Desenvolvimento de um Sistema de Gerenciamento de Clientes e Pedidos para um Point de Bubble Tea na Linguagem Python

Thiago Marques Reis[1]

[1]Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS) – Feira de Santana – BA – Brasil

theicefrosts@gmail.com

1. Introdução

O principal objetivo desse problema é desenvolver um sistema de gerenciamento de clientes e pedidos para um point de bubble tea que será inaugurado pela UEFS (Universidade Estadual de Feira de Santana) em parceria com a cooperativa MPFSA (Mulheres Poderosas de Feira de Santana). Quanto a motivação para a resolução do problema, pode-se dizer que é muito importante criar um espaço que traga bem-estar e saúde para a comunidade, o que implica diretamente em um sistema que seja simples e eficiente, ou seja, que facilite tanto a vida dos futuros funcionários como dos futuros clientes. Além disso, o incentivo ao empreendedorismo é um detalhe importante, afinal, sem possíveis colaboradores e incentivos, se torna mais difícil empreender em um sistema altamente competitivo e que visa ao máximo o lucro.

No que diz respeito a solução do problema, o projeto consiste em criar um programa baseado nos conceitos de modularização, objetos e persistência de dados, capaz de preservar a identificação dos clientes, armazenar os pedidos, que por sua vez contém uma ampla variedade de tipos de bubble tea, atribuir descontos a tipos de clientes específicos, prover um saldo de cashback para cada cliente utilizar quando quiser, exibir o valor final do pedido e reiniciar o programa.

Em síntese, foram utilizados diversos recursos e conceitos da linguagem Python. Em relação às entradas do programa, pode-se afirmar que as mesmas estão presentes na identificação do cliente, na afirmação do tipo de cliente, na escolha das bases e complementos do pedido, na adição de complementos e na escolha de processar um novo pedido. No viés contrário, ou seja, tratando das saídas, tem-se a impressão das frases de orientação para a continuação do programa, o menu/cardápio e a saída que apresenta toda a descrição do pedido. Os conceitos de modularização, estruturas condicionais, estruturas de repetição, listas, dicionários, bibliotecas, tratamento de dados, objetos, classes e persistência de dados também foram amplamente utilizados no desenvolvimento do projeto.

2. Metodologia

Ao longo da criação do programa, houveram tanto decisões individuais quanto coletivas. Em relação às principais decisões individuais, podem ser citadas o não desenvolvimento do histórico de pedidos, o uso do formato de dados intitulado JSON, o uso de mais

classes em relação ao que foi discutido nas sessões tutoriais, o uso do próprio nome para a identificação dos clientes e o uso de “variáveis constantes” . No que se refere às principais decisões coletivas, pode-se citar o uso constante da modularização e o uso do JSON ou Pickle como formatos de dados.

2.1. Requisitos

Entre os requisitos exigidos para o desenvolvimento do sistema de gerenciamento de clientes e pedidos para o point de bubble tea estão:

- Cada pedido possui apenas um chá base, que pode ser de leite, maracujá, rosa ou manga.
- Cada pedido pode ter quantos complementos o usuário desejar, dentre os que estão disponíveis (boba, lichia, geleia, taro e chia).
- Toda a comunidade poderá frequentar o point, mas estudantes da UEFS deverão ter um desconto de 25% do total do pedido e professores e funcionários da mesma instituição irão ter um desconto de R$1,00, independente do valor final do pedido.
- Todos os clientes possuirão um saldo de cashback, onde em cada pedido 10% do valor final é guardado e poderá ser utilizado em futuras compras.
- Antes do pedido ser finalizado, são exibidos o valor do pedido, o cashback disponível e a opção de usar ou não o cashback.
- Ao final do pedido, o programa exibe uma saída, que dentre outras informações informa o preço do pedido em reais.
- Pode-se pedir apenas um chá de cada vez.
- Após o processamento do pedido, o funcionário deve decidir se haverá um novo pedido ou não.
- As entradas e saídas necessárias devem ser identificadas e implementadas adequadamente.
- O programa deve conter o conceito de objetos, o conceito de persistência de dados e o conceito de modularização.
- No programa devem existir diferentes tipos de dados, que devem ser adequados para cada situação que surgir durante a resolução do problema.

