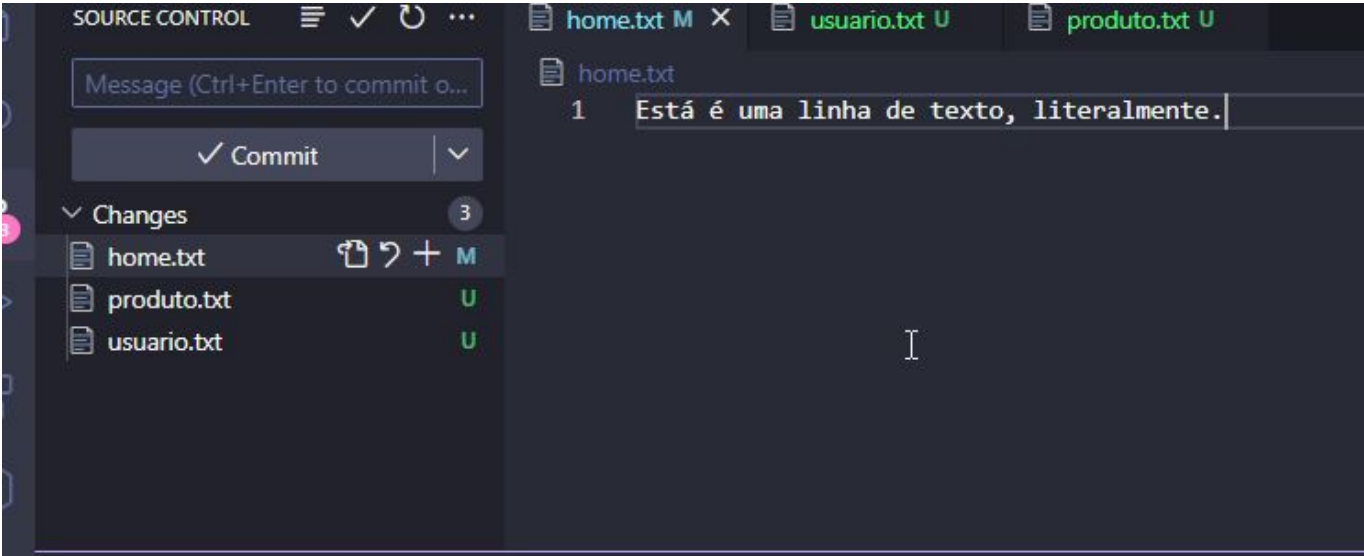
Control esse numeração indica que houve alteração no diretório.

Outra observação é que ao lado do texto criado, é o aparecimento da letra "U".

Esse 'U' significa que o arquivo **"Não está sendo rastreado"** pelo Git (de "Untracked" em inglês). Isso quer dizer que ainda precisamos dizer ao Git que queremos guardar um histórico de alterações nesse arquivo a cada commit (no futuro, veremos em quais situações podemos não querer que o Git "não guarde um histórico" de algum arquivo). Para indicar ao Git então que queremos "rastrear" esse arquivo, voltemos para o terminal e executemos o código **git add home.txt**

Após fazer isso, a letra ao lado do arquivo na barra lateral muda de 'U' para 'A'. Isso significa que o arquivo foi "Adicionado à área de staging". Voltando na analogia dos commits como uma fotografia do nosso projeto em um determinado momento, adicionar arquivos à área de staging é *como colocar os elementos que você vai querer na foto dentro do "campo de visão da câmera"*. Ou seja, todos os elementos que não estiverem na área de staging na hora de fazer o commit, ou "tirar a foto", não "vão aparecer na foto". Vamos então no terminal para fazer nosso primeiro commit com o comando **git commit -m "Arquivo home.txt criado"**

git log nos mostra as informações exibidas após realizar o commit, além do nome e email do autor do respectivo commit, e a data/hora em que o commit foi realizado.



Acima, foi criado dois novos arquivos e modificado outro. Observa-se o "U" nos dois novos e o "M" no home.txt, "M" de modified, pois já está sendo rastreado pelo Git, e foi adicionada uma linha de texto.

