

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Tabelas de decisão
Pré-processador em Python

Eduardo Ribeiro Silva de Oliveira

MONOGRAFIA FINAL
MAC 499 — TRABALHO DE
FORMATURA SUPERVISIONADO

Supervisor: Prof Dr Valdemar W. Setzer

São Paulo
2024

O conteúdo deste trabalho é publicado sob a licença CC BY-NC-ND 4.0
(Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License)

Resumo

Eduardo Ribeiro Silva de Oliveira. **Tabelas de decisão: *Pré-processador em Python***.
Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São
Paulo, São Paulo, 2024.

Tabelas de decisão (TDs) são um método compacto e gráfico para especificar condições lógicas, oferecendo vantagens significativas sobre árvores de decisão tradicionais (estruturas aninhadas de 'if...then...else'). Este trabalho explora a aplicação de TDs em Python, proporcionando uma abordagem mais intuitiva, compacta e eficiente para a programação de escolhas lógicas.

O objetivo deste estudo é desenvolver um pré-processador para TDs em Python, utilizando o modelo descrito na dissertação de [NAGAYAMA, 1990]. O pré-processador gera código em Python a partir de TDs embutidas no programa, facilitando uma expressão mais clara e concisa das condições. A tabela que originou o código e o código gerado constam no arquivo resultante, com novo código fonte. Além disso, o projeto enfatiza a importância da auto-documentação do código ao gerar código auto-documentado seguindo o modelo proposto por Valdemar W. Setzer [SETZER, 1988], pois isso facilita a compreensão do que o programa faz, facilita a colaboração, reduz o tempo de desenvolvimento, aumenta a manutenibilidade e confiabilidade do software.

A implementação do pré-processador de TDs foi testada por meio de diversos estudos de caso, incluindo exemplos do mundo real provenientes do estágio do autor em análise de dados em uma empresa que atua no mercado financeiro, gerindo um fundo multi-mercado. Os resultados demonstram o potencial das TDs para simplificar a compreensão do código, tornando-o mais acessível tanto para programadores quanto para não programadores.

Palavras-chave: Tabelas de decisão. Python. Pré-processador. Auto-documentação. Programação.

Abstract

Eduardo Ribeiro Silva de Oliveira. **Title of the document: *a subtitle*.** Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Decision tables (DTs) are a compact and graphical method for specifying logical conditions, offering significant advantages over traditional decision trees (nested ‘if...then...else’ structures). This work explores the application of DTs in Python, providing a more intuitive, compact, and efficient approach to programming logical choices.

The objective of this study is to develop a preprocessor for DTs in Python, using the model described in the master’s thesis by [NAGAYAMA, 1990]. The preprocessor generates Python code from DTs embedded in the program, enabling a clearer and more concise expression of conditions. The table that originated the code and the generated code are included in the resulting file, along with new source code. Additionally, the project emphasizes the importance of self-documenting code by generating self-documented code following the model proposed by Valdemar W. Setzer [SETZER, 1988], as this facilitates understanding of the program’s functionality, enhances collaboration, reduces development time, and improves software maintainability and reliability.

The implementation of the DT preprocessor was tested through various case studies, including real-world examples from the author’s internship in data analysis at a company operating in the financial market, managing a multi-market investment fund. The results demonstrate the potential of DTs to simplify code comprehension, making it more accessible to both programmers and non-programmers.

Keywords: Decision tables. Python. Pre-processor. Self-documentation. Programming.

Lista de abreviaturas

IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo
TD	Tabela de decisão
DT	Decision Table
IPCA	Índice de produtos do consumidor amplo
TCC	Trabalho de conclusão de curso
API	Application programming interface
IBGE	Instituto Brasileiro de Geografia e Estatística
VIP	Very important person

Sumário

Introdução	1
1 Resolução de problemas de forma estruturada: A utilização de TDs para abstrair a complexidade da programação	3
1.1 Enunciado do problema	3
1.1.1 Problema 1: Sistema de recomendação de produtos	3
1.1.2 Problema 2: Decidir se três pontos formam um triângulo	4
1.2 Foco no problema	4
1.3 A proposta das TDs	4
2 Sintaxe das TDs	7
2.1 Definições e sintaxe	8
2.1.1 Conjuntos (sets)	8
2.1.2 Condições (conditions)	9
2.1.3 Ações (actions)	10
2.2 Limitações e detalhes da sintaxe	11
2.3 Conclusão	12
3 Metodologia	13
3.1 Desenvolvimento do pré-processador de TDs	13
3.1.1 Análise de requisitos	13
3.1.2 Estruturação do código	13
3.2 A classe DecisionTable	14
3.2.1 Auto-documentação	14
3.2.2 Níveis de documentação	15
3.2.3 Testes e validação	18
3.2.4 Ferramentas utilizadas	18
3.3 Conclusão	18

4	Considerações	21
4.1	Implementação das TDs em Python	21
4.2	Auto-documentação e manutenção de código	22
4.2.1	Rótulos de auto-documentação	22
4.2.2	Estrutura interna das funções	22
4.3	Limitações e possíveis melhorias	23
4.4	Contribuições do trabalho	24
4.5	Impacto prático	24
4.5.1	Padrão de validação	24
4.6	Interpretação dos conjuntos	25
4.7	Processamento de condições	25
4.8	Execução de Ações	26
4.9	Geração e execução de código	26
4.10	Conclusão final	26
5	Estudo de Caso: Automação de Análise de Inflação com <code>sidra.py</code>	27
5.1	Descrição geral	29
5.2	Funcionalidades principais	30
5.3	Impacto prático	31
5.4	TDs no <code>sidra.py</code>	31
5.5	Código gerado	33
5.6	Conclusão	34
6	Conclusão	35
6.1	Resultados alcançados	35
6.2	Limitações	35
6.3	Conclusão final	36
A	Pseudocódigo do <code>switch-method</code>	37
A.1	Descrição do Funcionamento	37

Bibliografia	39
---------------------	-----------

Introdução

A programação, em sua essência, envolve a criação de programas contendo condições lógicas. Tradicionalmente, essas escolhas lógicas são implementadas por meio de estruturas como ‘if...then...else’, que, quando aninhadas, podem se tornar difíceis de ser programadas, compreendidas e mantidas. Nesse contexto, as TDs emergem como uma alternativa poderosa, oferecendo uma forma compacta, gráfica e organizada de especificar escolhas lógicas [SETZER, 2024].

As TDs deveriam ser amplamente utilizadas em áreas como engenharia de software e processamento de dados administrativos, onde é crucial representar combinações lógicas de maneira clara e eficiente. Elas permitem que os desenvolvedores visualizem de forma direta todas as possíveis condições e ações correspondentes, facilitando tanto a codificação quanto a validação de sistemas complexos.

Segundo [KING, 1967], uma TD é uma ferramenta útil quando as regras para manipulação de dados são mais complexas do que um simples teste com uma só condição. Em muitos casos, práticas usuais recorrem ao uso de fluxogramas para análise e registro de decisões, o que pode resultar em programas com várias ramificações que, embora escritos em uma linguagem de alto nível, podem ser difíceis de entender na ausência do fluxograma original.

Além de melhorar a clareza, as TDs incentivam o desenvolvimento de programas modulares e mais estruturados. Isso ocorre porque as condições lógicas são separadas em área da TD diferente da área com a indicação das ações a serem tomadas para cada combinação das condições. Além disso, refletem conjuntos de comandos logicamente interligados, facilitando a manutenção e evolução do código [NAGAYAMA, 1990].

Conforme aponta [POOCH, 1974], ainda que tenham sido desenvolvidas inicialmente como um veículo de comunicação entre pessoas, as TDs podem ajudar em problemas relacionados à programação e documentação em muitas aplicações, especialmente quando o uso de fluxogramas tradicionais ou descrições narrativas se torna inviável.

Outra perspectiva interessante é apresentada por [MORET, 1982], que descreve árvores de decisão como modelos de avaliação de funções discretas, onde o valor de uma variável é determinado e a próxima ação é escolhida de acordo com essa avaliação. Esse modelo pode ser comparado com o fluxo lógico das TDs, onde cada variável é avaliada e uma ação correspondente é tomada.

Este trabalho tem como objetivo explorar e implementar o uso de TDs na linguagem Python, por meio do desenvolvimento de um pré-processador que converte essas tabelas em código Python compilável ou interpretável. A abordagem é inspirada na dissertação

de mestrado de [NAGAYAMA, 1990], que descreve um sistema similar as linguagens de programação Pascal e C. Ao implementar este pré-processador, espera-se não apenas simplificar a codificação de escolhas lógicas, mas também promover a auto-documentação do código, uma prática que melhora significativamente a manutenção e a compreensão dos programas. Além disso, são dados alguns exemplos práticos de casos de uma empresa, destacando como essa abordagem pode reduzir o tempo de desenvolvimento e facilitar a comunicação entre programadores e outros profissionais.

O orientador deste TCC, V.W. Setzer, usou o pré-processador de Nagayama em disciplinas de compilação para a confecção dos projetos dos alunos. Os alunos começaram a usar TDs até mesmo quando havia uma única condição, pois alegaram que isso documentava melhor os seus programas. Exemplos simples de TDs estão na [Subseção 1.1.1](#) e o código do projeto, que é de domínio público, podem ser acessados na bibliografia (ver [EDUARDO, 2024]).

Capítulo 1

Resolução de problemas de forma estruturada: A utilização de TDs para abstrair a complexidade da programação

1.1 Enunciado do problema

A resolução de problemas com o uso de programação frequentemente envolve uma série de condições lógicas e ações que precisam ser executadas com base nos resultados dessas condições. A seguir, vamos apresentar dois problemas diferentes e como podemos estruturar suas condições e ações utilizando TDs. A estrutura e sintaxe das TDs é explicada no [Capítulo 2](#);

1.1.1 Problema 1: Sistema de recomendação de produtos

Vamos dar um exemplo de TD imaginando um cenário onde temos diferentes categorias de clientes (novos clientes, clientes frequentes e VIPs), e queremos aplicar descontos com base em suas compras anteriores. Além disso, queremos oferecer frete grátis para compras acima de determinado valor. Para resolver esse problema, precisamos identificar quais são as condições e quais são as ações necessárias.

Condições

- O cliente é novo, frequente ou VIP?
- A compra ultrapassa o valor mínimo para frete grátis?

Ações

- Aplicar desconto de 5% para clientes novos.
- Aplicar desconto de 10% para clientes frequentes.

- Aplicar desconto de 20% para clientes VIPs.
- Oferecer frete grátis para compras acima de R\$200,00.
- Enviar um e-mail de boas-vindas para novos clientes.

O cliente é novo, frequente ou VIP?	Novo	Novo	Frequente	Frequente	VIP	VIP
A compra ultrapassa o valor mínimo para frete grátis?	N	Y	N	Y	N	Y
Enviar um e-mail de boas-vindas para novos clientes	1	1	-	-	-	-
Aplicar desconto de 5% para clientes novos	-	2	-	-	-	-
Aplicar desconto de 10% para clientes frequentes	-	-	-	1	-	-
Aplicar desconto de 20% para clientes VIPs	-	-	-	-	1	1
Oferecer frete grátis para compras acima de R\$200,00	2	3	1	2	2	2

Tabela 1.1: TD: Sistema de recomendação de produtos

1.1.2 Problema 2: Decidir se três pontos formam um triângulo

Outro problema que podemos resolver usando TDs envolve determinar se três pontos no plano cartesiano formam um triângulo. Para que três pontos $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$ formem um triângulo, eles não devem estar alinhados.