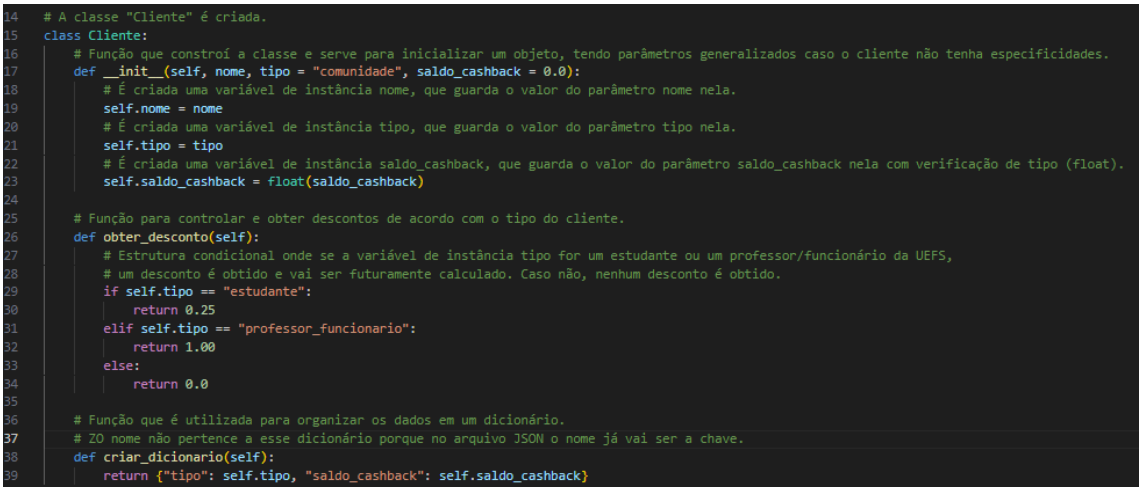
2.2. Descrição do Algoritmo

```python
# Importa a biblioteca JSON, utilizada geralmente para salvar dados em um arquivo (serializar) e carregar dados de um arquivo (desserializar).
import json
# Importa a biblioteca OS, que é utilizada para interagir com o sistema operacional.
# Nesse caso, ela é exclusivamente utilizada para verificar se um arquivo existe no sistema operacional.
import os

# Atribui-se o nome do arquivo onde serão armazenados os clientes a uma variável constante.
NOME_ARQUIVO_CLIENTES = "clientes_point.json"
# Atribui-se um dicionário com as bases e seus respectivos preços a uma variável constante.
PRECOS_BASE = {"Leite": 4.35, "Maracujá": 4.60, "Rosa": 5.85, "Manga": 5.47}
# Atribui-se um dicionário com os complementos e seus respectivos preços a uma variável constante.
PRECOS_COMPLEMENTO = {"Boba": 0.50, "Lichia": 0.75, "Geleia": 0.65, "Taro": 1.00, "Chia": 0.35}
```

Figura 1. Importação de bibliotecas e atribuição de constantes.

Inicialmente, são importadas 2 bibliotecas (json e os). A biblioteca “json” é necessária para salvar e carregar informações de clientes em um arquivo e a “os” permite a

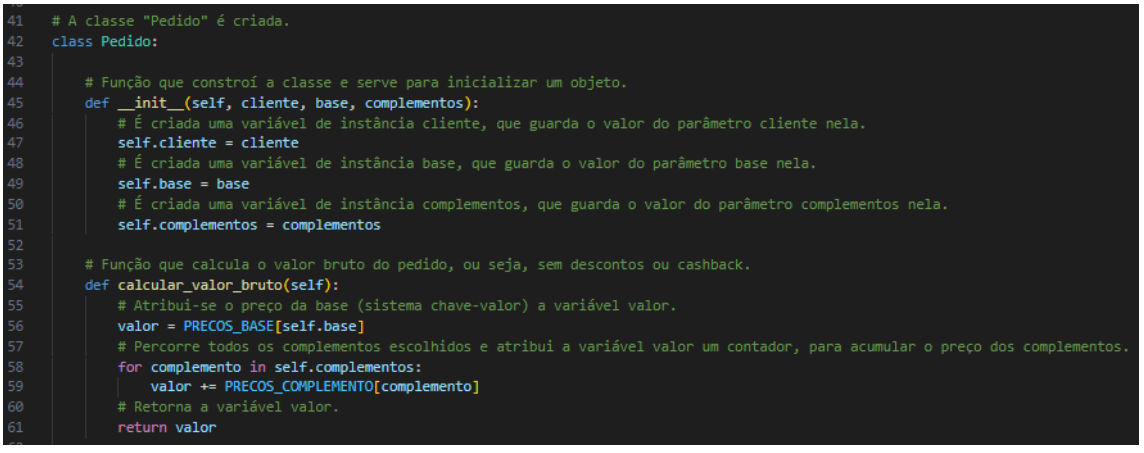
interação com o sistema operacional, nesse caso utilizada para verificar a existência do arquivo de clientes. Além disso, há a atribuição de constantes, entre elas o nome do arquivo de clientes e os preços das bases e complementos e seus respectivos nomes, cada uma sendo representada por um dicionário, como uma "tabela de preços".



```python
# A classe "Cliente" é criada.
class Cliente:
    # Função que constroí a classe e serve para inicializar um objeto, tendo parâmetros generalizados caso o cliente não tenha especificidades.
    def __init__(self, nome, tipo = "comunidade", saldo_cashback = 0.0):
        # É criada uma variável de instância nome, que guarda o valor do parâmetro nome nela.
        self.nome = nome
        # É criada uma variável de instância tipo, que guarda o valor do parâmetro tipo nela.
        self.tipo = tipo
        # É criada uma variável de instância saldo_cashback, que guarda o valor do parâmetro saldo_cashback nela com verificação de tipo (float).
        self.saldo_cashback = float(saldo_cashback)

    # Função para controlar e obter descontos de acordo com o tipo do cliente.
    def obter_desconto(self):
        # Estrutura condicional onde se a variável de instância tipo for um estudante ou um professor/funcionário da UEFS,
        # um desconto é obtido e vai ser futuramente calculado. Caso não, nenhum desconto é obtido.
        if self.tipo == "estudante":
            return 0.25
        elif self.tipo == "professor_funcionario":
            return 1.00
        else:
            return 0.0

    # Função que é utilizada para organizar os dados em um dicionário.
    # ZO nome não pertence a esse dicionário porque no arquivo JSON o nome já vai ser a chave.
    def criar_dicionario(self):
        return {"tipo": self.tipo, "saldo_cashback": self.saldo_cashback}
```

Figura 2. Criação da classe "Cliente" e de suas respectivas funções (modularização).

