INSTITUTO FEDERAL
Goiás
Câmpus Goiânia

PYTHON: PARADIGMA FUNCIONAL APLICADO À TRANSFORMAÇÃO DE DADOS

Autores: Felipe Crispim[1]
Higor Ferreira Alves Santos[2]
Israel Magalhães[3]
Rodolfo[4]

[1]

[2] Graduado em Engenharia de Computação pela PUC-GO <hfashigor@gmail.com>

[3]

[4] Graduado em Sistemas de Informações pela FIR-PE <rlayme@gmail.com>

RESUMO

Resumo aqui

1 INTRODUÇÃO

2 OBJETIVO

O objetivo deste estudo é escrever código manutenível e de fácil compreensão se utilizando da abordagem funcional afim de demonstrar os efeitos positivos da mesma.

3 METODOLOGIA

Este estudo adotará uma abordagem comparativa entre os paradigmas funcional e imperativo, analisando soluções para problemas comuns em ambas as abordagens. A metodologia consistirá em:

1. Seleção de Problemas: Escolha de um problema representativo, computacionalmente resolvível.
2. Implementação Comparada: Desenvolvimento das soluções em na abordagem funcional e imperativa
3. Análise Quantitativa: Avaliação de aspectos como:
   (a) Quantidade de linhas de código
4. Análise Qualitativa: Avaliação de aspectos como:
   (a) Facilidade de compreensão
   (b) Legibilidade (mediante revisão por pares)
   (c) Manutenibilidade (tempo para introduzir modificações)
   (d) Clareza (análise de estrutura e redundância)
5. Ferramentas: Todo o trabalho será desenvolvido utilizando a linguagem python e o ambiente de *notebooks* do *jupyter*[1]

4 FUNDAMENTOS TEÓRICOS

O Paradigma de Programação Funcional (abreviado do inglês como **FP**, ***Functional Programming***), é um modo de pensar a resolução dos problemas computáveis a partir da composição de funções[3].
Linguagens que oferecem total suporte a este paradigma geralmente permitem que funcções sejam atribuídas a variáveis, passadas como parâmetros e retornadas de outras funções.

4.1 Funções puras

Pode-se dizer que uma função é pura quando retorna sempre a mesma saída, dados os mesmos argumentos[3]. Em resumo, as funções puras não podem alterar nenhum estado externo ao seu escopo[2].
Exemplo: Digamos que queira-se implementar um contador para uso geral conforme código abaixo. Os incrementos de valores sempre são dados através da função **incrementa** definida na linha 3.

```
1 contador = 0
2
3 def incrementa(valor):
4     contador += valor
5     return contador
6
7 print(incrementa(5))
8 # Saída: 5 (esperado)
9 print(incrementa(2))
10 # Saída: 8 (esperado)
```

Código 1: Exemplo de função impura

Observa-se nas linhas 7 e 9 que as chamadas à função **incrementa** retorna os resultados esperados.
Agora, digamos que em algum ponto do programa a variável **contador** seja alterada conforme o Código 2:

[1] Ambiente web interativo para criar documentos com código, texto (Markdown), fórmulas e gráficos. Usa formato JSON (.ipynb) organizado em células de entrada/saída[5].

[2] Contexto onde variáveis/expressões são acessíveis. Escopos filhos herdam dos pais, mas não o contrário. Funções criam escopos isolados (ex.: variáveis internas não são acessíveis externamente)[1].

```
12 contador = 100
13 print(incrementa(5))
14 # Saída: 105 (inesperado)
```

Código 2: Retorno inesperado de função impura

Percebe-se na saída da linha 13, que a chamada da função **incrementa** com parâmetro 5 retorna resultados imprevisíveis. Ao comparar a linha 13 com a linha 7 do Código 1, nota-se claramente a quebra do conceito de função pura. Isto introduz comportamentos inconssistentes que podem levar a *bugs*[3] inesperados no código. Bugs desta natureza geralmente são difíceis de rastrear, a situação é ainda mais sensível em códigos de natureza ***Multi-Threading***[4].
Em resumo: Qualquer função que altere estados fora de seu escopo é impura por natureza, pois a alteração de estados globais geram escopos compartilhados em que um mesmo recurso pode ser concorrentemente disputado. É este tipo de abordagem que leva a problemas de deadlock[5] e os mais derivados problemas em computação concorrente e parelela. Desta forma o uso de semáforos, filas e etc acabam sendo obrigatórios, adicionando grandes complexidades ao código fonte.