A condição para que três pontos estejam alinhados é que a área do triângulo formada pelos pontos seja zero. A fórmula da área de um triângulo com vértices $A(x_1, y_1)$, $B(x_2, y_2)$ e $C(x_3, y_3)$ é dada por:

$$\text{Área} = \frac{1}{2} |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$$

Se a área for zero, os pontos estão alinhados, e portanto, não formam um triângulo.

Valor da área	>0	0
Avisa que é um triângulo	1	-
Avisa que não é um triângulo	-	1

Tabela 1.2: TD: Verificar se três pontos formam um triângulo

1.2 Foco no problema

Em ambos os exemplos, o foco principal é a resolução do problema através da definição clara de condições e ações. Observamos que o desafio não está na sintaxe da programação, mas sim em estruturar as condições corretas para apresentar o problema de modo visível, facilmente compreensível. A primeira situação lida com regras de negócio, enquanto a segunda trata de uma condição geométrica matemática.

1.3 A proposta das TDs

Como mostrado nos dois problemas, as TDs oferecem uma forma eficiente de organizar as condições e ações de maneira clara. Elas permitem que possamos focar no que

realmente importa: a lógica do problema, sem nos preocuparmos com detalhes técnicos de implementação. Com as TDs, temos:

- Uma visualização clara e compacta de todas as combinações possíveis de condições e suas respectivas ações. No entanto, caso existam muitos valores, a TD pode ficar extensa demais, fato este que dificulta a leitura da TD.
- Possibilitar testes de mesa para garantir que todas as regras e situações foram cobertas de maneira sistemática.
- Redução da complexidade na programação, pois a TD serve como um modelo claro para a geração de código.
- Em particular, se os conteúdos da primeira coluna forem nomes de funções do programa significativos em relação ao problema, há uma documentação clara do programa.

Capítulo 2

Sintaxe das TDs

A sintaxe da TD segue um formato específico para garantir clareza e funcionalidade. Todas as linhas que fazem parte de uma TD devem começar com `#TD`, para que o pré-processador identifique que se trata de uma TD. Além disso, é possível definir um nome para a TD utilizando a palavra reservada do pré-processador `decision table`. Outras palavras reservadas, como `sets`, `conditions`, `actions`, e `end table`, são utilizadas para delimitar as diferentes partes da TD. Não há necessidade de seguir o alinhamento vertical do código, já que a separação entre as colunas é feita por espaços em branco na mesma linha.

Ao todo existem 3 seções: `sets`, `conditions` e `actions`. `sets` definem rótulos para conjuntos que serão usados nas condições, como será detalhado na [Subseção 2.1.1](#). `conditions` tem duas partes: A coluna mais à esquerda contém variáveis ou funções do programa externo à TD. À direita há várias colunas, separadas por espaços em branco; em uma linha, o valor da variável ou função é comparado com o conteúdo das colunas (que podem ser rótulos de conjuntos), descrito na [Subseção 2.1.2](#). `actions` contém à esquerda ações, comandos e funções a serem executadas se todas as condições de uma coluna forem verdadeiras; a ordem das ações é especificada por meio de números (ver [Subseção 2.1.3](#)). Mais detalhes serão especificados na [Seção 2.1](#).

Também é possível adaptar o pré-processador para outras linguagens de programação. Para isso, seria necessário alterar a marca `"#TD"` para os comentários dessas linguagens e alterar a sintaxe do código gerado. Abaixo está um exemplo de como essa sintaxe é utilizada em um programa em Python. A seguir os vários elementos de um TD são explicados:

```

if __name__ == '__main__':
    print(f"> SIDRA.py is a program connected to a Notion database that can plot graphs of data collected from the IPCA (Índice Nacional de Preços ao Consumidor Amplo) by IBGE.\n")
    # It is recommended to update the database before performing any analysis.
    ibge = HandlerIBGE()
    database = HandlerDatabase()
    updater = HandlerUpdater(ibge, database)
    analysis = HandlerAnalysis(database)
    while True:
        option = int(input("> Please choose an option:\n")
        #TD decision table td_main
        #TD sets
        #TD opcao valida {1,2,3,12,13,123}
        #TD atualiza {1}
        #TD analisa {2}
        #TD sai {3}
        #TD atualiza_analisa {12}
        #TD atualiza_sai {13}
        #TD atualiza_analisa_sai {123}
        #TD conditions
        #TD option opcao_valida opcao_valida opcao_valida opcao_valida opcao_valida opcao_valida
        #TD option atualiza analisa sai atualiza_analisa atualiza_sai atualiza_analisa_sai
        #TD actions
        #TD analysis.visualization_core() 0 1 0 2 0 2
        #TD updater.update_core() 1 0 0 1 1 1
        #TD quit_program() 0 0 1 0 2 3
        #TD end table

```

Figura 2.1: Exemplo de TD em Python

2.1 Definições e sintaxe

Uma TD é composta por três seções: sets, conditions e actions, descritas a seguir.

2.1.1 Conjuntos (sets)

Definição: Conjuntos, ou sets, definem as coleções de valores possíveis para as variáveis que serão analisadas nas condições. No exemplo, o conjunto `opcao_valida` define os valores {1, 2, 3, 12, 13, 123}, que correspondem às opções válidas que o usuário pode selecionar.

Sintaxe: Cada conjunto deve ser significativo em termos do problema, seguido dos valores permitidos, e os valores devem estar entre chaves {}. O nome do conjunto é utilizado nas condições subsequentes para determinar as ações com base nos valores avaliados, além de permitir a documentação dos valores que serão utilizados na TD. A seguir, são apresentadas as principais formas de definir conjuntos na TD:

1. ****Conjuntos enumerados**:** Declarados entre chaves {} contendo os valores possíveis separados por vírgulas. Um exemplo básico é:

```
opcao_valida {1, 2, 3, 12, 13, 123}
```

Um exemplo mais avançado pode conter tanto números quanto strings, símbolos e caracteres especiais:

```
exemplo_complexo {"", Abb, !, #$, 1, 2, 3, '1'}
```

Note que não há distinção entre aspas duplas (") e aspas simples ('), porque são interpretados como marcas de cadeia de caracteres da Python, e que podem ser usadas indistintamente. Além disso, a cadeia "" foi adicionada no exemplo para mostrar como distinguir a cadeia "" do separador (virgula). Esse formato indica que a variável `exemplo_complexo` pode assumir qualquer um dos valores listados, sejam eles caracteres, strings ou números. Nesse exemplo, '1' é uma cadeia de caracteres,

enquanto que 1 é um número, mas ambos são traduzidos como strings de um conjunto na Python. Não foi implementado no pré-processador um tratamento para o caso em que os elementos do conjunto possam ser variáveis do programa externo a TD.

2. ****Operadores relacionais****: Conjuntos também podem ser definidos com base em operadores relacionais, como `>`, `<`, `>=`, `<=`, `<>` seguidos de um valor. Exemplo:

```
limite_inferior > 5
```

Isso indica que a variável `limite_inferior` deve ser maior que 5.

3. ****Igualdade simples****: Conjuntos podem ser definidos utilizando o operador `=`, indicando que o valor do conjunto deve ser igual ao valor especificado. Exemplo:

```
status = 1
```

Isso permite atribuir dar nomes a valores, para fins de documentação.

4. ****Intervalos de valores****: Para definir um intervalo de sequências de inteiros, o operador `..` é usado. Exemplo:

```
dias_validos 1..30
```

Isso significa que `dias_validos` pode assumir qualquer valor no intervalo entre 1 e 30, com 30 inclusive.

Limitação: Todos os valores dentro de um conjunto devem ser únicos e bem definidos. O uso correto de conjuntos é essencial para garantir que a TD funcione conforme o esperado, evitando ambiguidades ou sobreposições que possam comprometer a avaliação das condições.

2.1.2 Condições (conditions)

Definição: Condições são as expressões lógicas que determinam quais ações serão executadas com base nos valores das variáveis ou funções externas à TD. Cada valor da condição avalia uma variável ou função externa em relação aos conjuntos definidos anteriormente nos sets e mapeia essas avaliações para as ações subsequentes.

É necessário que todas as linhas possuam o mesmo número valores da condição. Os argumentos da condição são variáveis ou funções externas que são definidas à esquerda e são seguidas de um número qualquer de valores da condição, isto é, colunas indicando os valores dos argumentos da condição. Se uma célula na intersecção de uma linha com uma coluna deve ser vazia, usa-se um hífen.

Sintaxe: A seção de condições na TD é iniciada pela palavra reservada `conditions`. Na coluna à esquerda está o argumento da condição, em que define-se uma variável ou nome de função. As demais colunas da linha são os valores da condição, em que define-se os valores que ela pode assumir, sendo esses valores aqueles definidos nos sets, valores booleanos Y (verdadeiro) e N (falso) ou desigualdades. A separação entre colunas é feita por espaços, sem necessidade de seguir um alinhamento vertical específico. Abaixo estão

algumas diretrizes para a sintaxe das condições:

1. ****Definição de Variáveis****: Cada argumento da condição deve ter o nome da variável ou função do programa que será avaliada.
2. ****Valores Aceitos****: Após a definição da variável ou função, segue a lista de valores que ela pode assumir, um por coluna. Esses valores podem ser:
 - Valores pertencentes a um conjunto definido previamente nos sets, como no conjunto $A = \{ "", "!", "2", 2, "A", "a", "" \}$. A condição é verdadeira se o valor da variável ou função estiver em A.
 - Valores booleanos, como Y (verdadeiro) ou N (falso), quando se trata de condições lógicas simples. Por exemplo, quando o argumento da condição é uma função que retorna um valor booleano. Cada valor da condição avalia o valor do argumento da condição em relação aos conjuntos definidos anteriormente no sets e mapeia essas avaliações para as ações subsequentes.
 - Desigualdades, como ≥ 1 , < 0 , < 102 , ≤ -1 .
3. ****Separação entre Valores****: Os valores de uma condição são separados por espaços em branco na mesma linha. Não é necessário seguir um alinhamento vertical específico para que a tabela seja corretamente interpretada, porém recomenda-se alinhar verticalmente à esquerda os conteúdos de cada coluna. Cada coluna será mapeada para uma sequência de ações da coluna correspondente na seção de ações descrita abaixo.
4. ****Fim da seção de condições****: A seção de condições deve terminar com a palavra reservada `actions`, que marca o fim das condições e o início da definição de ações que serão executadas com base nas condições avaliadas.

Limitação: A clareza das condições, especialmente o nome de funções ou variáveis, são cruciais para garantir que a TD seja fácil de entender e manter. As condições devem ser bem definidas e consistentes com os valores nos sets ou booleanos, Y ou N, para garantir a correta execução das ações subsequentes. Condições muito complexas ou mal definidas podem comprometer a legibilidade e dificultar a depuração do código.