Nessa figura, a primeira classe do código é criada para criar e gerenciar as informações de cada cliente. Na primeira função, quando um objeto Cliente é criado, são armazenadas informações básicas, como seu nome, tipo e saldo de cashback. Na segunda, há o "cálculo do desconto", onde se o tipo do cliente tiver direito ao desconto, seu respectivo desconto será retornado. Por fim, na terceira função os dados do cliente são organizados no formato de um dicionário.



```python
# A classe "Pedido" é criada.
class Pedido:

    # Função que constroí a classe e serve para inicializar um objeto.
    def __init__(self, cliente, base, complementos):
        # É criada uma variável de instância cliente, que guarda o valor do parâmetro cliente nela.
        self.cliente = cliente
        # É criada uma variável de instância base, que guarda o valor do parâmetro base nela.
        self.base = base
        # É criada uma variável de instância complementos, que guarda o valor do parâmetro complementos nela.
        self.complementos = complementos

    # Função que calcula o valor bruto do pedido, ou seja, sem descontos ou cashback.
    def calcular_valor_bruto(self):
        # Atribui-se o preço da base (sistema chave-valor) a variável valor.
        valor = PRECOS_BASE[self.base]
        # Percorre todos os complementos escolhidos e atribui a variável valor um contador, para acumular o preço dos complementos.
        for complemento in self.complementos:
            valor += PRECOS_COMPLEMENTO[complemento]
        # Retorna a variável valor.
        return valor
```

Figura 3. Criação da classe "Pedido" e de suas respectivas funções (modularização).

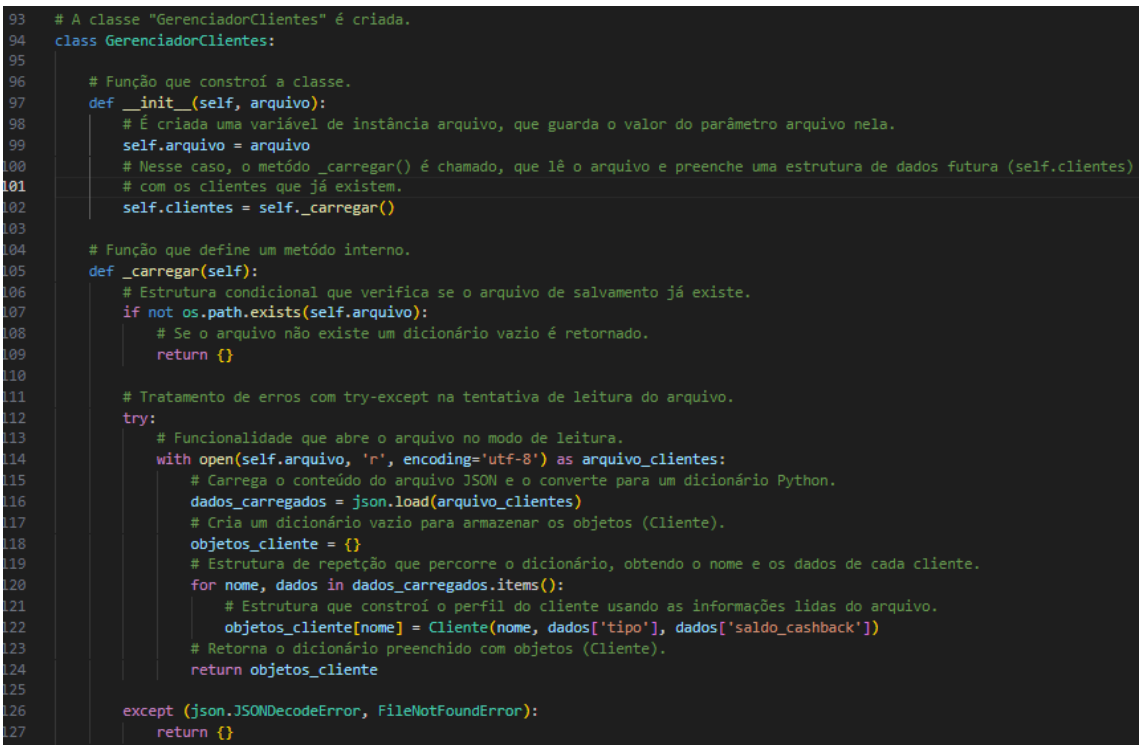
Nessa figura, a segunda classe do código é criada para representar um pedido feito por um cliente. Na primeira função, quando um objeto Pedido é criado, são guardadas informações importantes daquele pedido, como o nome do cliente, a base

