UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA

DEPARTAMENTO DE ESTATÍSTICA

DISCIPLINA: INTRODUÇÃO AOS SOFTWARES ESTATÍSTICOS

Fluxo de Trabalho: Integrando Git, GitHub e Quarto com Introdução à Programação em Python

Aila Soares Ferreira
matrícula: 20240045022

João Pessoa - PB

Julho 2025

1.0 INTRODUÇÃO

Com os avanços do *Big Data* - termo que descreve grandes volumes de dados de alta velocidade, complexos e variáveis, que passaram a exigir técnicas avançadas para captura, armazenamento, distribuição, gerenciamento e análise de informações (Foundation 2012) - observamos que o valor desses dados só é plenamente aproveitado quando utilizado para tomada de decisões estratégicas. Para isso, as organizações precisam de processos eficientes que transformem dados brutos, muitas vezes dispersos e dinâmicos, em *insights* significativos (Gandomi e Haider 2015).

Além disso, conforme um projeto avança, arquivos são constantemente modificados e compartilhados entre colaboradores, exigindo gestão adequada para garantir integridade e acessibilidade. O gerenciamento de dados envolve não apenas armazenamento, mas também preparação e recuperação eficiente para análise (Bryan 2017).

Nesse contexto, este trabalho foi desenvolvido como requisito da disciplina, Introdução aos Softwares Estatísticos, com o objetivo de integrar ferramentas fundamentais para a prática atual da estatística, como:

1. **Git e GitHub** - Sistema de controle de versões distribuído e plataforma de hospedagem
2. **Quarto** - Ferramenta para criação de relatórios dinâmicos e reprodutíveis
3. **Python** - Linguagem de programação para análise de dados

2.0 Controle de Versão

Antes de falarmos sobre Git e GitHub, precisamos entender o conceito de **controle de versão**. Esse sistema registra todas as mudanças em arquivos ao longo do tempo, permitindo que você volte a versões anteriores quando necessário. Ele guarda todo o histórico de alterações em um banco de dados. Se acontecer algum erro, é possível voltar atrás e comparar diferentes versões do projeto (Bryan e Hester 2020).

2.2 Benefícios do Controle de Versão

Os principais benefícios são:

1. **Histórico completo** de todas as mudanças:
    - Todas as alterações feitas por diferentes pessoas
    - Criação e exclusão de arquivos
    - Mudanças no conteúdo
2. **Ramificação (branching) e junção (merging)**:
    - Trabalhar em partes separadas do projeto
    - Criar "ramos" para desenvolver coisas novas
    - Juntar tudo depois no projeto principal
3. **Rastreamento**:

- Saber quem fez cada mudança
- Quando e por que foi feita
- Comparar versões diferentes

Os sistemas de controle de versão passaram por significativas melhorias nas últimas décadas. Dentre as soluções atualmente disponíveis, o Git se consolida como uma das ferramentas mais adotadas para controle de versão (Chacon e Straub 2014; Bryan e Hester 2020).

3.0 Git e GitHub

O Git é um sistema de controle de versão distribuído, criado por Linus Torvalds em 2005 com o objetivo de gerenciar o desenvolvimento colaborativo do kernel do Linux. Sua arquitetura permite que múltiplos desenvolvedores trabalhem simultaneamente em um projeto, mantendo um histórico completo e seguro das alterações realizadas.

Com o passar dos anos, o Git passou a ser amplamente adotado em diferentes áreas, incluindo a ciência de dados, onde tem se mostrado uma ferramenta eficiente no gerenciamento de projetos complexos. Esses projetos frequentemente envolvem uma coleção heterogênea de arquivos, como scripts de análise computacional, relatórios dinâmicos, bases de dados e visualizações gráficas (Bryan 2017) (Chacon e Straub 2014).