2.1.3 Ações (actions)

Definição: Ações especificam o que deve ser feito quando todas as condições na coluna correspondente são avaliadas como verdadeiras. Cada linha na seção `actions` da TD define uma ou mais funções ou procedimentos a serem executados com base nas condições avaliadas.

Sintaxe: A seção de ações é delimitada pelas palavras reservadas `actions` e `end table`. Para cada coluna, cada linha define uma ação, que será executada de acordo com a correspondência estabelecida nas condições anteriores. O número em cada célula (0, 1, 2, etc.) indica a ação que deve ser tomada, na sequência desses números:

1. ****Valor 1****: Indica que a ação correspondente deve ser executada.
2. ****Valor 0 ou vazio****: Indica que a ação deve ser ignorada.

3. ****Valores maiores que 1****: Indicam a ordem de execução quando múltiplas ações são mapeadas para as condições, todas verdadeiras, na mesma coluna. Por exemplo, valores 1, 2, 3 indicam que as ações devem ser executadas sequencialmente nessa ordem.

A sintaxe exige que as ações sejam listadas com clareza, e a separação entre as colunas deve ser feita por espaços em branco, sem a necessidade de um alinhamento vertical, que é aconselhável para maior clareza. Abaixo estão mais detalhes sobre a sintaxe das ações:

1. ****Ações múltiplas****: Quando mais de uma ação é associada às condições de uma coluna, os números indicam a ordem em que essas ações serão executadas. Por exemplo, em uma linha de ação, se houver valores, pela ordem, 1, 2, 0 ou vazio, isso significa que a primeira e a segunda ações serão executadas em sequência, enquanto a terceira será ignorada.
2. ****Estrutura clara****: Cada linha da tabela de ações deve seguir a mesma estrutura definida pelas condições, isto é, estarem na coluna correspondente. A ação associada a uma condição pode ser uma ação simples, como atualizar uma variável ou chamar uma função, ou uma sequência de ações, conforme especificado pelos números nas células.
3. ****Delimitação por end table****: A seção de ações é finalizada pela palavra reservada `end table`, que indica o término da TD.

Exemplo:

```
#TD actions
#TD analysis.visualization_core() 1 0 1
#TD updater.update_core()          0 1 2
#TD quit_program()                 0 0 3
#TD end table
```

Neste exemplo, `analysis.visualization_core()` será executada na primeira e terceira condições, `updater.update_core()` será executada na segunda e terceira condições, e `quit_program()` será executada como terceira ação na terceira condição.

Limitação: A ordem de execução das ações é crítica, especialmente quando múltiplas ações são encadeadas. Deve-se garantir que as ações que afetam o estado global do programa, como `quit_program()`, sejam cuidadosamente posicionadas para evitar encerramentos prematuros ou efeitos colaterais indesejados.

2.2 Limitações e detalhes da sintaxe

Ordem de avaliação: A TD é avaliada de cima para baixo, o que significa que a ordem das condições é crucial. Alterações na ordem podem mudar o comportamento do programa.

Clareza e manutenção: Embora a TD simplifique a lógica das condições, ela também impõe a necessidade de uma organização clara e concisa. Tabelas complexas podem se

tornar difíceis de manter se não forem bem documentadas. A escolha de nomes para os rótulos de conjuntos e o conteúdo das condições são essenciais para uma boa documentação.

Flexibilidade vs. rigor: A TD possibilita clareza e flexibilidade na definição de condições e ações, mas exige rigor na implementação para evitar inconsistências. Cada conjunto, condição e ação deve ser cuidadosamente definido para garantir que a tabela funcione conforme o esperado.

Extensibilidade: A adição de novas condições ou ações requer uma revisão completa da tabela para garantir que a nova lógica seja corretamente integrada. Isso pode aumentar a complexidade do código, exigindo mais atenção na manutenção.

2.3 Conclusão

A sintaxe da TD implementada em Python neste trabalho proporciona uma maneira estruturada e eficiente de lidar com lógicas condicionais complexas, substituindo árvores de decisão correspondentes com grande vantagem de clareza e documentação. Por exemplo, se há 3 condições a serem testadas, uma árvore de decisões terá 8 folhas e 15 nós, ao passo que a TD correspondente terá 3 linhas de condições com 8 colunas. Além disso, nos nós de uma árvore de decisões é possível colocar ações, ao passo que na TD as condições são claramente separadas das ações. Ao seguir as definições e sintaxe descritas, é possível criar sistemas mais legíveis e fáceis de manter, embora seja essencial estar ciente das limitações e desafios associados.

As TDs descritas neste trabalho na [Seção 5.4](#) exemplificam como uma abordagem disciplinada pode resultar em um código mais organizado e escalável, facilitando a evolução e manutenção do sistema ao longo do tempo.

Capítulo 3

Metodologia

3.1 Desenvolvimento do pré-processador de TDs

O desenvolvimento do pré-processador de TDs em Python foi dividido em várias etapas, cada uma com um foco específico na estruturação e automatização do processo de conversão de TDs em código executável. A seguir, detalha-se o fluxo metodológico utilizado:

3.1.1 Análise de requisitos

O primeiro passo foi a análise detalhada das necessidades que o pré-processador deve atender. Baseando-se na dissertação de Satoshi Nagayama, foram identificados os seguintes requisitos principais:

- **Leitura de TDs embutidas em programas em Python:** O sistema é capaz de identificar e extrair TDs formatadas de maneira específica.
- **Processamento das TDs:** As TDs extraídas são processadas para identificar condições, ações e conjuntos associados.
- **Geração de código em Python:** A partir das TDs processadas, o sistema gera código em Python correspondente, utilizando diferentes métodos de tradução.
- **Inserção do código gerado:** O código gerado para cada TD é inserido no local do programa onde a TD foi especificada, no código em Python original.

3.1.2 Estruturação do código

Para atender aos requisitos, o desenvolvimento foi estruturado em módulos, cada um responsável por uma parte específica do processo:

- **Módulo `code_reader.py`:** Responsável pela leitura do arquivo em Python original e pela extração das TDs, com foco na validação do caminho do arquivo e na extração precisa das TDs e suas posições no código.

- **Módulo `decision_table.py`:** Focado na interpretação das TDs, encapsula a lógica necessária para representar e manipular as tabelas extraídas, permitindo sua fácil integração nos processos subsequentes.
- **Módulos `condition.py`, `action.py`, `set.py`:** Processam e extraem condições, ações e conjuntos das TDs. Cada módulo interpreta uma parte específica da TD e prepara esses elementos para a geração de código.
- **Módulo `code_generator.py`:** O coração do pré-processador, este módulo traduz as TDs processadas em código Python compilável ou interpretável, com suporte a métodos de tradução de TDS, como `switch method`, fatorações sucessivas, busca exaustiva e programação dinâmica.
- **Módulo `code_inserter.py`:** Após a geração do código, insere o código gerado nos locais apropriados no arquivo original, preservando a estrutura original do arquivo enquanto integra o novo código.
- **Módulo `workflow_controller.py`:** Controla o fluxo completo do pré-processador, desde a leitura das TDs até a inserção do código gerado, garantindo a execução eficiente e coerente do processo.

3.2 A classe `DecisionTable`

A classe `DecisionTable` é o núcleo do sistema que gerencia a TD no pré-processador. Ela organiza e processa os conjuntos, condições e ações, extraindo essas informações de uma TD formatada como *string*. A seguir, explicamos como o código dessa classe é implementado e como interage com os diferentes parsers, e como cada parte do processo é gerenciada:

A função `__init__` é responsável por instanciar e inicializar os componentes da TD, que incluem o nome, os conjuntos (sets), as condições (conditions) e as ações (actions). Isso é feito chamando métodos auxiliares como `set_name`, `set_sets`, `set_conditions` e `set_actions`, que utilizam os *parsers* correspondentes para tratar e armazenar esses dados.

```
DecisionTable(extracted_decision_table, position)
```

Essa função é chamada quando a TD é identificada no código. A partir da *string* passada como parâmetro, ela organiza a TD em atributos que representam as diferentes seções da tabela.

3.2.1 Auto-documentação

Um dos aspectos centrais deste trabalho foi a implementação de uma metodologia de auto-documentação [SETZER, 1988]. Cada função e classe desenvolvida foi acompanhada de comentários detalhados que seguem um padrão pré-estabelecido. Estes comentários incluem informações sobre a finalidade, entradas, saídas, dependências e rotinas chamadas.

3.2.2 Níveis de documentação

A auto-documentação foi organizada em três níveis de abstração. Abaixo, apresentamos as representações gráficas desses níveis:

- **Nível 1:** Documentação mais básica, contendo cabeçalhos de funções e descrições gerais, como por exemplo data, autor, finalidade, versão, entradas e saídas, conforme a figura em [3.1]. As linhas desse nível são especificadas com comentários da linguagem, com os caracteres #1.

Nome do Arquivo
text_handler.py
Documentação
<pre>#1[#1 TITULO: TEXTHANDLER #1 AUTOR: EDUARDO RIBEIRO SILVA DE OLIVEIRA #1 DATA: 07/10/2024 #1 VERSAO: 1 #1 FINALIDADE: REALIZAR EXTRAÇÃO E FILTRAGEM DE TEXTO BRUTO A PARTIR DE CONTEÚDO HTML #1 ENTRADAS: NOME DO DIRETÓRIO (NO CONSTRUTOR), CONTEÚDO HTML (NA FUNÇÃO EXTRACT_RAW_TEXT) #1 SAIDAS: TEXTO LIMPO EM MINÚSCULAS EXTRAÍDO DO CONTEÚDO HTML #1 ROTINAS CHAMADAS: EXTRACT_RAW_TEXT, _FILTER_PORTUGUESE_TEXT #1] #1[#1 ROTINA: __INIT__ #1 FINALIDADE: INICIALIZA A CLASSE TEXTHANDLER E CONFIGURA O DIRETÓRIO #1 ENTRADAS: NOME DO DIRETÓRIO (STRING) #1 DEPENDENCIAS: LOGGINGHANDLER #1 CHAMADO POR: TEXTHANDLER #1 CHAMA: LOGGINGHANDLER.__INIT__ #1] #1[#1 ROTINA: EXTRACT_RAW_TEXT #1 FINALIDADE: EXTRAÍ O TEXTO BRUTO DO CONTEÚDO HTML E O FILTRA PARA REMOVER CARACTERES INDESEJADOS #1 ENTRADAS: CONTEÚDO HTML (STRING) #1 DEPENDENCIAS: BEAUTIFULSOUP, RE #1 CHAMADO POR: USUÁRIO #1 CHAMA: _FILTER_PORTUGUESE_TEXT #1] #1[#1 ROTINA: _FILTER_PORTUGUESE_TEXT #1 FINALIDADE: FILTRA O TEXTO PARA REMOVER CARACTERES QUE NÃO SÃO DO PORTUGUÊS #1 ENTRADAS: TEXTO BRUTO (STRING) #1 DEPENDENCIAS: RE #1 CHAMADO POR: EXTRACT_RAW_TEXT #1 CHAMA: NENHUMA #1]</pre>

Figura 3.1: Exemplo de Auto-Documentação Nível 1

- **Nível 2:** Inclui a documentação de nível 1 e adiciona os algoritmos em pseudo-código, o mais afastado possível da linguagem de programação, e o mais próximo possível do problema.