4.1.1 Purificando a função

O Código 3 demonstra a versão purificada da função **incrementa** na linha 3:

```
1 contador = 0
2
3 def incrementa(original, valor):
4     return original + valor
5
6
7 print(contador:=incrementa(contador, 5))
8 # Saída: 5 (esperado)
9 print(contador:=incrementa(contador, 2))
10 # Saída: 8 (esperado)
11
12 contador = 100
13 print(contador:=incrementa(contador, 5))
14 # Saída: 105 (esperado)
```

Código 3: Função purificada

Nota-se que as saídas nas linhas 7, 9 e 13 são as mesmas da versão anterior do código. A diferença é que agora a função **incrementa** não altera nenhuma variável/estado externo ao seu escopo. Sempre que **incrementa** é chamada, a variável **contador** é atualizada com o novo valor do retorno da função. Observa-se agora que na linha 13 a saída 105 é esperada, pois o parâmetro não depente mais somente do valor 5 a ser incrementado, mas também do valor original de **contador**. Desta forma, a função torna-se pura por não alterar escopos externos a ela. De certa forma, pode-se dizer que a função está a trabalhar com o conceito de **imutabilidade** dentro dos limites de seu escopo (embora **contador** seja reatribuído diversas vezes).

4.2 Imutabilidade

A imutabilidade se refere à impossibilidade de mudança de estado de um objeto/varíavel no programa. Em outras palavras: "Dizemos que uma variável é imutável, se após um valor ser vinculado a essa variável, a linguagem não permitir que lhe seja associado outro valor." (QUEIROZ, 2024, p.25). Segundo QUEIROZ (2024) a imutabilidade tem grande importância para que máquinas que se comunicam entre si em um ambiente de programação distribuída não tenham que lidar com estados inconsistentes. Ainda, segundo Gonçalves (2022) não é necessário sincronizar o acesso ao código quando a imutabilidade está aplicada, pois leituras concorrentes são inofensivas, tornando este cenário ideal para o uso de multi-threading sem que hajam os efeitos adversos do acesso sumltâneo a recursos compaartilhados (mutáveis).

5 PRÁTICA

5.1 List Comprehensions

As compreensões de lista (*list comprehensions*) são uma forma prática e expressiva de criar listas em Python a partir de sequências iteráveis. Com uma única linha de código, é possível aplicar transformações e filtros aos elementos, de maneira mais clara e concisa do que com laços tradicionais. Elas se alinham aos princípios do paradigma funcional por evitarem efeitos colaterais, privilegiarem a imutabilidade e expressarem a transformação de dados de forma declarativa.

5.1.1 Estrutura Sintática

A forma geral de uma compreensão de lista é:

```
1 [expressao for item in iteravel if condicao]
```

Código 4: Forma geral de uma list comprehension

Cada parte da expressão representa:

- **expressao**: a transformação aplicada a cada item
- **item in iteravel**: a iteração sobre os dados
- **if condicao**: (opcional) aplica um filtro

5.1.2 Comparativo com o Paradigma Imperativo

Imperativo:

Funcional com list comprehension:

[3] Erro em sistemas eletrônicos/software que causa comportamentos inesperados (ex.: travamentos, resultados incorretos). Pode surgir no código-fonte, frameworks, SO ou compiladores. Comum em computação, jogos e cibernética[6].

[4] Paradigma que aumenta a eficiência do sistema ao executar múltiplas threads/tarefas simultaneamente, assim como o multiprocessamento. Essencial para a computação moderna[8] (adaptado).

[5] Impasse em que processos ficam bloqueados mutuamente, cada um esperando por um recurso retido por outro. Comum em SOs e bancos de dados, ocorre mesmo com recursos não-preemptíveis (ex.: dispositivos, memória), independente da quantidade disponível. Pode envolver threads em um único processo[7].

```python
resultado = []
for x in range(10):
    if x % 2 == 0:
        resultado.append(x * x)
```

Código 5: List comprehension - Sequencial/imperativa

```python
resultado = [x * x for x in range(10) if x % 2 == 0]
```

Código 6: List comprehension - Forma funcional

A versão funcional torna o código mais claro e expressivo.

5.1.3 Integração com Funções de Alta Ordem[6]

List comprehensions podem substituir combinações de `map()` e `filter()`, mantendo a clareza:

```python
# map + filter
list(map(lambda x: x*x, filter(lambda x: x % 2 == 0, range(10))))

# list comprehension
[x*x for x in range(10) if x % 2 == 0]
```

Código 7: List comprehension - Comparação com Funções de Alta Ordem

5.1.4 Casos Avançados e Boas Práticas

- **Aninhamento**:

```python
[[i*j for j in range(1, 4)] for i in range(1, 4)]
```

- **Condicional ternário**:

```python
['par' if x % 2 == 0 else 'impar' for x in range(5)]
```

- **Múltiplos `for`**:

```python
[(x, y) for x in [1, 2] for y in [10, 20]]
```

5.1.5 Aplicação em Transformação de Dados

Um exemplo clássico na análise de dados:

```python
import csv

with open("dados.csv") as f:
    reader = csv.DictReader(f)
    dados = [int(row["idade"]) for row in reader if row["ativo"] == "sim"]
```

Código 8: List comprehension - Aplicação em Transformação de Dados

Esse exemplo mostra uma transformação funcional e imutável sobre os dados lidos de um CSV.