3.1 Fluxo de Trabalho Básico do Git

O fluxo de trabalho básico com Git pode ser resumido nas seguintes etapas:

1. **Modificação**
   - Edição de arquivos na árvore de trabalho
   - Criação, remoção ou alteração de documentos
2. **Preparação**
   - Seleção das mudanças que devem ser incluídas no próximo commit (`git add`)
   - Organização lógica das alterações
3. **Consolidação**
   - Registro definitivo das mudanças com uma mensagem descritiva (`git commit`)
   - Criação de pontos de restauração no histórico do projeto
4. **Sincronização**
   - Integração com repositórios remotos por meio dos comandos `git push` e `git pull`
   - Garantia de que todas as versões do projeto permaneçam consistentes entre os colaboradores

3.2 GitHub

O GitHub é uma plataforma de hospedagem de código-fonte que complementa o uso do Git, fornecendo uma interface gráfica baseada na web e recursos avançados para o gerenciamento de repositórios Git. Enquanto o Git é uma ferramenta local utilizada para registrar e controlar as alterações em arquivos, o GitHub funciona como um serviço remoto, permitindo que esses repositórios sejam armazenados, sincronizados e compartilhados pela internet.

Muitas operações podem ser feitas inteiramente no Github, como exemplo: **Visualizar e editar arquivos; revisar e gerenciar solicitações de mudanças; e Controlar permissões de acesso aos projetos**.

3.3 Comandos básicos: `git init`, `git add`, `git clone`, `git commit`, `git push`, `git pull`

3.3.1 `git init`

O comando `git init` cria um novo repositório do Git. Ele é utilizado para transformar uma pasta comum do computador em um repositório Git, ou seja, um ambiente com controle de versão. Isso permite acompanhar todas as alterações feitas no projeto ao longo do tempo.

A seguir, apresentamos um exemplo prático com base no desenvolvimento deste próprio relatório.

Exemplo prático: iniciando o Git em um projeto Quarto

Etapas

1. No Positron criar uma nova pasta
2. Dentro da pasta, criar um novo arquivo chamado `relatorio2.qmd`.
3. A seguir, abri o terminal navegar na pasta do projeto com o comando *c*

Transformando em repositório Git

No terminal, execute os seguintes comandos:

```
git init #Comando para iniciar repositório
git status #Comando para verificar o status dos arquivo
```

3.3.2 `git add`

O comando `git add` prepara arquivos para serem incluídos no próximo *commit*. Ou seja, ele diz ao Git quais arquivos você quer salvar no histórico de versões. Você deve usar sempre que, criar um arquivo, modificar ou deletar algum arquivo.

Usando o git add

No terminal, execute os seguintes comandos:

```
git add . # Adicionar todos os arquivos modificados.
          # O ponto . significa  "tudo no diretório atual e subpastas"
git add relatorio2.qmd #Adicionar um arquivo específico
```

Depois de adicionar, os arquivos ficam em um estado chamado **"staged"** (preparado). Eles só serão salvos de verdade no Git quando você fizer o commit: `git commit -m` "mensagem explicando a mudança".

3.3.3 `git clone`

O comando `git clone` é usado para copiar um repositório Git existente para o seu computador. Ele cria uma cópia completa do projeto, incluindo todos os arquivos, histórico de versões e estrutura de diretórios. Você pode clonar repositórios que estejam hospedados localmente ou em servidores remotos, como o GitHub. Esse comando é geralmente o primeiro passo para começar a colaborar em um projeto existente.

Usando o git clone

No terminal, execute o seguinte comando:

```
git clone https://github.com/usuario/nome-do-repositorio.git
```

Esse comando irá criar uma pasta chamada nome-do-repositorio com todos os arquivos e histórico do repositório remoto.

3.3.4 `git commit`

O comando `git commit` registra um instantâneo das alterações preparadas no projeto. Esse instantâneo representa uma versão segura do projeto naquele momento, que será armazenada no histórico do repositório.

Antes de executar o `git commit`, é necessário utilizar o comando `git add`, que serve para selecionar as alterações que serão incluídas nesse instantâneo. Assim, `git add` prepara os arquivos, e `git commit` os salva definitivamente no repositório local.

Além disso, o comando `git status` pode ser usado a qualquer momento para verificar quais arquivos foram modificados, quais já foram preparados com `git add` e quais ainda não foram incluídos no próximo commit.

Esses três comandos — `git add`, `git status` e `git commit` — são frequentemente utilizados em conjunto e formam o fluxo básico de trabalho com Git, permitindo um controle eficiente sobre a evolução de um projeto.