Nome do Arquivo
text_handler.py
Documentação
<pre>#1[#1 TITULO: TEXTHANDLER #1 AUTOR: EDUARDO RIBEIRO SILVA DE OLIVEIRA #1 DATA: 07/10/2024 #1 VERSAO: 1 #1 FINALIDADE: REALIZAR EXTRAÇÃO E FILTRAGEM DE TEXTO BRUTO A PARTIR DE CONTEÚDO HTML #1 ENTRADAS: NOME DO DIRETÓRIO (NO CONSTRUTOR), CONTEÚDO HTML (NA FUNÇÃO EXTRACT_RAW_TEXT) #1 SAIDAS: TEXTO LIMPO EM MINÚSCULAS EXTRAÍDO DO CONTEÚDO HTML #1 ROTINAS CHAMADAS: EXTRACT_RAW_TEXT, _FILTER_PORTUGUESE_TEXT #1] #1[#1 ROTINA: __INIT__ #1 FINALIDADE: INICIALIZA A CLASSE TEXTHANDLER E CONFIGURA O DIRETÓRIO #1 ENTRADAS: NOME DO DIRETÓRIO (STRING) #1 DEPENDENCIAS: LOGGINGHANDLER #1 CHAMADO POR: TEXTHANDLER #1 CHAMA: LOGGINGHANDLER.__INIT__ #1] #2[#2 PSEUDOCODIGO DE: __init__ #2 CHAMA O CONSTRUTOR DA CLASSE PAI E INICIALIZA O DIRETÓRIO #2 ARMAZENA O NOME DO DIRETÓRIO #2] #1[#1 ROTINA: EXTRACT_RAW_TEXT #1 FINALIDADE: EXTRAI O TEXTO BRUTO DO CONTEÚDO HTML E O FILTRA PARA REMOVER CARACTERES INDESEJADOS #1 ENTRADAS: CONTEÚDO HTML (STRING) #1 DEPENDENCIAS: BEAUTIFULSOUP, RE #1 CHAMADO POR: USUÁRIO #1 CHAMA: _FILTER_PORTUGUESE_TEXT #1] #2[#2 PSEUDOCODIGO DE: extract_raw_text</pre>

Figura 3.2: Exemplo de Auto-Documentação Nível 2

- **Nível 3:** Esse nível é indicado por um comentário em Python sem #1 e #2. Engloba os níveis 1 e 2, além de incluir todos os detalhes do código, com uma documentação detalhada das implementações. Os comentários de nível 3 também incluem linhas normais de comentário da linguagem (# sem especificar #1 e #2).

Nome do Arquivo
decision_table.py
Documentação
<pre>#1[#1 TITULO: DECISION_TABLE.PY #1 AUTOR: EDUARDO RIBEIRO SILVA DE OLIVEIRA #1 DATA: 25/08/2024 #1 VERSAO: 1 #1 FINALIDADE: GERENCIAR E PROCESSAR TABELAS DE DECISAO, INCLUINDO PARSING E VALIDACAO DOS DADOS #1 ENTRADAS: TABELA DE DECISAO EM FORMATO DE STRING #1 SAIDAS: ATRIBUTOS DA TABELA DE DECISAO, INCLUINDO NOME, CONJUNTOS, CONDICOES E ACOES #1 ROTINAS CHAMADAS: SETPARSER, CONDITIONPARSER, ACTIONPARSER #1] from .set import SetParser from .condition import ConditionParser from .action import ActionParser #USO DE PATH RELATIVO class DecisionTable: #2 IGNORE: CONSTANTE QUE REPRESENTA A IGNORANCIA DE UMA CONDICAO IGNORE = '!= None' #2 LISTA DE PALAVRAS-CHAVE QUE DEVEM ESTAR PRESENTES NA TABELA DE DECISAO keywords = ['DECISION TABLE','SETS','CONDITIONS','ACTIONS','END TABLE'] #2 INSTANCIA UM OBJETO DO PARSER DE CONJUNTOS _set_parser = SetParser() #2 INSTANCIA UM OBJETO DO PARSER DE CONDICOES _condition_parser = ConditionParser() #2 INSTANCIA UM OBJETO DO PARSER DE ACOES _action_parser = ActionParser() #1[#1 ROTINA: __init__ #1 FINALIDADE: INICIALIZAR A TABELA DE DECISAO COM OS ATRIBUTOS EXTRAIDOS DA TABELA FORNECIDA #1 ENTRADAS: EXTRACTED_DESION_TABLE (STR), POSITION (LIST) #1 DEPENDENCIAS: DECISION_TABLE.SET.SETPARSER, DECISION_TABLE.CONDITION.CONDITIONPARSER, DECISION_TABLE.ACTION.ACTIONPARSER #1 CHAMADO POR: N/A</pre>

Figura 3.3: Exemplo de Auto-Documentação Nível 3

Note que no exemplo 2 e 3 há um problema: O "#" em Python é um comentário de uma linha, mas existem linhas de auto-documentação sem o "#". Isso ocorre porque são imagens da versão em PDF da auto-documentação do código. Foi necessário limitar o tamanho da página para não diminuir a qualidade da fonte.

A auto-documentação foi projetada para:

- **Facilitar a manutenção do código:** Permitindo que outros desenvolvedores compreendam rapidamente o funcionamento do sistema.

- **Promover a clareza e a transparência:** Ajudando a evitar mal-entendidos e erros durante a evolução do código.
- **Servir como base para futuras melhorias:** A documentação clara permite que novos recursos sejam adicionados sem comprometer a integridade do código existente.
- **Permite que o projeto de um programa seja feito por equipes diferentes, uma para cada nível:** A equipe do nível 2 não deve conversar com a de nível 1, garantindo que a documentação dessa última seja clara e precisa, a menos de solicitações de correções ou melhor especificação dos comentários de nível 1. A equipe de nível 3 não deve conversar com as de níveis 1 e 2. Com isso, garante-se que a documentação seja bem feita.

3.2.3 Testes e validação

Após o desenvolvimento de cada módulo na [Subseção 3.1.2](#), foram realizados testes unitários para garantir que cada parte do sistema funcionasse conforme o esperado. O sistema foi testado utilizando TDs reais extraídas de atividades realizadas durante o estágio do autor em análise de dados. Os testes focaram em:

- Precisão da extração das TDs.
- Correta geração do código em Python a partir das TDs.
- Integridade do código inserido no código fonte original.

3.2.4 Ferramentas utilizadas

O desenvolvimento do pré-processador foi realizado utilizando as seguintes ferramentas:

- **Linguagem de programação Python:** Escolhida pela sua flexibilidade e pela ampla gama de bibliotecas disponíveis e ser uma das linguagens mais populares hoje em dia.
- **Bibliotecas padrão da Python (re, os, etc.):** Utilizadas para manipulação de strings, arquivos e expressões regulares.
- **Editor de código:** Visual Studio Code, para edição e depuração do código.
- **Sistema de controle de versão:** Git, para rastreamento das mudanças no código e colaboração.

3.3 Conclusão

A metodologia adotada permitiu o desenvolvimento de um pré-processador de TDs robusto e eficiente, com um alto grau de auto-documentação. O resultado é uma ferramenta poderosa para desenvolvedores Python que necessitam integrar TDs em seus projetos de forma automatizada, clara e documentada. Além disso, a criação de um extrator de documentação complementa o projeto, permitindo a geração de relatórios detalhados

3.3 | CONCLUSÃO

nos vários níveis de documentação que facilitam a compreensão e o compartilhamento do código.

Capítulo 4

Considerações

4.1 Implementação das TDs em Python

A implementação das TDs em Python, por meio do desenvolvimento de um pré-processador específico, revelou-se uma solução eficaz para automatizar e simplificar a codificação de condições lógicas complexas. As TDs oferecem uma representação clara e concisa das regras de negócio, sendo traduzidas diretamente para código em Python utilizando diferentes abordagens, como o método *switch case* (ver [Seção A.1](#)) usado neste trabalho, e outros métodos, como faturações sucessivas, busca exaustiva e programação dinâmica, enunciados em [NAGAYAMA, 1990].

Embora existam diversas abordagens além do método *switch* (ver [Seção A.1](#)), apenas este foi implementado até o momento. Entretanto, o código foi estruturado utilizando o padrão Strategy [FREEMAN *et al.*, 2004], um padrão de projeto comportamental que permite definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis. Dessa forma, futuras melhorias no pré-processador de TDs poderão incluir a implementação dos outros métodos de tradução de forma modular e extensível.

A implementação das TDs em Python foi estritamente baseada na dissertação de Satoshi Nagayama, "Tabelas de Decisão e Implementação do Gerador I-M-E"(1991), que serviu como a principal referência teórica para o desenvolvimento do pré-processador. A adaptação para Python seguiu de perto o modelo proposto por Nagayama, com as modificações limitadas à troca do símbolo de identificação das TDs. Essa mudança foi necessária devido às diferenças na sintaxe de comentários entre Pascal, usado por Nagayama, e Python. A parte do Gerador I-M-E de aplicações não foi implementada.

O uso da linguagem Python permitiu a criação de um sistema modular e extensível, onde cada parte do pré-processador, desde a leitura das tabelas até a inserção do código gerado, foi encapsulada em módulos separados, favorecendo a manutenção e a evolução futura do projeto.

4.2 Auto-documentação e manutenção de código

Um dos aspectos mais inovadores deste trabalho foi a integração da auto-documentação ao longo de todo o desenvolvimento do pré-processador. Cada função e classe foram acompanhadas de comentários detalhados, seguindo a estrutura padronizada, conforme enunciado em 3.2.2. Essa abordagem não só melhorou a clareza do código, mas também a manutenção do projeto.

A auto-documentação revelou-se para o autor deste trabalho uma prática muito útil para projetos de software de longo prazo, onde a manutenção e a evolução do código são inevitáveis. A capacidade de gerar relatórios de documentação em diferentes níveis (1, 2 e 3), utilizando o `extrator_de_documentacao.py`, permite que a documentação se mantenha sempre atualizada e alinhada com o código, reduzindo significativamente o tempo necessário para compreender e modificar o software.

4.2.1 Rótulos de auto-documentação

A auto-documentação eficaz requer que cada arquivo de código siga um formato padronizado de *headers*, como o exemplo abaixo, que deve estar presente no início de cada arquivo (os trechos entre <...> indicam onde devem ser inseridos os comentários correspondentes):

```
#1[
#1 TITULO: CONDITION.PY
#1 AUTOR: EDUARDO RIBEIRO SILVA DE OLIVEIRA
#1 DATA: <data de última atualização>
#1 VERSAO: <código da versão, eventualmente especificando alterações>
#1 FINALIDADE: <comentário sobre o objetivo do fragmento de código>
#1 ENTRADAS: <parâmetros necessários para o funcionamento>
#1 SAIDAS: <resultado do código>
#1 ROTINAS CHAMADAS: <funções chamadas>
#1]
```

Este padrão fornece todas as informações críticas para entender o propósito do programa completo, suas dependências e a evolução ao longo do tempo. Isso não só facilita o entendimento por novos desenvolvedores, como também torna o processo de manutenção mais ágil.