[6] Veja o capítulo 5.2

5.1.6 Considerações Finais

As compreensões de lista são bastante usadas em Python por facilitarem a escrita de um código mais claro e direto. Elas contribuem para um estilo mais funcional e ajudam a manter o código legível, conciso e fácil de testar. Por isso, são uma boa escolha em tarefas de transformação de dados, especialmente quando queremos montar pipelines simples e objetivos. Quando usadas com bom senso, tornam o código mais fácil de entender e manter no dia a dia.

5.2 Funções de Ordem Superior

Funções de ordem superior são funções que recebem outras funções como argumento, retornam funções como resultado, ou ambas as coisas. Essa característica é essencial no paradigma funcional, promovendo a composição e reutilização de lógica de forma declarativa e sem efeitos colaterais. Em Python, esse conceito é suportado nativamente por meio de funções como `map`, `filter`, `reduce` e também por funções definidas pelo usuário.

5.2.1 Definição e Conceito

Uma função de ordem superior é qualquer função que manipula outras funções como dados. Isso permite encapsular comportamentos, criar pipelines de transformação e escrever código mais expressivo.

1. Seleção do Problema

Deseja-se resolver o seguinte problema computacional com base na lista de trabalho `lista = [ 5, 12, 14, 0, 1, 2, 24, 49, 40, 3, 7 ]`:

- Ordenar a lista de números
- Filtrar apenas os números pares da lista ordenada
- Calcular o quadrado de cada número filtrado

2. Implementação Comparada

Abordagem Imperativa:

```python
lista = [ 5, 12, 14, 0, 1, 2, 24, 49, 40, 3, 7 ]
ordenada = sorted(lista)
resultado = []
for numero in ordenada:
    if numero % 2 == 0:
        resultado.append(numero**2)
```

Código 9: Solução imperativa

Abordagem Funcional com Funções de Ordem Superior:

```python
lista = [ 5, 12, 14, 0, 1, 2, 24, 49, 40, 3, 7 ]
resultado = list(map(lambda x: x**2, filter(lambda x: x
```

Código 10: Solução funcional com `map()`, `filter()` e `sorted()`

3. Análise Quantitativa

A

- **Imperativa**: 4 linhas de código (sem contar a inicialização da lista)
- **Funcional**: 1 linha de código (sem contar a inicialização da lista)

4. Análise Qualitativa

5.3 Composição de Funções

A composição de funções é um conceito no paradigma de programação funcional que permite combinar funções simples para criar outras mais complexas. Essa técnica é essencial para a construção de programas modulares, reutilizáveis e fáceis de testar.

5.3.1 Definição e Conceito

A composição de funções é como montar um quebra-cabeça, onde cada peça (função) se encaixa na próxima para formar uma agregação maior para apresentação do resultado. Em termos matemáticos, a composição de funções f e g resulta em uma nova função $h(x) = g(f(x))$, onde a saída de f se torna a entrada de g.

5.3.2 Exemplo Prático: Transformação de Dados

Deseja-se resolver o seguinte problema computacional com base na lista de trabalho:

```
lista = [5, 12, 14, 0, 1, 2, 24, 49, 40, 3, 7]
```

As etapas da transformação são:

- Ordenar a lista de números.
- Filtrar apenas os números pares da lista ordenada.
- Calcular o quadrado de cada número filtrado.

5.3.3 Comparativo: Imperativo vs. Funcional

Abordagem Imperativa:

Transformação de dados de forma imperativa

```
# Funcao Ordena Lista
def ordenaLista(lista):
  return sorted(lista)

# Funcao Filtra Lista
def filtraLista(lista):
  resultado_filtra = []
  for numero in lista:
    if numero % 2 == 0:
      resultado_filtra.append(numero)
  return resultado_filtra

# Funcao Calcula Lista
def calculaLista(lista):
  resultado_calcula = []
  for numero in lista:
    resultado_calcula.append(numero**2)
  return resultado_calcula

# Estrutura Principal
resultado_lista_ordenada = ordenaLista(lista)
resultado_lista_filtrada = filtraLista(resultado_lista_ordenada)
resultado_lista_calculada = calculaLista(resultado_lista_filtrada)
```

Abordagem Funcional com Composição: Transformação de dados com composição funcional

```
# Abordagem 1: Composição explícita (aninhada)
resultado_aninhado = calculaLista(filtraLista(ordenaList

# Abordagem 2: Pipeline funcional com map e filter
resultado_pipeline = list(map(lambda x: x**2,
    filter(lambda x: x % 2 == 0, sorted(lista))))
```

A versão funcional expressa o fluxo de dados de forma mais direta e declarativa.

5.3.4 Análise dos Resultados

Quantitativa

- **Imperativa:** 10 linhas de código (sem contar a inicialização, apenas as 3 funções de apoio).
- **Funcional:** 1 linha de código (sem contar a inicialização, para a versão com `map/filter`).