Adicionamos todas as alterações e inclusões de uma única vez fazendo uso do comando **git add -A** ou **git add --all**

git status, vai mostrar quais arquivos estão na **área de staging**.

```
Watiley@DONDONI-DELL MINGW64 /g/Meu Drive/
(master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   home.txt
    new file:   produto.txt
    new file:   usuario.txt
```

Os arquivos em verde são aqueles que sofreram mudanças desde o último commit e estão atualmente na área de staging. Caso tivéssemos arquivos que sofreram alterações, mas não foram adicionados ainda, eles apareceriam em vermelho. Além disso, o próprio Git nos indica um novo comando caso quisermos remover algum dos arquivos da área de staging (o comando git restore...), porém, não será necessário usá-lo. Para finalizar, realizemos nosso segundo commit usando o comando **git commit -m "Dois arquivos criados, home.txt alterado"**

Git - Comandos II

Comandos para branches

Comando	Opções/Argumentos	Função
git branch		Exibe a lista de branches disponíveis e destaca a branch atual
git branch	nome-da-branch	cria uma nova branch

Comando	Opções/Argumentos	Função
git checkout	nome-da-branch	Trocar o HEAD, ou ponteiro, para a branch indicada

git branch lista os branches do projeto

A terminal window showing the command `git branch` being executed. The output is `* master`. A red box highlights the output, with a red arrow pointing to a yellow box labeled "lista de branches". Another red box highlights the prompt `(master)` at the end of the command line, with a red arrow pointing to a yellow box labeled "HEAD, 'ponteiro'".

Uma vantagem de usar o Git Bash é que o HEAD é sempre exibido no final do nosso prompt de comando. O HEAD, também chamado de **ponteiro**, indica em qual branch do projeto estamos trabalhando atualmente. Por defeito, **sempre que inicializamos um repositório**, o HEAD estará apontando para a **branch main** (é possível que, ao invés de main, a branch principal esteja com o nome master).

Criar uma nova branch não nos joga pra lá automático. Devemos "ir" através do comando git checkout "NOME DA BRANCH"

A terminal window showing the command `git branch` being executed. The output is `* master` and `nova-branch`. Red arrows point from the text "Duas branches, a verde é a que estamos." to the `* master` and `nova-branch` lines.

Duas branches, a verde é a que estamos.

A terminal window showing the command `git checkout nova-branch` being executed. The output is `Switched to branch 'nova-branch'`. Red boxes highlight the prompt `(master)` before the command and `(nova-branch)` after the command. Red arrows point from the text "Realizamos a mudança da branch Master para nova-branch." to the `Switched to branch 'nova-branch'` line and the `(nova-branch)` prompt.

A terminal window showing the command `git branch` being executed. The output is `master` and `* nova-branch`. A red arrow points from the text "Realizamos a mudança da branch Master para nova-branch." to the `* nova-branch` line.

Realizamos a mudança da branch Master para nova-branch.

MODULO 03

GitHub

GitHub o que é?

Lembre-se Git é != de GitHub

Git é um VCS, Version Control System, que controla o versionamento, as alterações de um projeto, desde seu nascimento até sua conclusão, e até após isso.

GitHub, é uma rede social, o qual é uma vitrine de nossos projetos, e podemos até colaborar com outras equipes em vários projetos.

Cadastro no GitHub

GitHub é um site que **hospeda** não só nossos projetos em nuvem, mas também **todo o histórico de versões desses projetos gerado pelo Git**. Por um lado, ele nos permite, por exemplo, começar um projeto no nosso PC no escritório, depois baixar e continuar trabalhando no projeto no nosso notebook em casa, retomar ele novamente no PC quando voltamos no escritório, e assim pela frente.

Link para cadastro no GitHub <https://github.com/>

ATENÇÃO!

É recomendável você usar o mesmo endereço de email que usou para configurar o Git no seu computador.