4.2.2 Estrutura interna das funções

Cada função também segue uma estrutura de auto-documentação. O padrão de comentários é dividido em três níveis. Nível 1, indicado por #1, com informações gerais da função. Nível 2, indicado por #2, com pseudocódigo descrevendo os algoritmos. Nível 3, com marcas de comentário sem #1 e #2, contendo o código propriamente dito e comentários sobre os seus detalhes. Esse nível não é indicado por um rótulo específico. Tudo do programa que não é rotulado com nível 1 ou 2 é de nível 3.

Abaixo está um exemplo de como uma função auto-documentada pode ser escrita

nos três níveis de documentação:

```
#1[
#1 ROTINA: PARSE
#1 FINALIDADE: EXTRAIR E PROCESSAR AS CONDIÇÕES DA TABELA DE DECISÃO FORNECIDA COMO STRING
#1 ENTRADAS: EXTRACTED_DECISION_TABLE (STR) - STRING CONTENDO A TABELA DE DECISÃO EXTRAÍDA
#1 SAÍDAS: UM OBJETO DECISION_TABLE
#1 DEPENDÊNCIAS: RE (MÓDULO DE EXPRESSÕES REGULARES)
#1 CHAMADO POR: N/A
#1 CHAMA: N/A
#1]
#2[
#2 PSEUDOCÓDIGO DE: parse
def parse(self, extracted_decision_table: str) -> list:
#2 TRATA OS TIPOS DE CONDIÇÕES PASSADAS
    conditions = []
#2 PROCURA A SEÇÃO DA TABELA DE DECISÃO QUE CONTEM AS CONDIÇÕES E AÇÕES
    match = re.search(r'(#TD conditions.*?#TD actions)', extracted_decision_table, re.DOTALL)
#2 LEVANTA UMA EXCEÇÃO SE A DEFINIÇÃO DE CONDIÇÕES NÃO FOR ENCONTRADA
    if match is None:
        # EXCEÇÃO DEVE INTERROMPER O FLUXO DO PROGRAMA
        raise ValueError(f' [-] Erro na busca pela definição do condition: {match} e {extracted_decision_table}')
#2 VARRE AS LINHAS DE CONDIÇÕES EXTRAÍDAS E AS ADICIONA A LISTA
    for condition_line in [line.split()[1:] for line in match.group(0).split('\n')][1:-1]:
        # O PRIMEIRO ITEM DA LISTA É O NOME DA CONDIÇÃO, OS DEMAIS SÃO OS VALORES DAS COLUNAS
        conditions.append([condition_line[0], condition_line[1:]])
#2 RETORNA A LISTA DE CONDIÇÕES
    return conditions
#2]
```

Essa estrutura garante que cada parte do código seja documentada e compreendida, permitindo uma fácil manutenção e evolução do software. Além disso, permite extrair apenas os comentários de nível 1, mostrando as funções definidas no programa e suas especificações. No nível 2, além dos cabeçalhos das funções, aparecem, em pseudo-código, os algoritmos usados em cada função, mostrando como ela é processada. No nível 3 podem-se colocar comentários especificando detalhes do código, que não cabem no pseudo-código do nível 2, por exemplo alguma manipulação de bits.

4.3 Limitações e possíveis melhorias

Embora o pré-processador desenvolvido tenha atendido aos objetivos propostos, algumas limitações foram identificadas. A principal limitação está relacionada à complexidade dos métodos de tradução utilizados, especialmente em TDs com um grande número de condições e ações. Nesses casos, o código gerado pode se tornar menos legível, especialmente para desenvolvedores menos experientes. No entanto, a recomendação é que não se faça nenhuma alteração no código gerado para as TDs, e sim que elas sejam modificadas para atender correções ou novos requisitos.

Para mitigar essas limitações, futuras versões do pré-processador podem incluir:

- **Melhorias na interface de usuário:** Desenvolver uma interface gráfica ou um sistema web para facilitar a especificação e visualização das TDs.
- **Otimização do código gerado:** Implementar técnicas de otimização que possam reduzir a redundância e melhorar o desempenho do código em cenários complexos.
- **Integração com outras ferramentas:** Expandir a compatibilidade do pré-

processador para que ele possa ser integrado com outras linguagens de programação além da Python, aumentando sua aplicabilidade em diferentes contextos.

- **Expandir o pré-processador para pré-processar as especificações do gerador de aplicações I-M-E:** Especificar com exemplos, como o que o prof. Setzer cunhou como "Programação super-estruturada" poderia ser usada manualmente, isto é, estruturando-se os programas segundo as estruturas básicas leitura-escolhas-ações, onde a parte de escolhas lógicas são especificadas por TDs.

4.4 Contribuições do trabalho

Este trabalho contribuiu significativamente para a área de engenharia de software ao demonstrar como TDs podem ser efetivamente implementadas e automatizadas em Python. Além disso, a introdução de práticas de auto-documentação no desenvolvimento do pré-processador destaca a importância de uma documentação clara e acessível, não apenas para a manutenção do código, mas também para a sua evolução.

O desenvolvimento do `extrator_de_documentacao.py` é outra contribuição importante, pois permite que a documentação seja gerada automaticamente em diferentes níveis, facilitando a compreensão e o compartilhamento do código. Essa ferramenta é particularmente útil em ambientes colaborativos, onde diferentes níveis de documentação podem ser necessários para diferentes públicos, como desenvolvedores, gerentes de projeto e stakeholders.

4.5 Impacto prático

O impacto prático deste trabalho é evidente na simplificação e automatização do processo de codificação de condições lógicas complexas. O pré-processador desenvolvido pode ser diretamente aplicado em projetos reais, onde a clareza e a eficiência do código são cruciais. Além disso, a abordagem de auto-documentação implementada neste projeto serve como um modelo para outros desenvolvedores e equipes de software, incentivando a adoção de práticas que promovem a manutenção e a longevidade do código.

Estas perspectivas futuras mostram que o trabalho realizado até agora é apenas o começo de um campo promissor dentro da engenharia de software, com potencial para influenciar positivamente a forma como desenvolvedores abordam a automatização de regras de negócio e a documentação de código.

4.5.1 Padrão de validação

A função `set_extracted_decision_table` valida a TD utilizando o método `_is_valid_decision_table`. Este método faz uso de uma lista de palavras-chave, como `DECISION TABLE`, `SETS`, `CONDITIONS`, `ACTIONS`, para garantir que a tabela esteja no formato correto, incluindo a sequência correta das partes de uma TD. A validação é feita utilizando uma busca por expressões regulares (regex), que verificam se cada uma dessas palavras-chave está presente na *string* da TD.

```
re.search(r'(#TD.*?#TD end table)', extracted_decision_table, re.DOTALL)
```

O uso de regex permite que o código localize seções inteiras da TD de maneira eficiente, delimitando as partes para processamento. Após a validação, as demais partes da tabela (nome, condições, conjuntos e ações) são extraídas e armazenadas.

4.6 Interpretação dos conjuntos

A classe `SetParser` processa os conjuntos definidos na TD. Através do método `parse`, a *string* da TD é analisada e os conjuntos são extraídos. A regex principal usada para localizar os conjuntos é:

```
re.search(r'(#TD sets.*?#TD conditions)', extracted_decision_table, re.DOTALL)
```

Após identificar o bloco de conjuntos, a `SetParser` processa diferentes tipos de definições de conjuntos, como:

- Conjuntos delimitados por chaves (`{a, b, c}`), que são convertidos em `sets` da Python.
- Operadores relacionais.
- Igualdade simples (`=`), que é convertida para `==` no código Python.
- Intervalos de valores, como `(1..10)`, que são convertidos para intervalos com `range(1, 10)`.

Esses conjuntos são armazenados na TD e podem ser utilizados durante a avaliação das condições.

4.7 Processamento de condições

A classe `ConditionParser` lida com a extração das condições, utilizando regex para encontrar e processar a seção de condições da tabela:

```
re.search(r'(#TD conditions.*?#TD actions)', extracted_decision_table, re.DOTALL)
```

A *string* correspondente à seção de condições é dividida em linhas, e cada linha é processada para identificar as variáveis que estão sendo avaliadas e os valores ou comparações que essas variáveis podem assumir. As condições são organizadas em uma matriz, onde cada célula representa uma combinação de valores para as variáveis, e estas são mapeadas diretamente para as ações subsequentes.

4.8 Execução de Ações

O `ActionParser` é responsável por processar a seção de ações da TD. Ele utiliza a seguinte regex para identificar a seção de ações:

```
re.search(r'(#TD actions.*?#TD end table)', extracted_decision_table, re.DOTALL)
```

Após identificar o bloco de ações, o parser itera sobre cada linha, mapeando os números associados às ações (0, 1, 2, etc.) para determinar quais ações devem ser executadas. Por exemplo:

- 1: A ação deve ser executada.
- 0 ou vazio: A ação é ignorada.
- Valores maiores que 1: Indicam a ordem de execução para múltiplas ações associadas à mesma condição.

As ações são organizadas de maneira sequencial e armazenadas na TD, prontas para serem executadas quando as condições forem avaliadas como verdadeiras.

4.9 Geração e execução de código

Após o processamento das diferentes seções da TD, o sistema gera código em Python que avalia as condições e executa as ações correspondentes. Cada TD é convertida em uma sequência de comandos em Python, que são executadas em tempo real, dependendo dos valores fornecidos como entrada.

4.10 Conclusão final

A estrutura modular do pré-processador, que divide o processamento da TD entre diferentes parsers (`SetParser`, `ConditionParser`, `ActionParser`), facilitou a manutenção e evolução do projeto. As regex utilizadas garantem uma identificação robusta das diferentes partes da tabela, enquanto as classes permitem uma abstração clara da lógica da TD para o código Python.

Capítulo 5

Estudo de Caso: Automação de Análise de Inflação com `sidra.py`

Os grupos de itens do IPCA (Índice de Preços do Consumidor Amplo), como "Alimentação e Bebidas", "Habitação", "Transportes", entre outros, representam conjuntos de itens de consumo da população brasileira que são medidos periodicamente para calcular a inflação. Esses grupos contêm itens granulares organizados de forma hierárquica, onde os índices podem ter tamanhos diferentes (1, 2, 4 ou 7 dígitos). Quanto maior o índice, maior a granularidade e detalhamento do item. Por exemplo, um índice de 1 dígito representa um grupo amplo (como "Alimentação"), enquanto um índice de 7 dígitos representa um item específico dentro desse grupo (como "Leite condensado"). Essa estrutura permite uma visão detalhada das variações de preços em diferentes níveis de consumo, facilitando a análise de setores específicos da economia. É relevante citar que a cesta de produtos do IPCA abrange famílias com o rendimento entre 1 a 40 salários-mínimos.