Qualitativa

- **Reutilização:** Funções individuais (`ordena`, `filtra`, `calcula`) podem ser reutilizadas em diferentes composições, aumentando a modularidade.
- **Testabilidade:** Funções puras, sem efeitos colaterais, são fáceis de testar individualmente.
- **Clareza:** A composição de funções pode tornar o código mais legível, pois cada função tem uma responsabilidade específica e o fluxo de dados é explícito.
- **Manutenção:** Funções pequenas e bem definidas são mais fáceis de manter e modificar.

5.3.5 Considerações Finais

A programação funcional facilita a programação modular, já que permite construir funções complexas a partir de funções simples e programas a partir da composição de outros programas. Uma função é uma regra de correspondência, como uma função matemática que mapeia membros de um conjunto domínio para um conjunto imagem (PELLEGRINI, 2013).

Uma linguagem funcional oferece: 1) um conjunto de funções primitivas; 2) um conjunto de formas funcionais para construir funções complexas; 3) uma operação de aplicação das funções; e 4) estruturas para representar dados (PELLEGRINI, 2013).

Assim, a composição de funções é uma técnica poderosa na programação funcional, permitindo a construção de código modular, reutilizável e fácil de entender e manter, seguindo os princípios de funções puras e imutabilidade.

5.4 Pipelines de Transformação

Pipelines de transformação de dados são sequências de etapas que automatizam o processo de preparação e transformação de dados, desde a coleta até a análise. Eles ajudam a garantir que os dados sejam processados de maneira consistente e eficiente.

Imagine uma empresa que coleta dados de vendas de várias fontes diferentes, sistemas de gerenciamento de

clientes e redes sociais. Com o crescente volume dessas informações, o tratamento dos dados torna-se exaustivo, gastando horas limpando, organizando e analisando esses dados manualmente. Muitas vezes, erros passam despercebidos, resultando em decisões baseadas em informações imprecisas.

É nesse cenário caótico que os pipelines de transformação de dados se tornam essenciais. Eles automatizam todo o processo, desde a coleta até a análise, garantindo que os dados sejam consistentes, confiáveis e prontos para serem usados na tomada de decisões estratégicas. Com um pipeline eficiente, as empresas conseguem transformar dados brutos em insights valiosos em questão de minutos, não horas.

5.4.1 Definição e Conceito

No paradigma funcional, pipelines de transformação referem-se à aplicação sequencial de funções sobre dados, onde a saída de uma função se torna a entrada da próxima. Essa abordagem, conhecida como composição funcional, permite criar transformações complexas de forma modular e reutilizável, evitando efeitos colaterais e tornando o código mais fácil de entender e manter, especialmente em pipelines de dados ETL (Extração, Transformação, Carregamento).

5.4.2 Por que usar Pipelines?

- **Automatização:** Pipelines permitem que o processo de transformação de dados seja automatizado, reduzindo a necessidade de intervenção manual e minimizando erros.
- **Reprodutibilidade:** Uma vez configurado, o pipeline pode ser executado repetidamente com diferentes conjuntos de dados, garantindo que o processo seja consistente.
- **Eficiência:** Pipelines ajudam a economizar tempo, permitindo que as equipes se concentrem na análise e interpretação dos dados, em vez de se perderem em tarefas manuais.
- **Escalabilidade:** Com o aumento do volume de dados, pipelines podem ser escalados para lidar com grandes quantidades de informações sem comprometer a performance.

5.4.3 Características das Pipelines Funcionais

No contexto de Programação Funcional, os pipelines de transformação apresentam as seguintes características:

- **Funções Puras:** As funções devem ser puras, ou seja, para a mesma entrada, a saída é sempre a mesma, e não há efeitos colaterais (como modificar variáveis externas).
- **Funções como blocos:** Em vez de manipular dados diretamente com laços e condicionais, o paradigma funcional usa funções como blocos de construção. Cada função realiza uma transformação específica.
- **Composição de Funções:** As funções são "encadeadas" de forma que a saída de uma alimenta a entrada da próxima. Isso cria um fluxo de dados através de uma série de transformações.
- **Imutabilidade:** Dados em um pipeline funcional são tipicamente imutáveis. Cada função cria uma nova versão modificada dos dados, preservando a integridade dos dados originais.
- **Paralelização:** A imutabilidade e a natureza pura das funções facilitam a paralelização das operações, permitindo que diferentes etapas sejam executadas simultaneamente.
- **Padrões de projeto:** Funções de ordem superior, como `map`, `filter` e `reduce`, são frequentemente utilizadas para manipular coleções de dados.