Caso o endereço for inválido ou já existir uma conta no GitHub associada a ele, uma mensagem de advertência será mostrada embaixo do input e não será possível clicar no botão Continue

Em definir um nome de usuário, este não precisa necessariamente ser nosso nome real, porém, **é recomendável que seja um nome pelo qual nossos colegas consigam nos reconhecer** (ex. Se meu nome é "João Alves", o nome de usuário "JAlves87" é um nome recomendável, "gamerVHalen5522" e afins não é um nome recomendável.

Quando o cadastro é feito sem erros no sistema, ele nos leva para a uma tela o qual será necessário então conferir o código 8 dígitos enviado no email que informamos.

Repositórios Remotos

Pastas criadas só no PC e iniciadas pelo comando git init ocupam memória e espaço do computador localmente, e são os chamados de **"Repositórios Locais"**

Porém podemos *enviar essa pasta para o GitHub*, e esse envio é chamado de **"Clone"**, onde será utilizado espaço remoto.

Diferentemente do nome de usuário, o nome do nosso repositório deve ser inédito **apenas entre nossos repositórios** (ex. não podemos ter dois repositórios chamados meu_projeto).

Boa prática que o nome da pasta(Projeto) seja o mesmo entre os repositórios locais e web

O repositório pode ser público, onde qualquer tem acesso. Ou pode ser Privado, apenas quem criou tem acesso.

Formas para SINCRONIZAR:

1. Utilizar o link do repositório remoto, caso já tivermos experiência com os comandos do Git.
2. casos em que ainda não temos um repositório local
3. caso já tivermos um repositório local e quisermos sincronizá-lo via linha de comando (terminal)

4. subir os arquivos do nosso repositório local de forma manual (vale apontar que esta opção não sincroniza ambos repositórios de forma automática, e serve mais como um backup).

GitHub - Comandos

A tríade dos comandos do Git mais usados no dia a dia de um desenvolvedor:

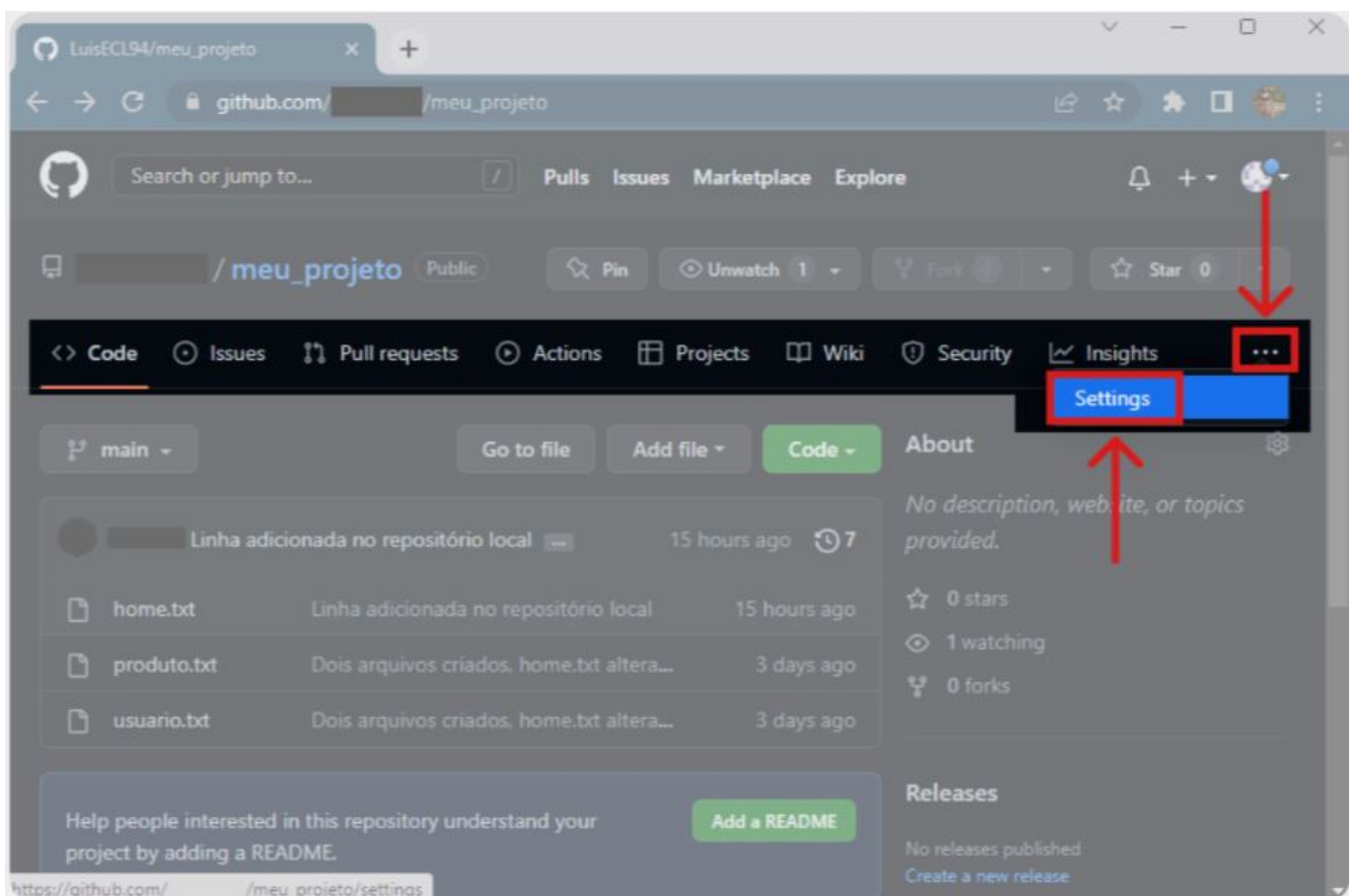
git add -A, coloca os arquivos no staging(cena da foto)