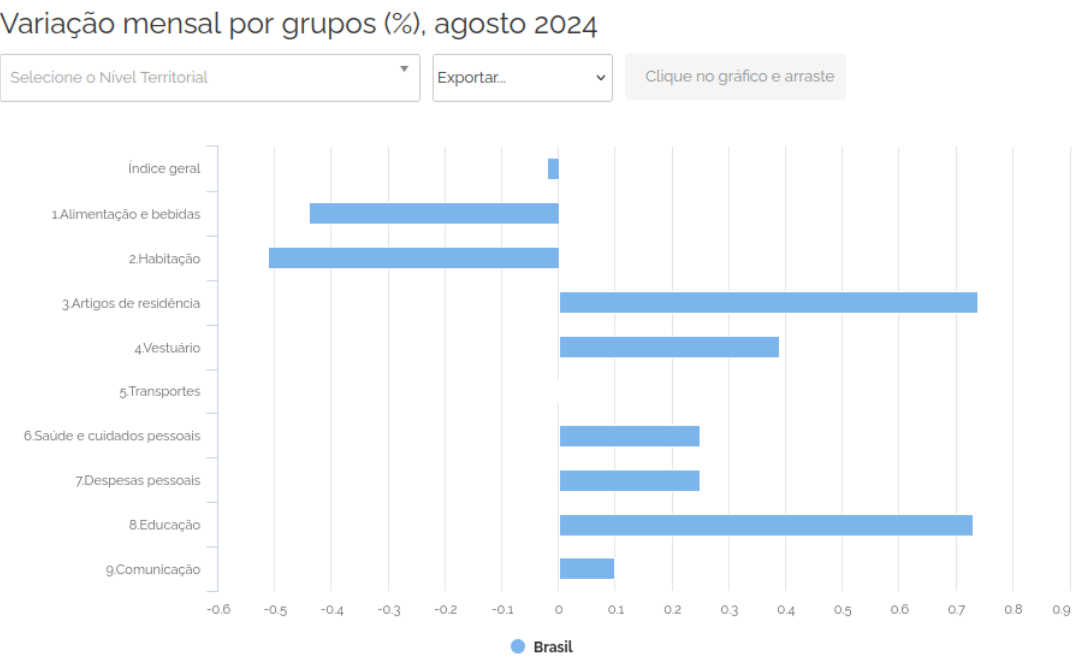


Figura 5.1: Variação mensal por grupos do IPCA, agosto 2024

As aberturas definidas pelo Banco Central para o IPCA, são utilizados para categorizar os produtos e serviços com base em critérios como comercializáveis ou não, duráveis ou semi-duráveis, livres ou monitorados, alimentos in natura ou alimentos industrializados, etc. Essas classificações auxiliam na separação dos produtos por características econômicas e de consumo, permitindo uma análise segmentada das variações de preços na economia. Por exemplo, itens comercializáveis, como o "Frutas", estão mais expostos às variações de mercado global, enquanto itens não comercializáveis, como "Pão Frances", são mais sensíveis a fatores locais.

A distinção entre itens duráveis e semi-duráveis também é relevante. Produtos duráveis, como tapetes, possuem uma vida útil maior, enquanto os semi-duráveis, como roupas, têm durabilidade intermediária. Itens como o "Aparelho ortodôntico" entram na categoria de semi-duráveis, mas envolvem também serviços, como instalação e manutenção, que são classificados como serviços-subjacente.

Além disso, serviços como conselhos de classe, incluídos na categoria de serviços monitorados, têm preços regulados pelo governo, o que limita suas variações e traz previsibilidade.

Essas aberturas fornecem uma visão estruturada e detalhada da economia brasileira, permitindo que economistas e gestores compreendam melhor como os preços variam em diferentes setores. Elas ajudam a monitorar o impacto de políticas econômicas, como o controle da inflação, e a ajustar estratégias de investimento e mercado com base nas variações de preços em diferentes categorias de bens e serviços. Classificações mais amplas, como alimentos in natura e serviços monitorados, permitem uma análise granular e segmentada, facilitando a identificação de setores com maiores pressões inflacionárias e contribuindo para decisões econômicas mais assertivas.

Tabela 5 – Classificações do IPCA a partir de janeiro de 2020: estrutura completa¹

Livres	Calculado por exclusão de monitorados.
Alimentos	Alimentação no domicílio.
Alimentos <i>in natura</i>	Tubérculos, raízes e legumes; Hortaliças e verduras; Frutas; Ovo de galinha.
Alimentos semi-elaborados	Cereais, leguminosas e oleaginosas; Carnes; Pescados; Frango inteiro; Frango em pedaços; Leite longa vida.
Alimentos industrializados	Farinhas, féculas e massas; Açúcares e derivados; Carnes e peixes industrializados; Leite e derivados (exceto Leite longa vida); Panificados; Óleos e gorduras; Bebidas e infusões; Enlatados e conservas; Sal e condimentos.
Serviços	Alimentação fora do domicílio; Aluguel residencial; Condomínio; Mudança; Mão de obra (Reparos); Consertos e manutenção; Passagem aérea; Transporte escolar; Transporte por aplicativo; Seguro voluntário de veículo; Conserto de automóvel; Estacionamento; Pintura de veículo; Aluguel de veículo; Serviços médicos e dentários; Serviços laboratoriais e hospitalares; Serviços pessoais (exceto Cartório e Conselho de classe); Recreação (exceto Jogos de azar, Instrumento musical, Bicicleta, Alimento para animais, Brinquedo e Material de caça e pesca); Cursos regulares; Cursos diversos; Plano de telefonia móvel; TV por assinatura; Acesso à internet; Serviços de streaming; Combo de telefonia, internet e TV por assinatura.
Serviços - Subjacente	Alimentação fora do domicílio; Aluguel residencial; Condomínio; Mudança; Consertos e manutenção; Transporte escolar; Seguro voluntário de veículo; Conserto de automóvel; Estacionamento; Pintura de veículo; Aluguel de veículo; Serviços médicos e dentários; Serviços laboratoriais e hospitalares; Costureira; Manicure; Cabeleireiro e barbeiro; Depilação; Despachante; Serviço bancário; Sobrancelha; Clube; Tratamento de animais (clínica); Casa noturna; Serviço de higiene para animais; Cinema, teatro e concertos.
Serviços - Ex-Subjacente	Calculado pela exclusão do grupo Serviços ex-subjacente.
Duráveis	Mobiliário; Artigos de iluminação; Tapete; Refrigerador; Ar-condicionado; Máquina de lavar roupa; Fogão; Chuveiro elétrico; Televisor; Aparelho de som; Computador pessoal; Joias e bijuterias; Automóvel novo; Automóvel usado; Motocicleta; Óculos de grau; Instrumento musical; Bicicleta e Aparelho telefônico
Semi-duráveis	Cortina; Utensílios de metal; Utensílios de vidro e louça; Utensílios de plástico; Utensílios para bebê; Cama, mesa e banho; Ventilador; Videogame (console); Roupas; Calçados e acessórios; Tecidos e armarinhos; Acessórios e peças (Veículos); Pneu; Brinquedo; Material de caça e pesca; Livro didático; Livro não didático.
Não duráveis	Alimentação no domicílio; Reparos (habitação) exceto mão-de-obra; Artigos de limpeza; Carvão vegetal; Flores naturais; Óleo lubrificante; Etanol; Higiene pessoal; Alimento para animais; Cigarro; Jornal diário; Revista; Caderno; Artigos de papelaria.
Monitorados	Taxa de água e esgoto; Gás de botijão; Gás encanado; Energia elétrica residencial; Ônibus urbano; Táxi; Trem; Ônibus intermunicipal; Ônibus interestadual; Metrô; Integração transporte público; Emplacamento e licença; Multa; Pedágio; Gasolina; Óleo diesel; Gás veicular; Produtos farmacêuticos; Plano de saúde; Cartório; Conselho de classe; Jogos de azar; Correio; Plano de telefonia fixa.
Comercializáveis	Calculado por exclusão de Não comercializáveis e Monitorados
Não comercializáveis	Todos os tipos de feijão; Flocos de milho; Farinha de mandioca; Tubérculos; raízes e legumes; Hortaliças e verduras; Pescados (exceto salmão); Leite e derivados (exceto leite em pó); Pão francês; Pão doce; Bolo; Cimento (Reparos); Tijolo; Areia; Carvão vegetal; Automóvel usado; Alimento para animais; Leitura; Serviços.
Núcleo EX0	Exclui Alimentação no domicílio e Monitorados.
Núcleo EX1	Exclui Cereais, leguminosas e oleaginosas; Tubérculos, raízes e legumes; Açúcares e derivados; Hortaliças e Verduras; Frutas; Carnes; Pescados; Aves e ovos; Leite e derivados; Óleos e gorduras; Combustíveis (domésticos); Combustíveis (veículos).
Núcleo EX2	Exclui Cereais, leguminosas e oleaginosas; Farinhas, féculas e massas; Tubérculos, raízes e legumes; Açúcares e derivados; Hortaliças e Verduras; Frutas; Carnes; Pescados; Aves e ovos; Leite e derivados; Óleos e gorduras; Sal e condimentos; Aparelhos eletroeletrônicos; Automóvel novo; Automóvel usado; Etanol; Fumo; Serviços Ex-Subjacente; Monitorados.
Núcleo EX3	Exclui Alimentação no domicílio; Aparelhos eletroeletrônicos; Automóvel novo; Automóvel usado; Etanol; Fumo; Serviços Ex-subjacente; Monitorados.

^{1/} Sempre que o nível de agregação considerado for superior ao subitem, significa que todos os subitens pertencentes àquela agregação recebem a mesma classificação.

Figura 5.2: Aberturas do Banco Central

O script `sidra.py` (ver [EDUARDO, 2024]) foi desenvolvido pelo autor como parte deste trabalho para demonstrar uma aplicação prática de automação na coleta, processamento e análise de dados econômicos, especificamente a inflação no Brasil, utilizando TDs e integração com APIs externas.

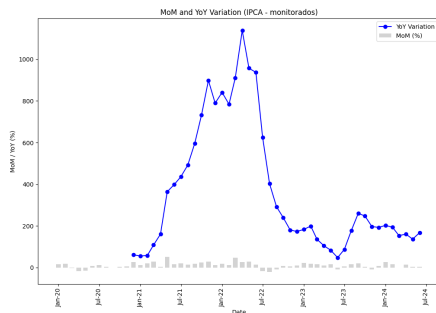
5.1 Descrição geral

O `sidra.py` integra dados da API Sidra do IBGE com uma base de dados no Notion¹), automatizando a atualização e manutenção dessas informações. Além disso, ele realiza cálculos baseados nos dados coletados e gera gráficos que refletem a variação do Índice Nacional de Preços ao Consumidor Amplo (IPCA) e do IPCA-15 ao longo do tempo.