5.4.4 Aplicação Prática

Exemplo 1: Pipeline Genérico

Neste exemplo, dado uma lista de números, implementa-se um pipeline para percorrer a lista e aplicar, para cada elemento, as seguintes funções:

1. Adicionar 1 (`adicionar_um(x)`).
2. Multiplicar por 2 (`multiplicar_por_dois(x)`).

Implementação de um pipeline de funções

```
# Funcao para incrementar um valor "X"
def adicionar_um(x):
  return x + 1

# Funcao para multiplicar um valor "X" por 2
def multiplicar_por_dois(x):
  return x * 2

# Funcao Pipeline que aplica uma lista de funcoes a uma
def pipeline(dados, funcoes):
  resultado = []
  for item in dados:
    processado = item
    for funcao in funcoes:
      processado = funcao(processado)
    resultado.append(processado)
  return resultado

# Dados iniciais
numeros = [1, 2, 3, 4, 5]
# Criando o pipeline com a sequencia de transformacoes
transformacoes = [adicionar_um, multiplicar_por_dois]

# Aplicando o pipeline
resultado_final = pipeline(numeros, transformacoes)

# Resultado: [4, 6, 8, 10, 12]
print(resultado_final)
```

Exemplo 2: Pipeline com Funções Lambda e Filtro

Neste exemplo, um pipeline de transformação e filtragem é aplicado a uma lista de entrada para:

1. Aplicar a função `dobro`.
2. Aplicar a função `somar_dez`.
3. Filtrar os resultados com a função `filtro` (manter apenas valores > 10).

Pipeline com lambda e filter

```
# Funcao de pipeline definida no exemplo anterior
def pipeline(dados, funcoes):
    # ... (mesma implementacao)
    resultado = []
    for item in dados:
        processado = item
        for funcao in funcoes:
            processado = funcao(processado)
        resultado.append(processado)
    return resultado

entrada = [1, 2, 3, 4, 5]
dobro = lambda x: x * 2
somar_dez = lambda x: x + 10
filtro = lambda x: x > 10

# Criando o pipeline de transformacao
transformacoes = [dobro, somar_dez]
resultado_transformado = pipeline(entrada, transformacoes)

# Aplicando o Filtro apos a transformacao
resultado_filtrado = filter(filtro, resultado_transformado)

# Resultado Final: [12, 14, 16, 18, 20]
print(list(resultado_filtrado))
```

5.4.5 Benefícios dos Pipelines Funcionais

- **Reutilização:** Funções podem ser combinadas de diferentes maneiras para criar outras transformações em diferentes pipelines.
- **Testabilidade:** Funções individuais e puras são mais fáceis de testar, pois o resultado é sempre previsível.
- **Clareza:** O fluxo de dados é mais fácil de seguir, pois a lógica é dividida em funções pequenas e independentes.
- **Manutenção:** Modificações em uma parte do pipeline geralmente não afetam outras partes e são mais fáceis de entender.
- **Escalabilidade:** A imutabilidade e a natureza pura facilitam a paralelização e a escalabilidade do pipeline.

5.4.6 Aplicações de Pipelines

- **ETL:** Pipelines são amplamente usados em sistemas ETL (Extração, Transformação e Carga) para processar grandes volumes de dados.
- **Processamento de imagens:** Podem ser usados para aplicar uma série de transformações em imagens, como redimensionamento, correção de cores e filtros.
- **Análise de dados:** São essenciais para o pré-processamento dos dados antes da análise, como limpeza, agregação e normalização.

Considerações Finais

Pipelines de transformação no paradigma de programação funcional oferecem uma abordagem modular, reutilizável e escalável para processar dados, com benefícios significativos em termos de testabilidade, manutenção e desempenho.

6 CASO DE USO: ÁRVORE BINÁRIA DE PESQUISA

O Código 11 abaixo demonstra a instanciação de uma árvore binária de pesquisa, os detalhes da implementação serão discutidos adiante. O objetivo em um primeiro momento é demonstrar a facilidade de usar implementações funcionais sobre a árvore afim de realizar algumas operações comuns.

```
1 from Btree import BinaryTree, Node
2 from toolz import compose, compose_left
3
4 tree = BinaryTree([ 5, 2, 6, 10, 1, 3, 4, 9
  ↪  ])
```

Código 11: Árvore Binária de Pesquisa

Nota-se na linha 4 que a árvore **tree** é criada a partir de uma lista de inteiros. As inserções destes inteiros produzem a árvore representada na Figura 1:

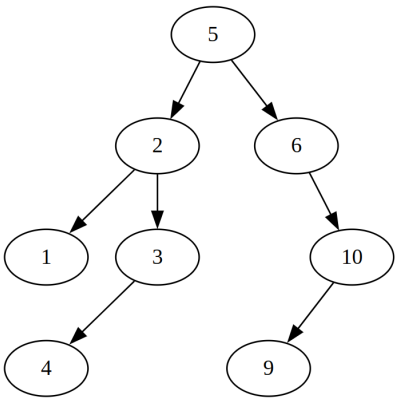

Figura 1: Árvore binária

O objetivo é analisar as chamadas e usos de funções sobre operações comuns desta estrutura de dados, tais como:

1. Percorrimento da árvore em ordem
2. Uso da Busca em Profundidade, do inglês: *Depth First Search* (DFS)[7], para obter informações, como:
   (a) Checar a presença de um determinado nó na árvore
   (b) Listar os nós folha da árvore
   (c) Percorrer determinado nó até a raiz
   (d) Calcular a altura da árvore
   (e) Calcular o balanceamento da árvore

6.1 Usos

6.1.1 Percorrimento em ordem

Conforme mostrado no Código 11, a árvore binária é incializada com uma lista de inteiros totalmente desordenada. Um dos usos mais comuns da Árvore Binária são os percorrimentos em:

- **Pré-Ordem**
- **Em-Ordem**
- **Pós-Ordem**

O percorrimento **Em-Ordem** pode ser utilizado para percorrer todos os nós da árvore de forma que os nós mais à esquerda são visitados primeiro. Dada a natureza de inserção dos itens em uma Árvore Binária, sabe-se que os items cujos valores sejam menores estarão à esquerda, ao passo que os maiores à direita. Desta forma, pode-se imprimir a lista totalmente ordenada conforme mostrado no Côdigo 12:

```
1 tree.inOrderWalk(lambda node:
  ↪  print(node.value, end=" "))
2 # Saída: 1 2 3 4 5 6 9 10
```

Código 12: Percorrimento em ordem

Nota-se no código 12 o nível de abstração quando aplicada a programação funcional, pois a medida que o percorrimento ocorre, cada nó encontrado chama a funçåo lambda[8] passada como segundo parâmetro de **inOrderWalk**. A função então desempenha o papel de imprimir o valor do nó. Observa-se que ao contrário de simplesmente imprimir o nó, poderia fazer-se qualquer outra operação com o mesmo, como empilhá-los para formar uma nova lista ordenada, por exemplo.

6.1.2 Depth First Search (DFS)

A Busca em profundidade (*Depth First Search*) é um algoritmo que também percorre os nós de uma árvore, assim como o percorrimento em ordem. Mas diferente de percorrer em ordem, segundo Wikipédia[4]: A Busca em profundidade é uma busca não-informada que progride se aprofundando tanto quanto possível a partir do primeiro nó filho. Quando a busca chega a um nó folha, retroage-se até o próximo nó e a busca e começa novamente. O Código 13 mostra o percorrimento utilizando DFS.

```
1 tree.dfs(lambda node: print(node.value,
  ↪  end=" "))
2 # Saída: 5 2 1 3 4 6 10 9
```

Código 13: DFS: Percorrimento

Nota-se que o resultado apresentado agora é diferente do percorrimento em ordem apresentado no Código 12. A busca começa a partir do nó raiz e segue até o último nó folha. Evidentemente, os valores impressos não são exibidos de forma ordenada.

6.1.3 DFS: Encontrar um nó

A função passada como argumento para **dfs** (ao contrário das já apresentadas), espera um valor booleano de retorno. Este valor é utilizado para controlar quando parar a busca **dfs**. Caso uma chamada à função passada a **dfs** retorne o valor **True**, a busca é interrompida. Quando **False**, a busca procede. O Código 14 mostra a definição de duas funções nas linhas 1 e 2 para serem utilizadas na **dfs**:

```
1 def retorna_verdadeiro(node: Node):
  ↪  print("Nó visitado:", node); return True
2 def encontre_no(node_number: int):
3     def closure(node: Node): print("Nó
      ↪  visitado:", node); return node.value
      ↪  == node_number
4     return closure
```

Código 14: DFS: Funções de busca

A função **retorna_verdadeiro** apenas exibe o nó visitado e imediatamente retorna **True**. Enquanto que **encontre_no** retorna uma *Closure*[9] que compara o nó atual com um valor passado como argumento. Desta forma, **encontre_no** se utiliza de DFS para procurar um nó na árvore. Conforme se observa no Código 15, apenas o nó raiz é visitado quando se usa **retorna_verdadeiro**. Isto se deve ao fato de que **retorna_verdadeiro** imediatamente retorna o valor **True**. Quando uma função retorna **True**, a **dfs** retorna o nó atual que está sendo visitado:

```
1 tree.dfs(retorna_verdadeiro)
2 # Saída: Nó visitado: Node(5)
3 # Saída:
4 # Saída: Node(5)
```

Código 15: DFS: retorna_verdadeiro

No Código 16 pode-se observar que mais nós são visitados quando se usa **encontre_no**. Ao final, depois de percorrer os nós $(5, 2, 1)$, o nó 1 é encontrado e retornado.

[7] Algoritmo de exploração de grafos que percorre ramificações até o fim antes de retroceder (backtracking). Usado em árvores, grafos e labirintos – estudado desde o século XIX por Trémaux (Wikipédia, 2020)[4].