escolhida e os complementos solicitados. Na segunda função, há o “cálculo” do preço bruto do pedido, sem a aplicação de descontos ou de cashback. Nela, esse preço é encontrado através da busca no dicionário pelo preço da base e pelo percorrimento dos complementos escolhidos, também no dicionário.

```
63      # Função que exibe um resumo do pedido.
64      def exibir_resumo_pedido(self, valor_final, cashback_usado):
65          # Imprime os dados do pedido.
66          print("\nRESUMO DO PEDIDO:")
67          print(f"Cliente: {self.cliente.nome}")
68          print(f"Base: {self.base}")
69          # Estrutura condicional que verifica se foram escolhidos complementos ou não.
70          if self.complementos:
71              print(f"Adicionais: {', '.join(self.complementos)}")
72          else:
73              print("Adicionais: Nenhum")
74
75          # Estrutura condicional que verifica se o cliente tem desconto especial e qual o seu tipo.
76          if self.cliente.tipo == "estudante":
77              print("Cliente com desconto especial de estudante")
78          elif self.cliente.tipo == "professor_funcionario":
79              print("Cliente com desconto especial de professor ou funcionário")
80
81          # Imprime os dados do pedido.
82          print(f"Valor Total: R$ {valor_final:.2f}")
83
84          # Estrutura condicional que verifica o cashback usado e o saldo do mesmo.
85          if cashback_usado > 0:
86              print(f"Cashback utilizado: R$ {cashback_usado:.2f}")
87          else:
88              print("Cashback não utilizado.")
89
90          # Imprime o saldo de cashback atualizado.
91          print(f"Saldo de cashback atual: R$ {self.cliente.saldo_cashback:.2f}\n")
```

Figura 4. Continuação da classe “Pedido” e de suas respectivas funções (modularização).

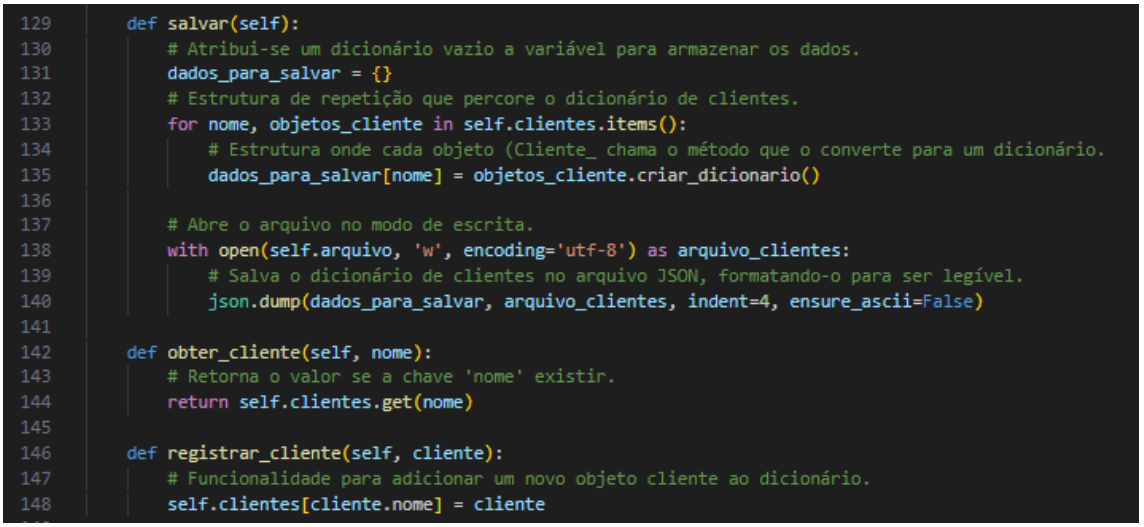
Nessa figura, é apresentada a terceira função da classe Pedido, utilizada para obtenção de um “recibo” detalhado do pedido do cliente, onde são impressos os dados e itens do pedido, sobre a possibilidade de ter descontos por ser um “cliente especial”, o valor do pedido e o saldo de cashback. Em todas as partes onde existia mais de “uma opção”, foram utilizadas estruturas condicionais (if - elif - else).



```python
# A classe "GerenciadorClientes" é criada.
class GerenciadorClientes:

    # Função que constroí a classe.
    def __init__(self, arquivo):
        # É criada uma variável de instância arquivo, que guarda o valor do parâmetro arquivo nela.
        self.arquivo = arquivo
        # Nesse caso, o metódo _carregar() é chamado, que lê o arquivo e preenche uma estrutura de dados futura (self.clientes)
        # com os clientes que já existem.
        self.clientes = self._carregar()

    # Função que define um metódo interno.
    def _carregar(self):
        # Estrutura condicional que verifica se o arquivo de salvamento já existe.
        if not os.path.exists(self.arquivo):
            # Se o arquivo não existe um dicionário vazio é retornado.
            return {}

        # Tratamento de erros com try-except na tentativa de leitura do arquivo.
        try:
            # Funcionalidade que abre o arquivo no modo de leitura.
            with open(self.arquivo, 'r', encoding='utf-8') as arquivo_clientes:
                # Carrega o conteúdo do arquivo JSON e o converte para um dicionário Python.
                dados_carregados = json.load(arquivo_clientes)
                # Cria um dicionário vazio para armazenar os objetos (Cliente).
                objetos_cliente = {}
                # Estrutura de repetção que percorre o dicionário, obtendo o nome e os dados de cada cliente.
                for nome, dados in dados_carregados.items():
                    # Estrutura que constroí o perfil do cliente usando as informações lidas do arquivo.
                    objetos_cliente[nome] = Cliente(nome, dados['tipo'], dados['saldo_cashback'])
                # Retorna o dicionário preenchido com objetos (Cliente).
                return objetos_cliente

        except (json.JSONDecodeError, FileNotFoundError):
            return {}
```

Figura 5. Criação da classe “GerenciadorClientes” e de suas respectivas funções (modularização).