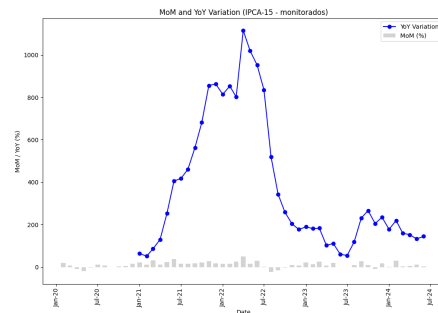
¹ O Notion é uma ferramenta online que não tem o código aberto, que é utilizada para anotações e possui recursos interessantes, como a possibilidade de criar bases de dados (ver [NOTION, 2024])

5.2 Funcionalidades principais

- **Coleta de Dados:** Utilizando a biblioteca `sidrapy`, o script coleta dados relacionados ao IPCA e IPCA-15 diretamente da API do IBGE. A classe `HandlerIBGE` é responsável por esse processo, gerando e formatando os dados para análises subsequentes.
- **Integração com o Notion:** A classe `HandlerDatabase` gerencia a conexão e as operações no banco de dados do Notion. Ela permite a atualização, inserção e sincronização de informações entre os dados obtidos do IBGE e as tabelas armazenadas no Notion, garantindo que a base de dados esteja sempre atualizada.
- **Análise e Visualização de Dados:** Através da classe `HandlerAnalysis`, o script oferece funcionalidades para gerar gráficos de contribuição e variação, permitindo que os usuários visualizem a evolução dos índices ao longo do tempo. Esses gráficos são gerados utilizando o `matplotlib`, proporcionando uma maneira clara e visual de interpretar os dados.
- **Automação e Manutenção:** A classe `HandlerUpdater` coordena a atualização da base de dados do Notion com os dados mais recentes obtidos da API do IBGE. Ela valida e corrige registros, garantindo a integridade e atualidade das informações.



(a) MoM e YoY (IPCA - Monitorados)



(b) MoM e YoY (IPCA-15 - Monitorados)

A partir das imagens, é possível observar a variação tanto mensal (MoM) quanto anual (YoY) dos itens monitorados, tanto no IPCA quanto no IPCA-15. O gráfico da variação do IPCA monitorados mostra um pico significativo entre o início de 2021 e meados de 2022, indicando um aumento expressivo nos preços monitorados durante esse período. Após o pico, há uma queda constante até 2023, com uma estabilização ao longo de 2024.

De forma semelhante, o gráfico do IPCA-15 monitorados apresenta o mesmo padrão de variação, com um aumento acentuado em 2021, seguido por uma queda até 2023. A análise desses gráficos nos permite identificar tendências inflacionárias, como os períodos de alta inflação em itens monitorados, e auxiliar na previsão de políticas econômicas. A comparação entre o IPCA e o IPCA-15 também pode ser útil para analisar a convergência entre os índices e a consistência das variações inflacionárias.

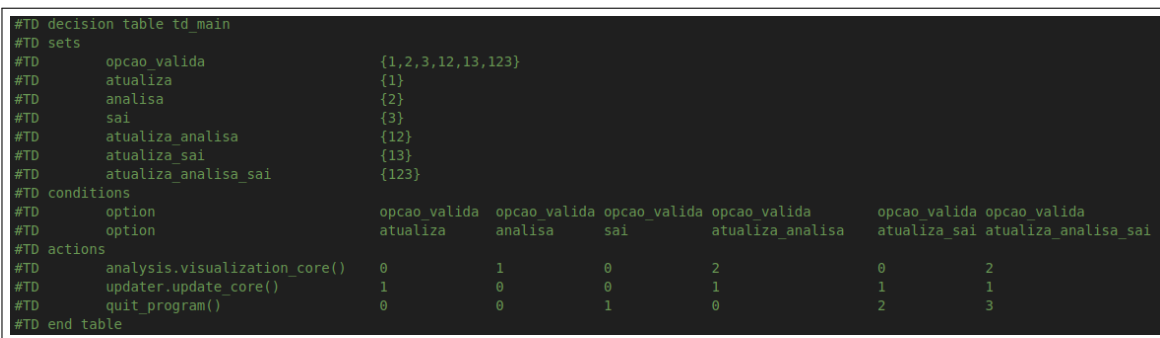
5.3 Impacto prático

O `sidra.py` exemplifica como a automação pode ser aplicada para facilitar a análise de dados complexos, como os relacionados à inflação. A integração com APIs externas e a capacidade de gerar visualizações automáticas tornam o script uma ferramenta poderosa para economistas, analistas de dados e outros profissionais que precisam monitorar indicadores econômicos de forma eficiente e precisa.

Além disso, a modularidade do código e o uso de práticas de auto-documentação asseguram que o sistema seja fácil de manter e expandir, permitindo adaptações futuras conforme as necessidades dos usuários.

5.4 TDs no `sidra.py`

Nesta seção, será discutido o uso das TDs implementadas no `sidra.py` para automatizar a análise de dados econômicos.



```
#TD decision table td_main
#TD sets
#TD      opcao_valida      {1,2,3,12,13,123}
#TD      atualiza           {1}
#TD      analisa           {2}
#TD      sai               {3}
#TD      atualiza_analisa   {12}
#TD      atualiza_sai       {13}
#TD      atualiza_analisa_sai {123}
#TD conditions
#TD      option            opcao_valida opcao_valida opcao_valida opcao_valida opcao_valida opcao_valida
#TD      option            atualiza      analisa      sai          atualiza_analisa atualiza_sai atualiza_analisa_sai
#TD actions
#TD      analysis.visualization_core() 0      1      0      2      0      2
#TD      updater.update_core()         1      0      0      1      1      1
#TD      quit_program()                0      0      1      0      2      3
#TD end table
```

Figura 5.4: TD utilizada para controlar o fluxo do programa

A figura 5.4 apresenta a TD `td_main`, utilizada para gerenciar o fluxo principal do programa `SIDRA.py`. Este programa está conectado a um banco de dados no Notion e realiza análises de dados do Índice Nacional de Preços ao Consumidor Amplo (IPCA), coletados pelo IBGE. A TD define os conjuntos, condições e ações para diferentes opções no menu principal do programa.

Os conjuntos definidos especificam os valores válidos para a variável `option`, representando as escolhas disponíveis no menu. Por exemplo:

- `opcao_valida`: Conjunto contendo todas as opções válidas (`{1, 2, 3, 12, 13, 123}`).
- `atualiza`: Representa a escolha 1, que atualiza o banco de dados.
- `analisa`: Representa a escolha 2, que realiza a análise de dados e cria gráficos.
- `sai`: Representa a escolha 3, que encerra o programa.
- Outros conjuntos, como `atualiza_analisa`, combinam ações.

As condições avaliam se o valor de `option` é válido e pertence a um desses conjuntos. A partir disso, as ações correspondentes são executadas:

- `analysis.visualization_core()`: Invoca a função responsável pela análise.
- `updater.update_core()`: Invoca a função responsável pela atualização do banco de dados.
- `quit_program()`: Invoca a função responsável pelo encerramento do programa.

A abstração de TDs utilizando funções permite que uma TD invoque outra TD de forma modular e transparente, criando um fluxo lógico bem estruturado e altamente reutilizável. Essa capacidade é ilustrada no funcionamento da `td_main`, onde, ao identificar que a opção escolhida pelo usuário requer a execução da ação `analysis.visualization_core()`, ocorre, implicitamente, a invocação de outra TD, a `td_analysis`. Essa arquitetura permite que cada TD seja responsável por uma parte específica da lógica do programa, enquanto mantém a possibilidade de integração com outras TDs. Essa abordagem modular melhora a legibilidade e facilita a manutenção do código, já que cada TD pode ser desenvolvida, testada e ajustada independentemente, enquanto colabora com o restante do sistema de maneira harmoniosa.

Essa abstração é similar ao uso dos comandos `goto` ou `exec` presentes na linguagem PASCAL, que foi utilizada por Satoshi Nagayama para desenvolver o pré-processador I-M-E. Esses comandos permitiam a transferência de controle dentro do fluxo do programa, mas as TDs oferecem uma abordagem mais estruturada e clara para organizar essa lógica complexa, conforme discutido em sua dissertação [NAGAYAMA, 1990].

```
#TD decision table td_analysis
#TD sets
#TD      1      {1}
#TD      2      {2}
#TD      3      {3}
#TD      4      {4}
#TD conditions
#TD      option      1      2      3      4
#TD actions
#TD      self.set_choosed_options()      1      0      0      0
#TD      self.contribution_graph()      2      0      0      0
#TD      self.variation_graph()      0      0      0      0
#TD      self.show_available_options()      0      1      0      0
#TD      self.show_visualization_core_exit_message()      0      0      1      1
#TD      self.show_invalid_option_message(graph_option)      0      0      0      0
#TD      self.show_success_message()      3      0      0      0
#TD end table
```

Figura 5.5: TD utilizada para gerenciar a lógica de visualização de gráficos

A figura 5.5 apresenta a TD `td_analysis`, que gerencia a lógica central para a análise e visualização de gráficos no programa. Esta tabela define os conjuntos, condições e ações associadas às opções de menu apresentadas ao usuário para análise de dados.

Os conjuntos definidos indicam os valores possíveis para a variável `option`, representando as escolhas disponíveis na lista abaixo:

- 1: Escolha para exibir o gráfico de contribuição (gráfico de barras empilhadas por data).
- 2: Escolha para exibir o gráfico de variação (gráfico de linha com barras ao fundo).
- 3: Escolha para exibir as opções disponíveis.
- 4: Escolha para sair do menu de visualização.

As condições avaliam se a variável `option` corresponde a um desses conjuntos. Dependendo do valor, as ações correspondentes são executadas:

- `self.set_choosed_options()`: Define as opções selecionadas para o gráfico de contribuição.
- `self.contribution_graph()`: Gera o gráfico de contribuição com base nas opções escolhidas.
- `self.variation_graph()`: Gera o gráfico de variação.
- `self.show_available_options()`: Exibe as opções disponíveis para o usuário.
- `self.show_visualization_core_exit_message()`: Exibe uma mensagem de saída ao retornar ao menu principal.
- `self.show_invalid_option_message(option)`: Exibe uma mensagem de erro para opções inválidas.
- `self.show_success_message()`: Informa o sucesso na geração e análise do gráfico.

A TD `td_analysis` está embutida dentro de um `while` loop, permitindo uma interação contínua com o usuário. Esse laço de repetição é utilizado para apresentar o menu de opções de visualização de gráficos até que o usuário decida encerrar a execução, escolhendo explicitamente a opção de saída (4). A cada iteração, o programa solicita ao usuário que insira uma escolha através de uma entrada numérica, que é validada contra as condições definidas na TD. Essa abordagem garante flexibilidade e interatividade, permitindo que o usuário execute várias ações sequenciais, como alternar entre diferentes tipos de gráficos ou consultar as opções disponíveis, sem a necessidade de reiniciar o programa. O uso do `while` loop combinado com a lógica clara da TD reforça a modularidade e facilita futuras expansões do sistema.