[8] Explicar a função lambda

[9] Explicar closure

```
tree.dfs(encontre_no(1))
# Saída: Nó visitado: Node(5)
# Saída: Nó visitado: Node(2)
# Saída: Nó visitado: Node(1)
# Saída:
# Saída: Node(1)
```

Código 16: DFS: encontre_no(1)

Finalmente, quando um determinado nó não é encontrado, a árvore inteira é percorrida sem que nada seja retornado:

```
tree.dfs(encontre_no(11))
# Saída: Nó visitado: Node(5)
# Saída: Nó visitado: Node(2)
# Saída: Nó visitado: Node(1)
# Saída: Nó visitado: Node(3)
# Saída: Nó visitado: Node(4)
# Saída: Nó visitado: Node(6)
# Saída: Nó visitado: Node(10)
# Saída: Nó visitado: Node(9)
```

Código 17: DFS: encontre_no(11)

6.1.4 DFS: Encontrar nós folha

O Código 18 demonstra como obter uma lista dos nós folha usando **dfs** a partir da árvore. A natureza do percorrimento **dfs** é ideal para encontrar tais nós. São encontrados 3 nós de acordo com a linha 3:

```
leafs: list[Node] = []
tree.dfs(lambda node: leafs.append(node) if
↪  node.left is None and node.right is None
↪  else False)
leafs
# Saída: [Node(1), Node(4), Node(9)]
```

Código 18: DFS: Nós folha

6.1.5 DFS: Nó folha até a raiz

Uma vez que se tem a lista de nós folha, pode-se utilizar o método **goToRoot** para saber quais nós estão entre a folha e a raiz. O Código 19 mostra que os nós $(4, 3, 2, 5)$ formam o caminho entre o nó 4 (folha), e 5 (raiz).

```
tree.goToRoot(leafs[1], lambda node:
↪  print(node))
# Saída: Node(4)
# Saída: Node(3)
# Saída: Node(2)
# Saída: Node(5)
```

Código 19: DFS: Folha à raiz

6.1.6 DFS: Altura da árvore

O Código 20 demonstra a utilização de composição de funções para calcular a altura da árvore. Nas linhas 1 a 9 são definidas as funções utilitárias. Por fim, descobre-se na linha 15 que a árvore tem altura 4.

```
def pathToRoot(node: Node, tree:
↪  BinaryTree):
    arr: list[Node] = []
    tree.goToRoot(node, lambda n:
    ↪  arr.append(n))
    return arr

def node_paths(lfs: list[Node]): return [
↪  (leaf, pathToRoot(leaf, tree)) for leaf
↪  in lfs ]
def node_length(node: Node, path:
↪  list[Node]): return ( node, len(path) )

tree_heights_pipe = compose_left(
    node_paths,
    lambda arr: [ node_length(node, path)
    ↪  for node, path in arr ],
    lambda arr: [ size for _, size in arr ],
)

compose(max, tree_heights_pipe)(leafs)
# Saída: 4
```

Código 20: DFS: Altura da árvore

6.1.7 DFS: Balanceamento da árvore

Por fim, tendo-se as informações de altura dos nós folha com a ajuda da função **tree__heights__pipe** (Código 20, linha 9), pode-se calcular o balanceamento da árvore conoforme mostra a linha 8 do Código 21:

```
min_height = compose(min,
↪  tree_heights_pipe)(leafs)
diff_height = compose_left(
    lambda arr: tree_heights_pipe(arr),
    lambda arr: [ i - min_height for i in
    ↪  arr ],
    max
)(leafs)

print(f"A árvore está { "des" if diff_height
↪  > 1 else "" }balanceada")
# Saída: A árvore está balanceada
```

Código 21: DFS: Altura da árvore

Fica evidente que a árvore está totalmente balanceada pois não há nós folha com diferença de altura maior que 1.

6.2 Implementação

Conforme visto no Código 11 (linha 1), a árvore mostrada nos exemplos pertence à bilbioreca **Btree** que exporta dois objetos: **BinaryTree** e **Node**. A implementação deste módulo consiste em uma estrutura de pastas com três arquivos conforme mosta a

Figura 2:

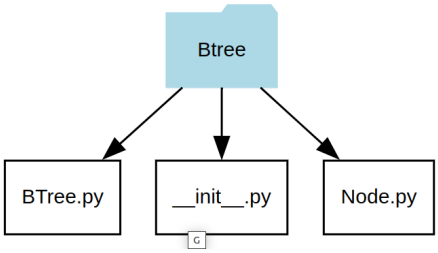

Figura 2: Módulo Btree

6.2.1 Node

A árvore binária é montada com base em seus nós. Um nó é um modelo que representa um dado numérico, contento também as referências para os nós filhos da direita e esquerda, assim como uma referência ao nó pai. O Código 22 demonstra a implementação do nó:

```
1  from typing import Optional
2
3  class Node:
4      value: int
5      parent: Optional['Node']
6      right: Optional['Node']
7      left: Optional['Node']
8
9      def __init__(self, value: int):
10         self.value = value
11         self.left = None
12         self.right = None
13         self.parent = None
14
15     def __repr__(self): return
       ↪  f'Node({self.value})'
```

Código 22: Node: Implementação

A linha 15 define o método **___repr___**. Este método retorna um formato de texto que dita como o nó deve ser exibido quando sua instância é referenciada no notebook, ou quando passada para uma função **print**.