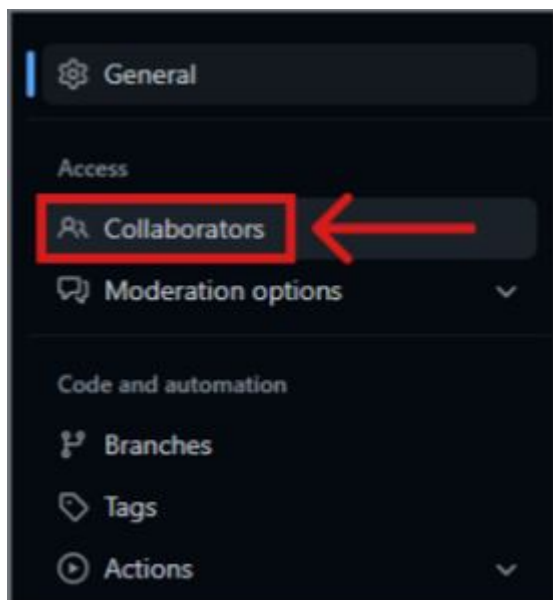
git commit -m "MENSAGEM", "tira a foto", ou "embala numa caixa os add e rotula".

git push ENVIA(empurra as informações) commits do nosso repositório LOCAL ao repositório REMOTO.

Adicionar colaboradores

o dono do repositório precisa ir na Navegação do repositório remoto e clicar na opção Settings (caso a resolução da sua tela for menor, é necessário clicar nos três pontos para aparecer a opção)





Procuraremos pelos nossos colaboradores via nome de usuário, nome completo, ou email. Após achar, clique em **Add... to this repository**