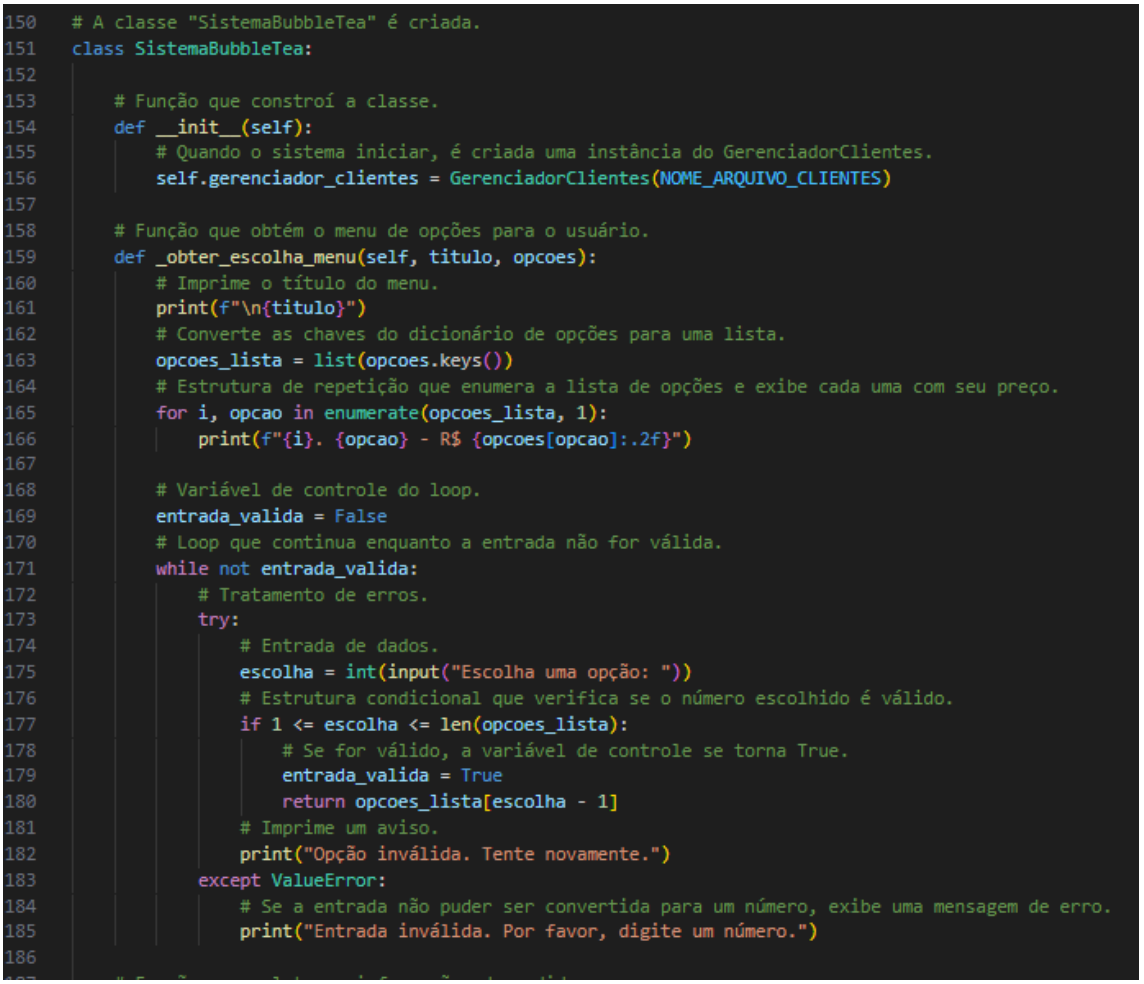
Nessa figura, a terceira classe do código é criada para carregar e gerenciar todos os dados dos clientes a partir de um arquivo. Na primeira função, quando um objeto GerenciadorClientes é criado, ele recebe o nome do arquivo onde os dados ficarão salvos e o método “_carregar ()” é chamado para ler o arquivo e preencher uma futura estrutura de dados com os clientes existentes. Na segunda função, há uma verificação da existência do arquivo, há a possível leitura do arquivo e conversão para um dicionário e por fim há o percorrimento dos dados lidos e para cada cliente encontrado, cria-se um objeto da classe Cliente. Além disso, há o tratamento de erros com try-except.



```python
    def salvar(self):
        # Atribui-se um dicionário vazio a variável para armazenar os dados.
        dados_para_salvar = {}
        # Estrutura de repetição que percore o dicionário de clientes.
        for nome, objetos_cliente in self.clientes.items():
            # Estrutura onde cada objeto (Cliente_ chama o método que o converte para um dicionário.
            dados_para_salvar[nome] = objetos_cliente.criar_dicionario()

        # Abre o arquivo no modo de escrita.
        with open(self.arquivo, 'w', encoding='utf-8') as arquivo_clientes:
            # Salva o dicionário de clientes no arquivo JSON, formatando-o para ser legível.
            json.dump(dados_para_salvar, arquivo_clientes, indent=4, ensure_ascii=False)

    def obter_cliente(self, nome):
        # Retorna o valor se a chave 'nome' existir.
        return self.clientes.get(nome)

    def registrar_cliente(self, cliente):
        # Funcionalidade para adicionar um novo objeto cliente ao dicionário.
        self.clientes[cliente.nome] = cliente
```

Figura 6. Continuação da classe “GerenciadorClientes” e de suas respectivas funções (modularização).