Usando o git add, git status e git commit

No terminal, execute os seguintes comandos:

```
git status # Comando para verificar o status dos arquivo
git add . # Adicionar todos os arquivos modificados
git add relatorio2.qmd # Adicionar um arquivo específico
git status # Verificar se o arquivo foi adicionado corretamente
git commit -m "mensagem explicando a mudança" # Comando para fazer o commit
```

3.3.5 `git push`

O comando `git push` é usado para enviar os commits feitos localmente para um repositório remoto, como o GitHub. Ele efetivamente publica suas alterações para que fiquem disponíveis online e possam ser vistas ou utilizadas por outras pessoas. O `git push` pode ser considerado um comando de *"upload"*, enquanto `git fetch` e `git pull` podem ser considerados comandos de *"download"* . Esse comando deve ser utilizado após o `git commit`, pois apenas os commits podem ser enviados ao repositório remoto.

Usando o git push

Após adicionar e fazer commit das alterações, use o seguinte comando para enviar ao repositório remoto:

```
git push origin main
```

Esse comando envia as alterações da branch principal (main) para o repositório remoto chamado origin.

Origin é o nome padrão que o Git atribui ao repositório remoto ao usar `git clone`. A *branch* principal pode ser chamada de *main* (padrão atual adotado desde 2020 por GitHub, GitLab e Bitbucket para promover linguagem mais inclusiva) ou *master* (termo tradicional ainda encontrado em projetos antigos). Ambos funcionam da mesma forma, variando apenas na nomenclatura. Para verificar qual seu repositório utiliza, execute `git branch -a`.

3.3.6 `git pull`

O comando `git pull` é usado para buscar e baixar conteúdo de repositórios remotos, atualizando imediatamente o repositório local para que os conteúdos fiquem iguais. Ele combina automaticamente duas operações: `git fetch` (que baixa as alterações do repositório remoto) e `git merge` (que integra essas mudanças ao branch local). Esse comando é essencial para manter seu trabalho atualizado com as contribuições de outros colaboradores, sendo um dos principais responsáveis pela sincronização de conteúdo remoto.

Usando o git pull:

```
git pull origin main
```

Esse comando é equivalente aos comandos `git fetch` + `git merge`

3.3.7 Resumo do Fluxo de Trabalho

O fluxo de trabalho com **Git**, **GitHub** e **Quarto** segue uma sequência simples e eficiente:

1. **Criar ou clonar um repositório** → Começa-se criando um repositório local ou clonando um repositório existente do GitHub.

2. **Modificar os arquivos** → Edita-se o conteúdo do projeto (por exemplo, um arquivo `.qmd` no Quarto).

3. **Verificar o status** (`git status`) → Confere-se quais arquivos foram alterados.

4. **Adicionar mudanças** (`git add`) → Seleciona-se os arquivos a serem salvos.

5. **Salvar com um commit** (`git commit -m "mensagem"`) → Registra-se um ponto da versão do projeto.

6. **Enviar para o GitHub** (`git push`) → As alterações locais são enviadas para o repositório remoto.

Esse processo garante que o projeto esteja **organizado, versionado e disponível online**, facilitando a colaboração e a atualização contínua do trabalho.

4.0 Python

Python foi criada por Guido van Rossum em 1991 como uma linguagem de programação de alto nível e multiparadigma (Artima Developer, *The Making of Python*). Seu código-fonte está disponível sob a licença GNU General Public License (GPL).

É uma linguagem poderosa e de fácil aprendizado. Sua filosofia de design enfatiza a legibilidade do código, permitindo que os programadores expressem conceitos com menos linhas do que seria necessário em linguagens como C++ ou Java.

Um programa em Python segue uma estrutura simples, composta por um **algoritmo**, que é uma sequência de passos definidos para realizar uma tarefa. Esse algoritmo geralmente envolve três partes:

- **Entrada:** dados fornecidos ao programa;
- **Processamento:** etapas e operações que transformam os dados;
- **Saída:** o resultado final.

Exemplo simples de um programa em Python:

```
a = 1
b = 2
soma = a + b
print(soma) #3
```

Esse pequeno código realiza a soma de dois números e exibe o resultado (3) na tela.

4.1 Conceitos iniciais de Python

4.1.1 Tipos de dados: `int`, `float`, `str`, `bool`

Em Python, os tipos de dados primitivos são fundamentais para a construção de qualquer programa. Entre os principais tipos estão os numéricos (`int`, `float`, `complex`), as cadeias de texto (`str`) e os valores lógicos (`bool`).