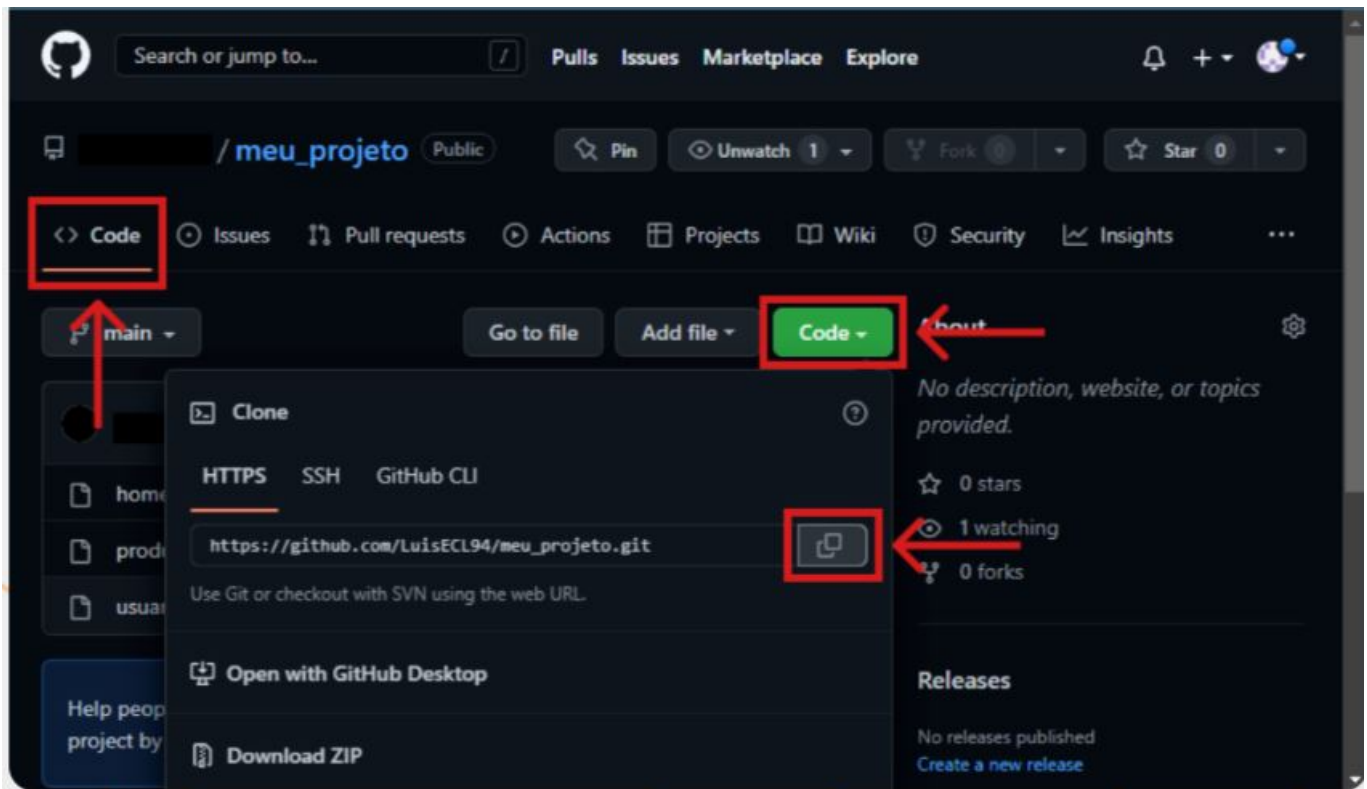
eles poderão acessar o repositório desde a barra lateral nas suas respectivas telas iniciais

