INSTITUTO FEDERAL
Goiano

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]

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[4].

[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

Persebe-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[5].

[4] Paradigma que aumenta a eficiência do sistema ao executar múltiplas threads/tarefas simultaneamente, assim como o multiprocessamento. Essencial para a computação moderna[7] (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[6].

```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

Table 1: Comparação qualitativa entre abordagens

Critério	Imperativa	Funcional
Facilidade de compreensão	Alta para iniciantes	Requer familiaridade com HOFs
Legibilidade	Alta, pois descreve o "como"	Concisa, pois descreve o "quê"
Manutenibilidade	Modificações são passo a passo	Pode exigir reescrita da expressão
Clareza estrutural	Lógica explícita e detalhada	Expressão enxuta e aninhada

5.3 Composição de Funções

(Rodolfo)

5.4 Pipelines de Transformação

6 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. Projeto jupyter, 2024. URL https://pt.wikipedia.org/wiki/Projeto_Jupyter. Acessado em: 22/06/2025.

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

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

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