Nessa figura, é apresentada a terceira função da classe GerenciadorClientes, que salva todos os dados dos clientes no arquivo JSON. Nela, há o percorrimento da lista de objetos Cliente e para cada objeto o método “criar_dicionario ()” é evocado. Depois o modo de escrita é ativado e a biblioteca “json” escreve os dados no arquivo. Na quarta função, há uma busca e retorno de um cliente pelo seu nome e na quinta função há a adição de um novo cliente ao “sistema”.



```python
# A classe "SistemaBubbleTea" é criada.
class SistemaBubbleTea:

    # Função que constroí a classe.
    def __init__(self):
        # Quando o sistema iniciar, é criada uma instância do GerenciadorClientes.
        self.gerenciador_clientes = GerenciadorClientes(NOME_ARQUIVO_CLIENTES)

    # Função que obtém o menu de opções para o usuário.
    def _obter_escolha_menu(self, titulo, opcoes):
        # Imprime o título do menu.
        print(f"\n{titulo}")
        # Converte as chaves do dicionário de opções para uma lista.
        opcoes_lista = list(opcoes.keys())
        # Estrutura de repetição que enumera a lista de opções e exibe cada uma com seu preço.
        for i, opcao in enumerate(opcoes_lista, 1):
            print(f"{i}. {opcao} - R$ {opcoes[opcao]:.2f}")

        # Variável de controle do loop.
        entrada_valida = False
        # Loop que continua enquanto a entrada não for válida.
        while not entrada_valida:
            # Tratamento de erros.
            try:
                # Entrada de dados.
                escolha = int(input("Escolha uma opção: "))
                # Estrutura condicional que verifica se o número escolhido é válido.
                if 1 <= escolha <= len(opcoes_lista):
                    # Se for válido, a variável de controle se torna True.
                    entrada_valida = True
                    return opcoes_lista[escolha - 1]
                # Imprime um aviso.
                print("Opção inválida. Tente novamente.")
            except ValueError:
                # Se a entrada não puder ser convertida para um número, exibe uma mensagem de erro.
                print("Entrada inválida. Por favor, digite um número.")
```

Figura 7. Criação da classe “SistemaBubbleTea” e de suas respectivas funções (modularização).

Nessa figura, a quarta classe do código é criada para organizar o funcionamento do programa. Na primeira função, quando a classe é iniciada, há a criação de uma instância de “GerenciadorClientes”. Na segunda função, o objetivo é exibir o menu e fornecer uma entrada para o usuário, até que ele escolha uma opção válida do menu de opções.

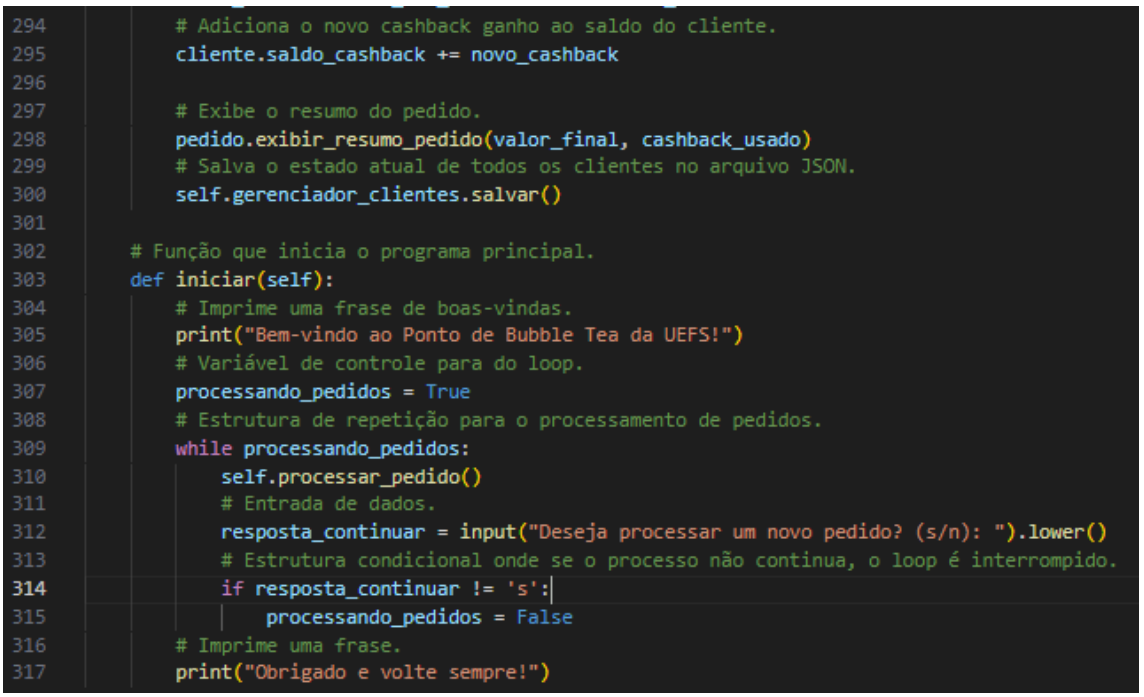
```python
    # Função que coleta as informações do pedido.
    def _coletar_informacoes_pedido(self):
        # Entrada de dados.
        nome_cliente = input("Digite o nome do cliente: ").strip().title()
        # Usa a funcionalidade do gerenciador para verificar se o cliente existe.
        cliente = self.gerenciador_clientes.obter_cliente(nome_cliente)

        # Estrutura condicional onde se o cliente não for encontrado, ele é um cliente novo.
        if not cliente:
            # Imprime uma frase.
            print("Cliente novo! Vamos fazer um rápido cadastro.")
            # Variável de controle para o loop.
            tipo_definido = False
            # Estrutura de repetição para garantir que um tipo de cliente válido seja escolhido.
            while not tipo_definido:
                # Entrada de dados.
                tipo_input = input("O cliente é (1) Estudante UEFS, (2) Professor/Funcionário UEFS ou (3) Comunidade? ")
                # Estrutura condicional que verifica o tipo de cliente.
                if tipo_input == '1':
                    tipo = "estudante"
                    tipo_definido = True
                elif tipo_input == '2':
                    tipo = "professor_funcionario"
                    tipo_definido = True
                elif tipo_input == '3':
                    tipo = "comunidade"
                    tipo_definido = True
                else:
                    print("Opção inválida.")
            # Cria um novo objeto Cliente com as informações coletadas.
            cliente = Cliente(nome_cliente, tipo)
            # Registra o novo cliente no gerenciador de clientes.
            self.gerenciador_clientes.registrar_cliente(cliente)

        # Usa um dos métodos para obter a escolha da base do chá.
        base_escolhida = self._obter_escolha_menu("BASES", PRECOS_BASE)
```

```python

        # Atribui-se a uma variável uma lista vazia.
        complementos_escolhidos = []
        # Variável de controle para o loop.
        adicionando_complementos = True
        # Estrutura de repetição que permite que o cliente adicione vários complementos.
        while adicionando_complementos:
            # Imprime uma pergunta.
            print("\nAdicionar um complemento?")
            # Usa um dos métodos para obter a escolha do complemento.
            complemento = self._obter_escolha_menu("COMPLEMENTOS", PRECOS_COMPLEMENTO)
            # Adiciona o complemento escolhido à lista.
            complementos_escolhidos.append(complemento)

            # Variável de controle para o loop.
            continuar = True

            # Estrutura de repetição para continuar enquanto complementos ainda sejam adicionados.
            while continuar:
                # Entrada de dados
                resposta_continuar = input("Deseja adicionar outro complemento? (s/n): ").lower()
                if resposta_continuar in ("s", "n"):
                    # Interrupção de loop.
                    continuar = False

                print("Opção inválida. Por favor, digite 's' ou 'n'.")

            # Estrutura condicional onde se o processo não continua, o loop é interrompido.
            if resposta_continuar == "n":
                adicionando_complementos = False

        # Retorna o objeto cliente e as escolhas de produtos.
        return cliente, base_escolhida, complementos_escolhidos
```