Clonar um repositório remoto

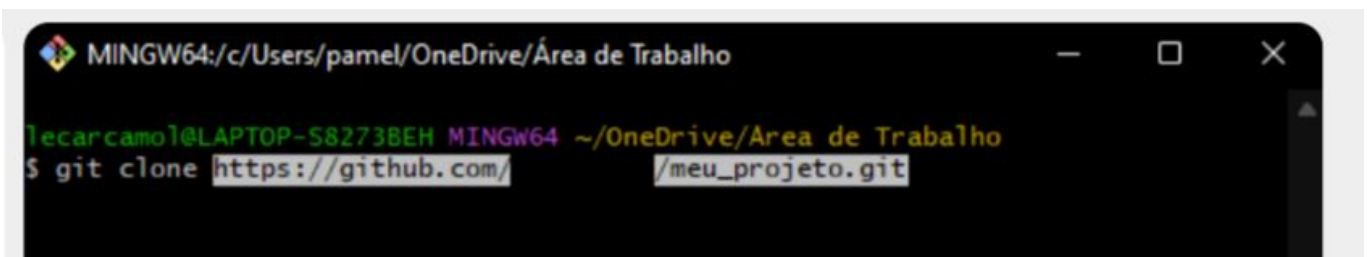
Vamos criar um repositório local a partir de um repositório remoto. Isso é útil tanto para os colaboradores que acabamos de adicionar no nosso repositório remoto, quanto para nós mesmos se quisermos continuar trabalhando no nosso projeto desde qualquer outro computador que não seja aquele que tem o repositório local original.

git clone LINK-DO-REPOSITÓRIO

O comando acima CRIA um repositório LOCAL a partir de um repositório REMOTO



Acima estamos pegando o link para fazer o clone do servidor remoto para o computador local.



Atualizar nosso repositório local

git remote show origin, faz uma comparação entre o repositório local e o remoto e retorna se houve ou não alguma alteração.

git pull, ATUALIZAR repositório LOCAL(puxa as informações) com informações do repositório REMOTO.

ATENÇÃO

Embora o comando `git remote show origin` nos mostra informações úteis que podemos usar em várias outras circunstâncias, para casos nos quais estamos trabalhando com uma equipe de desenvolvedores e sabemos que sempre existe a possibilidade de um colaborador ter commitado mudanças no repositório remoto, é uma boa prática usar o comando `git pull` antes de começar trabalhar no nosso repositório local.

MODULO 04

REVISÃO - ARRAYS, GIT E GITHUB

Arrays

Também chamados de vetores ou matrizes; São estruturas que usamos para **guardar e organizar** dados em uma **única variável**; **Aceitam todos os tipos de dados** estudados como elementos.

Declaração dos arrays

- Nome da variável
- Operador de atribuição '='
- Par de colchetes '[]'
- Itens separados por vírgulas
 - Ex. nomes = ["Luciano","Juliana","Caroline"]

índices

- Usados para acessar um elemento do array por vez;
- O primeiro elemento de um array terá **sempre o índice 0**;
- Para usá-los escrevemos o nome da variável que guarda o array, seguindo do índice entre colchetes;
- Ex. nomes[0] nos dará acesso ao primeiro item do array 'nomes'

Função len()

- Recebe como argumento a variável que guarda um array;
- Retorna a quantidade de itens no array;
- Ex. len(nomes) imprimirá '4', pois o array 'nomes' tem quatro elementos.

Percorrer arrays

- O ato de acessar todos os elementos de um array para manipulá-los de alguma forma;
- Uma das formas mais comuns para percorrer arrays é usando a variável contadora de um laço **for** como "índice dinâmico";
- É comum usar o laço **for** em conjunto com as funções **range()** e **len()** para definir a quantidade de elementos do array como condição de parada;
- **for i in range(len(nomes)):print(num[i])**

```
for i in range(len(nomes)):
    print(nomes[i])
```

Alterar elementos usando índices

- Podemos acessar um elemento de um array e definir um novo valor a ele usando o operador de atribuição:

```
nomes[0] = "Lu"
```

- Podemos acessar mais de um elemento e atribuir mais de um valor com um operador de atribuição só, separando os elementos e novos valores por vírgulas:

```
nomes[1], nomes[3] = "Ju", "Carol"
```

Adicionar e remover elementos

- Função **append()** para adicionar elementos *no final de um array*;
- Escrevemos o nome do array, seguido de um ponto, e a função com o novo elemento passado como argumento:

```
nomes.append("Pamela")
```

- Função **pop()** para *remover o último elemento de um array*;
- Escrevemos o nome do array, seguido de um ponto, e a função:

```
nomes.pop("Pamela")
```

Sistemas de controle de versão (VCS)

- Servem para **gerar um fluxo de trabalho organizado**, salvando um histórico de versões do nosso projeto e criando “linhas de trabalho” paralelas.