Tipos numéricos

- `int`: representa números inteiros (sem parte decimal), positivos ou negativos.
- `float`: representa números de ponto flutuante (com parte decimal).
- `complex`: representa números complexos, com parte real e imaginária.

```
# Exemplos
numero_inteiro = 10           # int
numero_decimal = 3.14         # float
numero_complexo = 2 + 3j       # complex

print(type(numero_inteiro))   # <class 'int'>
print(type(numero_decimal))   # <class 'float'>
print(type(numero_complexo))  # <class 'complex'>
```

Tipo `str` (string)

Representa sequências de caracteres (texto). Pode ser definido com aspas simples (`' '`) ou duplas (`" "`):

```
# Exemplos de strings
nome = "Aila"
mensagem = 'Olá, mundo!'

# Operações com strings
print(type(nome))      # Saída: <class 'str'>
print(nome.upper())    # Saída: AILA
```

Tipo `bool` (booleano)

O tipo `bool` representa valores lógicos: `True` (verdadeiro) ou `False` (falso). É usado principalmente em expressões condicionais.

```
maior_de_idade = True
tem_carteira = False

print(type(maior_de_idade))    # <class 'bool'>

# Exemplo de uso em comparação
idade = 20
print(idade > 18)  # True
```

4.1.2 Variáveis e Operadores Básicos

Variáveis em Python

Em Python, as variáveis são **dinamicamente tipadas**, diferentemente de linguagens *statically typed* onde o tipo é fixo após a declaração. Isso significa que uma mesma variável pode receber valores de tipos diferentes durante sua vida útil.

Operadores em Python

Operadores são símbolos especiais que executam operações aritméticas ou lógicas. Eles são classificados em:

Principais Categorias:

1. **Aritméticos**: Realizam cálculos matemáticos
   ```
   print(10 + 3)  # Adição
   print(10 ** 2) # Exponenciação
   ```
2. **Atribuição**: Atribuem valores a variáveis
   ```
   x = 5
   x += 3  # Equivalente a x = x + 3
   ```
3. **Comparação**: Comparam valores
   ```
   print(5 > 3)   # True
   print(5 == 3)  # False
   ```
4. **Lógicos**: Combinam expressões booleanas
   ```
   print(True and False)  # False
   print(True or False)   # True
   ```

5. **Identidade**: Verificam se objetos são os mesmos

```
a = [1,2]
b = a
print(a is b)  # True
```

6. **Associação**: Verificam se um elemento está presente

```
print('a' in 'Python')  # False
```

4.1.3 Listas, Tuplas e Dicionários

Python possui quatro estruturas de dados para coleções:

1. Listas Coleção **ordenada** e **mutável**. Permite itens duplicados.

```
# Criando uma lista
L = ['V', 'e', 'j', 'a']
print(L)  # ['V', 'e', 'j', 'a']

# Slicing (acesso a partes da lista)
print(L[1:3])  # ['e', 'j']
print(L[-1])   # 'a' (último elemento)

# Modificando a lista
L.append('!')  # Adiciona no final
L[0] = 'v'     # Altera um elemento
```

2. Tuplas Coleção **ordenada** e **imutável**. Permite duplicados.

```
# Criando uma tupla
t = (1, 3, 5)
print(t[1])  # 3

# Tentativa de modificação (gera erro)
# t[0] = 2  # TypeError
```

3. Sets Coleção **não-ordenada** e **não-indexada**. Não permite duplicados.

```
s = {"maçã", "banana", "maçã"}
print(s)  # {"maçã", "banana"} (duplicado removido)
```