5.5 Código gerado

Conforme discutido na [Seção A.1](#), o código gerado utiliza a estrutura `match case` para indexar a ação correspondente a cada regra. A seguir, apresentamos os códigos Python gerados a partir da [Figura 5.4](#) e a [Figura 5.5](#).

```
def decision_table_td_main() ->None:
    I_0 = 0 #Inicialização do auxiliar da condição option
    I_1 = 0 #Inicialização do auxiliar da condição option
    I = 0 #Inicialização do número da regra
    if option in set([1,2,3,12,13,123]):
        I_0 = 0
    if option in set([1]):
        I_1 = 0
    elif option in set([2]):
        I_1 = 1
    elif option in set([3]):
        I_1 = 2
    elif option in set([12]):
        I_1 = 3
    elif option in set([13]):
        I_1 = 4
    elif option in set([123]):
        I_1 = 5
    I = (6*I_0 + (1)*I_1
    match I:
        case 0:
            updater.update_core()
        case 1:
            analysis.visualization_core()
        case 2:
            quit_program()
        case 3:
            updater.update_core()
            analysis.visualization_core()
        case 4:
            updater.update_core()
            quit_program()
        case 5:
            updater.update_core()
            analysis.visualization_core()
            quit_program()
        case _:
            exit()
decision_table_td_main()
```

(a) Código gerado para a *td_main*

```
def decision_table_td_analysis() ->None:
    I_0 = 0 #Inicialização do auxiliar da condição option
    I = 0 #Inicialização do número da regra
    if option in set([1]):
        I_0 = 0
    elif option in set([2]):
        I_0 = 1
    elif option in set([3]):
        I_0 = 2
    elif option in set([4]):
        I_0 = 3
    I = (1)*I_0
    match I:
        case 0:
            self.set_chosed_options()
            self.contribution_graph()
            self.show_success_message()
        case 1:
            self.show_available_options()
        case 2:
            self.show_visualization_core_exit_message()
        case 3:
            self.show_visualization_core_exit_message()
        case _:
            exit()
decision_table_td_analysis()
```

(b) Código gerado para a *td_analysis*

5.6 Conclusão

O desenvolvimento do *sidra.py* dentro do contexto deste trabalho demonstra como as TDs e as práticas de auto-documentação podem ser integradas em soluções reais, proporcionando não apenas uma melhoria na eficiência operacional, mas também um aumento na qualidade e acessibilidade dos dados analisados. Este estudo de caso reforça a aplicabilidade prática dos conceitos explorados ao longo deste TCC. O código do projeto é de domínio público (ver [EDUARDO, 2024]).

Capítulo 6

Conclusão

O presente trabalho teve como objetivo desenvolver um pré-processador de TDs em Python, com ênfase na automatização da codificação de condições lógicas complexas e na promoção de práticas de auto-documentação. Ao longo do desenvolvimento, foram alcançados resultados significativos que contribuem para a simplificação do processo de programação e para a melhoria da manutenção de código em projetos de software.

6.1 Resultados alcançados

O pré-processador desenvolvido foi capaz de traduzir TDs em código Python executável, utilizando o método de tradução *switch method* (ver [Seção A.1](#)), mas o pré-processador foi projetado usando o *Strategy Pattern* [FREEMAN *et al.*, 2004], um padrão de projeto comportamental que permite definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis, possibilitando a fácil extensão de novos métodos de tradução, como fatora-ções sucessivas, busca exaustiva e programação dinâmica. Essa ferramenta demonstrou ser eficiente na conversão de regras de negócio complexas em código de forma automatizada, clara e concisa. A modularidade do sistema permitiu uma fácil manutenção e evolução do projeto, garantindo que futuras melhorias possam ser implementadas com facilidade.

A integração de práticas de auto-documentação foi outro marco importante do trabalho. A criação de comentários detalhados em cada função e classe, juntamente com o desenvolvimento do `extrator_de_documentacao.py`, garantiu que a documentação do código fosse sempre clara e atualizada. Isso não apenas facilita a compreensão do sistema por outros desenvolvedores, mas também promove uma manutenção mais eficiente e menos propensa a erros.

6.2 Limitações

Embora o pré-processador desenvolvido tenha alcançado seus objetivos, foram identificadas algumas limitações, especialmente em relação à complexidade e legibilidade do código gerado em casos de TDs com grande número de condições e ações. Essas limitações

abrem caminho para futuras pesquisas focadas em otimização de código e expansão do suporte do pré-processador para outras linguagens de programação.

Outras oportunidades de pesquisa incluem o desenvolvimento de interfaces gráficas para facilitar a criação e visualização de TDs, bem como a investigação de novas técnicas de otimização que possam melhorar ainda mais a performance do código gerado, além de se implementar o gerador de aplicações I-M-E descrito por [NAGAYAMA, 1990].

6.3 Conclusão final

Em síntese, este trabalho não apenas alcançou os objetivos propostos, mas também abriu novas perspectivas para o desenvolvimento de ferramentas de automatização e documentação em projetos de software. A integração de TDs em Python, juntamente com a prática de auto-documentação, oferece uma abordagem eficaz para enfrentar os desafios da programação moderna. Além disso, o pré-processador desenvolvido é de domínio público (ver [EDUARDO, 2024]), permitindo que a comunidade utilize, modifique e contribua com melhorias futuras.

Além disso, este trabalho serviu também para mostrar que focar exclusivamente na solução do problema, sem se preocupar com a forma como um computador processa as informações, é um desafio significativo. As tabelas de decisão mostraram-se uma ferramenta poderosa nesse contexto, pois permitem concentrar-se na essência do problema a ser resolvido, abstraindo os detalhes da implementação. Na graduação em ciência da computação, frequentemente nos concentramos em otimizações e custos computacionais, o que pode nos afastar da perspectiva do problema em si. Isso dificulta a inclusão de pessoas que têm profundo conhecimento sobre o problema, mas que não possuem familiaridade com os detalhes técnicos de programação. Este trabalho reforçou em mim a importância de encontrar soluções que aproximem essas perspectivas, facilitando a colaboração interdisciplinar e promovendo a inclusão no desenvolvimento de software.

Apêndice A

Pseudocódigo do switch-method

O pseudo código abaixo descreve o funcionamento do *switch method*, conforme enunciado na dissertação de [NAGAYAMA, 1990]

A.1 Descrição do Funcionamento

O algoritmo processa uma TD e gera uma estrutura de comandos condicionais *if-elif* para avaliar e indexar corretamente as condições e suas respectivas ações.

- **Iteração sobre as condições:** Para cada linha de condição C_i , um conjunto C é iniciado vazio, e um contador n_i é zerado. Para cada coluna j e linha i , o valor C_{ij} é verificado:
 - Se for um novo valor (ainda não presente no conjunto C), ele é adicionado ao conjunto C e o contador n_i é incrementado.
 - Se for o primeiro valor da linha, um comando `if` é gerado. Caso contrário, um `else if` é produzido.
- **Cálculo do índice I :** A expressão para calcular I é gerada. I é uma variável para indexação da ação a ser executada em cada uma das regras da TD.
- **Mapeamento das regras e ações:** Para cada regra (totalizando $n_1 \times n_2 \times \dots \times n_m$ casos), os índices I_1, I_2, \dots, I_m são determinados.
 - O valor I , indicando cada uma das regras, é associado a ação a ser executada pelo *match case*.

O resultado final é um conjunto estruturado de condições e ações derivadas da TD, permitindo a correta execução das regras definidas.

Algorithm 0: Switch method para tradução de tabelas de decisão

Input: Tabela de decisão com m linhas de condições C_i e n colunas de valores C_{ij}

Output: Código gerado a partir da tabela de decisão

```

1  Início;
2  foreach linha de condição  $C_i$ ,  $1 \leq i \leq m$  do
3       $C \leftarrow \emptyset$ ;
4       $n_i \leftarrow 0$ ;
5      foreach  $C_{ij}$ ,  $1 \leq j \leq n$  do
6          if  $C_{ij} \notin C$  then
7              incrementa  $n_i$ ;
8               $c_{in_i} \leftarrow C_{ij}$ ;
9              if  $C = \emptyset$  then
10                  $\lfloor$  produz comando if  $C_i \dots$  then  $I_i \leftarrow 0$ ;
11                 else
12                      $\lfloor$  produz comando else if  $C_i \dots$  then  $I_i \leftarrow n_i - 1$ ;
13                  $C \leftarrow C \cup \{c_{in_i}\}$ ;
14 grava a expressão que calcula o valor de  $I$ ;
15 foreach  $M$ ,  $1 \leq M \leq n_1 \times n_2 \times \dots \times n_m = n$  do
16     determine  $I_1, I_2, \dots, I_m$  tais que:
           
$$(n_2 \cdot n_3 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot n_4 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_m) \cdot I_{m-1} + I_m + 1 = M$$

           determine a regra  $k$ , tal que  $C_{ik} = c_{i,I_i+1}$ ,  $1 \leq i \leq m$ ;
17      $\lfloor$  grava o comando que executa a ação da regra  $R_k$  através de um match case;
18 Fim;
  
```

Bibliografia

- [EDUARDO 2024] EDUARDO. *Repositório oficial no GitHub - Tabelas de Decisão*. https://github.com/EduardoRSO/tabelas_de_decisao. Acessado em: 9 de Fevereiro de 2025. 2024 (citado nas pgs. 2, 29, 34, 36).
- [FREEMAN *et al.* 2004] Eric FREEMAN, Elisabeth ROBSON, Bert BATES e Kathy SIERRA. *Head First Design Patterns: A Brain-Friendly Guide*. Sebastopol, CA, USA: O'Reilly Media, nov. de 2004 (citado nas pgs. 21, 35).
- [KING 1967] P. J. H. KING. *Decision Tables*. London, UK: Macmillan, 1967 (citado na pg. 1).
- [MORET 1982] Bernard M. E. MORET. "Decision trees and diagrams". *ACM Computing Surveys* 14.4 (1982), pp. 593–623 (citado na pg. 1).
- [NAGAYAMA 1990] Satoshi NAGAYAMA. "Implementação de Gerador I-M-E com Tabelas de Decisão". Tese de dout. São Carlos, SP, Brasil: Instituto de Ciências Matemáticas e de Computação, USP, 1990 (citado nas pgs. iii, v, 1, 2, 21, 32, 36, 37).
- [NOTION 2024] NOTION. *Site oficial do Notion*. <https://www.notion.so/>. Acessado em: 9 de Fevereiro de 2025. 2024 (citado na pg. 29).
- [POOCH 1974] Udo W. POOCH. "Translation of decision tables". In: *Proceedings of the 7th Annual Symposium on Switching and Automata Theory (SWAT '74)*. Washington, DC, USA: IEEE Computer Society, 1974, pp. 212–222 (citado na pg. 1).
- [SETZER 1988] Valdemar W. SETZER. *Um Sistema Simples para Documentação Semi-Automática de Programas*. Rel. técn. RT-MAC-8808. Relatório Técnico. São Carlos, SP, Brasil: Instituto de Matemática e Computação, Universidade de São Paulo (USP), setembro de 1988 (citado nas pgs. iii, v, 14).
- [SETZER 2024] Valdemar W. SETZER. *Comunicação pessoal*. Orientador do Trabalho de Conclusão de Curso. 2024 (citado na pg. 1).