Figuras 8 e 9. Continuação da classe “SistemaBubbleTea” e de suas respectivas funções (modularização).

Nessas figuras, temos a terceira função da classe SistemaBubbleTea, onde o objetivo é fornecer uma entrada de dados e obter alguns dados do pedido. O primeiro passo é verificar o nome do cliente e utiliza-se o “gerenciador_clientes” para se verificar isso. Caso ele não exista, o usuário escolhe seu “tipo” (estrutura condicional) e um novo objeto Cliente é criado. Caso exista, o programa pula direto para o pedido. No pedido, ele escolhe sua base por meio do menu e na parte de complementos um loop while continua até que ele escolha todos os complementos desejados. Há outra entrada de dados, onde ele digita “s” ou “n” para continuar ou não. Depois, o cliente, a base e os complementos são retornados na função.

```
257     # Função para o processamento do pedido.
258     def processar_pedido(self):
259         # Obtém todas as informações do pedido.
260         cliente, base, complementos = self._coletar_informacoes_pedido()
261         # Cria um objeto Pedido com as informações.
262         pedido = Pedido(cliente, base, complementos)
263 
264         # Realização dos cálculos de preço e desconto.
265         valor_bruto = pedido.calcular_valor_bruto()
266         desconto = cliente.obter_desconto()
267 
268         # Aplica o desconto de acordo com o tipo de cliente.
269         valor_com_desconto = valor_bruto
270         # Estrutura condicional.
271         if cliente.tipo == "estudante":
272             valor_com_desconto -= valor_bruto * desconto
273         elif cliente.tipo == "professor_funcionario":
274             valor_com_desconto -= desconto
275 
276         # Cálculos do novo cashback do cliente.
277         novo_cashback = valor_com_desconto * 0.10
278         cashback_usado = 0.0
279 
280         # Estrutura condicional que verifica se o cliente tem saldo de cashback.
281         if cliente.saldo_cashback > 0:
282             # Imprime algumas informações do pedido.
283             print(f"\nValor do pedido: R$ {valor_com_desconto:.2f}")
284             print(f"Você possui R$ {cliente.saldo_cashback:.2f} de cashback.")
285             usar = input("Deseja usar seu cashback? (s/n): ").lower()
286             if usar == 's':
287                 # Usa o menor valor entre o saldo de cashback e o valor do pedido.
288                 cashback_usado = min(cliente.saldo_cashback, valor_com_desconto)
289                 # Variável contadora que subtrai o cashback usado do saldo do cliente.
290                 cliente.saldo_cashback -= cashback_usado
291 
292         # Calcula o valor final.
293         valor_final = valor_com_desconto - cashback_usado
```



```
294         # Adiciona o novo cashback ganho ao saldo do cliente.
295         cliente.saldo_cashback += novo_cashback
296
297         # Exibe o resumo do pedido.
298         pedido.exibir_resumo_pedido(valor_final, cashback_usado)
299         # Salva o estado atual de todos os clientes no arquivo JSON.
300         self.gerenciador_clientes.salvar()
301
302     # Função que inicia o programa principal.
303     def iniciar(self):
304         # Imprime uma frase de boas-vindas.
305         print("Bem-vindo ao Ponto de Bubble Tea da UEFS!")
306         # Variável de controle para do loop.
307         processando_pedidos = True
308         # Estrutura de repetição para o processamento de pedidos.
309         while processando_pedidos:
310             self.processar_pedido()
311             # Entrada de dados.
312             resposta_continuar = input("Deseja processar um novo pedido? (s/n): ").lower()
313             # Estrutura condicional onde se o processo não continua, o loop é interrompido.
314             if resposta_continuar != 's':
315                 processando_pedidos = False
316         # Imprime uma frase.
317         print("Obrigado e volte sempre!")
```