4. Dicionários Coleção **não-ordenada**, **mutável** e **indexada por chaves**.

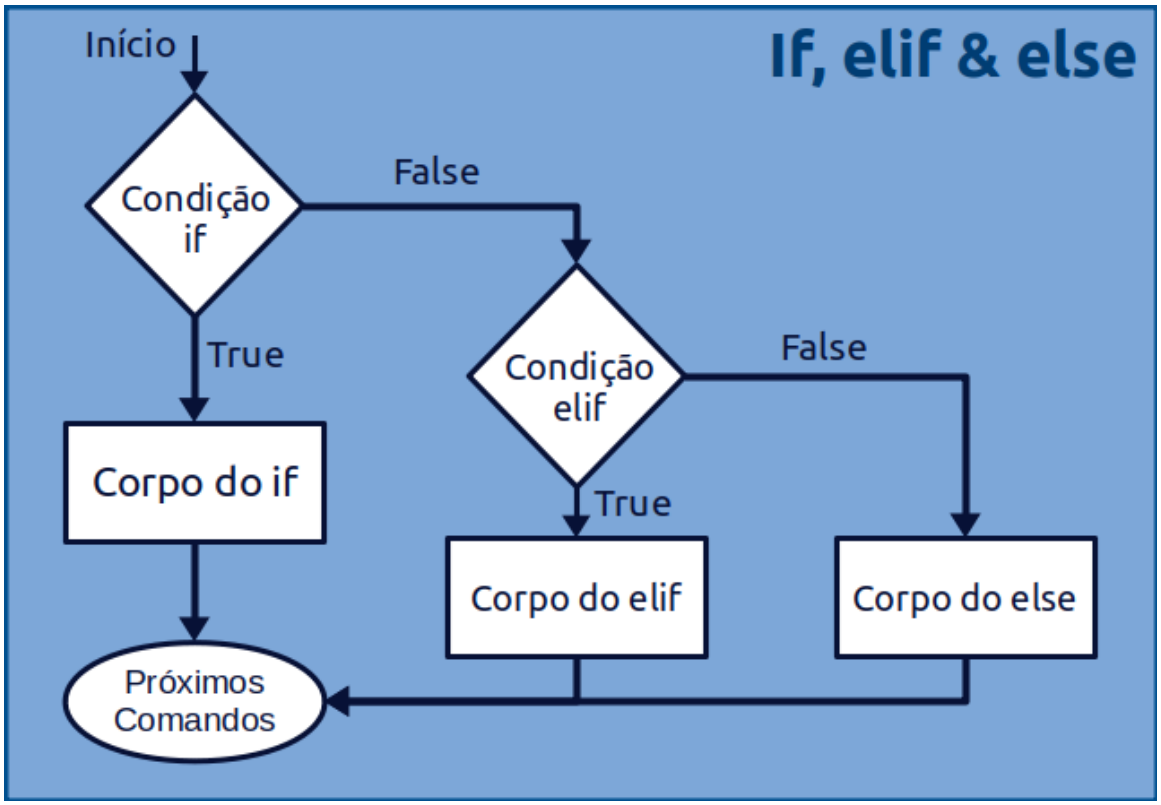
```
# Criando um dicionário
d = {"nome": "Maria", "idade": 35}
print(d["nome"])  # Acessando valor
d["idade"] = 36   # Modificando valor
```

4.1.4 Estruturas de Controle: if, else, elif

A tomada de decisão é necessária quando queremos executar um código apenas se uma determinada condição for satisfeita.

Diagrama de Fluxo

Aqui está o fluxo das estruturas condicionais:



Referência: Python Iluminado

Funcionamento Básico

1. O programa avalia a condição do `if` primeiro
2. Se for falsa, avalia a condição do `elif` (se houver)
3. Se todas forem falsas, executa o `else` (se existir)

Exemplo 1: Avaliação de Notas

```
nota = 7.5

if nota >= 9.0:
    print("Excelente! Conceito A")
elif nota >= 7.0:
    print("Bom! Conceito B")
elif nota >= 5.0:
    print("Regular! Conceito C")
```

```
else:
    print("Precisa melhorar! Conceito D")
```

REFERÊNCIAS

Bryan, Jennifer. 2017. «Excuse me, do you have a moment to talk about version control?» *PeerJ Preprints* 5: e3159v2. https://doi.org/10.7287/peerj.preprints.3159v2.

Bryan, Jennifer, e Jim Hester. 2020. *Happy Git and GitHub for the useR*. https://happygitwithr.com/.

Chacon, Scott, e Ben Straub. 2014. *Pro Git*. 2.ª ed. Apress. https://git-scm.com/book/en/v2.

Foundation, TechAmerica. 2012. *Big Data Recommendations*. TechAmerica.

Gandomi, A., e M. Haider. 2015. «Beyond the hype: Big data concepts, methods, and analytics». *International Journal of Information Management* 35 (2): 137–44.