6.2.2 BinaryTree

A implementação de BinaryTree é a árvore em si. O método de inserção e demais funcionalidades não serão abordados, mas sua implementação completa pode ser encontrada no Apêndice A. A parte em que se aplica a programação funcional na árvore está presente nos métodos: **inOrderWalk**; **dfs** e **goToRoot**

6.2.3 BinaryTree: inOrderWalk

O percorrimento em ordem é implementado de forma recursiva[10]. Nota-se na linha 29 que o há um argumento recebido pelo método denominado callback e anotado como:

```
Callable[[Node], None] | None
```

[10]Explicar recursividade

Isto seguinifica que callback é uma função que tem um parâmetro do tipo Node e não retorna nada (None). A anotação serve como uma documentação para que o desenvolvedor não se perca ao passar a função, uma vez que as APIS da tipagem aparecem no itellisense. Na linha 33 a função de callback é chamda caso a mesma tenha sido passada por quem a implementará. No caso, sua chamda passa como argumento uma cópia do nó que está sendo visitado. Este mesmo conceito se repete nos demais métodos.

```
28 ...
29     def inOrderWalk(self, callback:
       ↪  Callable[[Node], None] | None =
       ↪  None):
30         def inOrderRecursive(node: Node |
           ↪  None):
31             if node is not None:
32                 inOrderRecursive(node.left)
33                 if callback:
                   ↪  callback(deepcopy(node))
34                 inOrderRecursive(node.right)
35
36         inOrderRecursive(self.root)
37 ...
```

Código 23: BinaryTree: Método inOrderWalk

6.2.4 BinaryTree: dfs

No Código 24, diferente do anterior, a anotação da função de callback diz que a mesma espera o retorno de um valor booleano. Observa-se nas linhas 45, 47 e 48 como o valor de retorno da callback é aproveitado para controlar quando a dfs vai parar:

```
38 ...
39     def dfs(self, callback: Callable[[Node],
       ↪  bool] | None = None) -> Node | None:
40         def _dfs(node: Node):
41             if node is None: return
42             # print(node)
43             if callback:
44                 res = callback(node)
45                 if res:
46                     return node
47             if res:=_dfs(node.left): return
               ↪  res
48             if res:=_dfs(node.right): return
               ↪  res
49
50         return _dfs(self.root)
51 ...
```

Código 24: BinaryTree: Método dfs

6.2.5 BinaryTree: goToRoot

Por fim, no Código 25 **goToRoot** recursivamente percorre os nós a partir do nó dado no primeiro parâmetro,

até que se chegue ao nó raiz. Para cada nó visitado no caminho, a função de callback é chamada:

```
51 ...
52     def goToRoot(self, node: Node, callback:
       ↪  Callable[[Node], None] | None =
       ↪  None):
53         callback(node)
54         if node == self.root: return
55         self.goToRoot(node.parent, callback)
56 ...
```

Código 25: BinaryTree: Método dfs

7 REFERÊNCIAS

References

[1] MDN Web Docs. Escopo, 2024. URL https://pt.wikipedia.org/wiki/Falha_(tecnologia). Acessado em: 22/06/2025.

[2] Marcelo M. Gonçalves. Programação funcional: Teoria e conceitos, 2022. URL https://medium.com/@marcelomg21/programa%C3%A7%C3%A3o-funcional-teoria-e-conceitos-975375cfb010. Acessado em: 22/06/2025.

[3] PHILLIPE CÉSAR GOMES DE QUEIROZ. Conhecendo a programaÇÃo funcional, 2024. URL https://repositorio.ufc.br/bitstream/riufc/78403/3/2024_tcc_pcgqueiroz.pdf. Acessado em: 2025-06-22.

[4] Wikipédia. Busca em profundidade, 2020. URL https://pt.wikipedia.org/wiki/Busca_em_profundidade. Acessado em: 23/06/2025.

[5] Wikipédia. Projeto jupyter, 2024. URL https://pt.wikipedia.org/wiki/Projeto_Jupyter. Acessado em: 22/06/2025.

[6] Wikipédia. Falha (tecnologia), 2025. URL https://developer.mozilla.org/pt-BR/docs/Glossary/Scope. Acessado em: 22/06/2025.

[7] Wikipédia. Deadlock, 2025. URL https://pt.wikipedia.org/wiki/Deadlock. Acessado em: 22/06/2025.

[8] Wikipédia. Multithreading, 2025. URL https://en.wikipedia.org/wiki/Multithreading_(computer_architecture). Acessado em: 22/06/2025.