Repositórios

- Lugar onde guardamos os arquivos e pastas dos nossos projetos;
- Podem ser locais (ex. no nosso computador) ou remotos (ex. No GitHub)

Commits

- Uma “**fotografia**” das mudanças feitas no nosso projeto em um determinado estágio do seu desenvolvimento;
- Uma boa prática é realizar commits *após a conclusão de tarefas específicas*.

Branch's

- Ramificações **no** fluxo principal de trabalho;



- Ajudam com o trabalho colaborativo e com a **manutenção da linha de trabalho principal**.

Git'

*VCS muito popular atualmente, especialmente no mundo de desenvolvimento web

Estrutura dos comandos'

- Palavra reservada - é sempre **git**;
- Nome do comando – pode ser um ou mais de um;
- Opção/Opções - são escritas com um hífen e uma letra, ou dois hífen e uma palavra (ex. -A ou --all);
- Argumento – Input do desenvolvedor para nomear commits, branches, etc.



COMANDO	OPÇÃO/OPÇÕES	FUNÇÃO
git init		Inicializar um repositório na pasta atual
git config user.name "NOME"		Definir o nome do usuário que usará Git neste projeto
git config user.name "NOME"	--global (depois de config)	Definir o nome do usuário para todos os projetos neste computador
git config user.email "EMAIL"		Definir o email do usuário que usará Git neste projeto
git config user.email "EMAIL"	--global (depois de config)	Definir o email do usuário para todos os projetos neste computador
git add	nome_do_arquivo.txt	Adiciona o arquivo especificado à "area de staging"
git add	--all ou -A	Adiciona todos os arquivos atuais (novos, atualizados e deletados)
git commit	-m "MENSAGEM"	Realizar um commit com uma mensagem que o identifique
git log		Mostra os últimos commits do repositório atual




COMANDO	OPÇÃO/OPÇÕES	FUNÇÃO
git status		Mostrar os arquivos e pastas que estão na área de staging
git branch		Exibe a lista de branches disponíveis, e destaca a branch atual
git branch	nome-da-branch	Cria uma nova branch
git checkout	nome-da-branch	Trocar o HEAD, ou ponteiro, para a branch indicada
git merge	nome-da-branch	Trazer os elementos da branch indicada no comando, para a branch atual
git merge	--abort	Aborta o processo de merge em casos em que Git identifica um conflito, retornando o projeto ao seu estado anterior à tentativa de merge







GitHub'

- GitHub é um site que **hospeda nossos projetos em nuvem, e todo o histórico de versões desses projetos** gerado pelo Git;

- Ele funciona como uma rede social, podemos seguir e acompanhar os projetos públicos de outros desenvolvedores, e trabalhar de forma colaborativa;
- Após criar uma conta, conseguimos criar repositórios **remotos** e sincronizá-los com nossos repositórios **locais**;

Comandos para efetuar as interações entre repositório remoto e local

[Link da Tabela](#)

GITHUB		
Comandos para efetuar as interações entre o repositório remoto e os repositórios locais		
COMANDO	OPÇÃO/OPÇÕES	FUNÇÃO
	link-do-repositório	Criar um repositório local a partir de um repositório remoto
		Enviar commits do nosso repositório local ao repositório remoto
		Verificar se nosso repositório local está atualizado com o repositório remoto
		Atualizar nosso repositório local com as últimas informações no repositório remoto

[Form para questionário de revisão](#)

Quiz de revisão - Arrays, Recursividade e Git

Total de pontos **10/10**

Responda o seguinte questionário para autoavaliar seu desempenho até esse momento do curso. Cada questão tem apenas uma resposta correta. Sua pontuação final e as respostas corretas para cada questão serão exibidas após o envio das respostas ;)