Figuras 10 e 11. Continuação da classe “SistemaBubbleTea” e de suas respectivas funções (modularização).

Nessas figuras, temos a quarta e a quinta função da classe SistemaBubbleTea, onde o objetivo da função “iniciar” é controlar o programa, executando um “laço” que evoca a função “processar_pedido” continuamente. A funcionalidade de “processar_pedido” é gerenciar uma “transação” completa. Seu algoritmo primeiro coleta os dados do pedido, calcula o valor com descontos específicos do cliente, e em seguida gerencia o cashback, perguntando se o cliente deseja usar seu saldo existente. Ao final, o saldo de cashback do cliente é atualizado com o valor ganho e o valor que foi utilizado. Depois disso, um resumo detalhado é exibido e tudo é salvo no arquivo JSON.

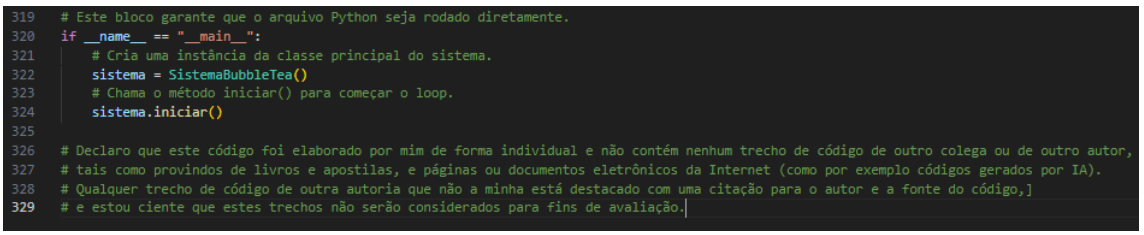


Figura 12. Estrutura condicional para o main e evocação do mesmo.

O último passo garante que o código só será executado diretamente pelo script e não por importações como módulos, por exemplo. Além disso, há uma criação de uma instância da classe principal do sistema e o método de inicialização é evocado.

2.3. Ordem de Codificação

Nesse terceiro problema PBL, houveram muitas dificuldades a serem enfrentadas, principalmente em relação a essa ideia de classes e objetos e a persistência de dados. A complexidade do problema aumentou um pouco, mas no sentido de funcionalidades "mais internas" do programa. Tendo isso em vista, utilizei a seguinte ordem de codificação:

1. Estabeleci quais seriam os dados fixos do programa, como o nome do arquivo e os nomes e preços do point de bubble tea.
2. Pensei em criar somente uma classe, que seria a classe Cliente, mas percebi que se eu utilizasse mais algumas classes talvez o programa ficasse um pouco mais organizado.
3. Criei funções (modularização) para diversos itens do programa, como para o menu e para cálculos no sistema.
4. Em relação a persistência de dados, optei por utilizar JSON e fui inserindo as informações e formatando o arquivo onde seriam guardadas as informações.
5. Criei um programa principal para organização e execução do código.

O sistema de gerenciamento para o point de bubble tea foi produzido utilizando a linguagem de programação Python 3.13.2, com o Visual Studio Code 1.98 como Ambiente Integrado de Desenvolvimento (*IDE*) e o sistema operacional Windows 10 Home Single Language.

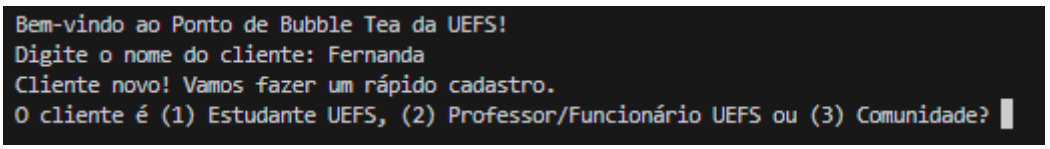
3. Resultados e Discussões

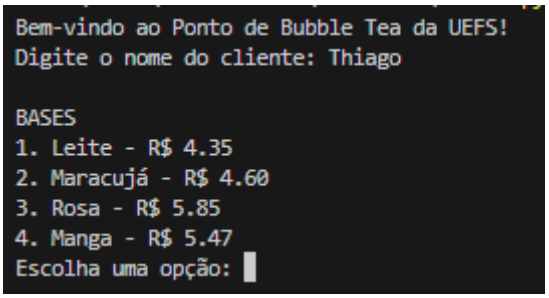
Basicamente, na execução do programa, ele já estará pronto para processar o primeiro pedido. O cliente digita seu nome e se o cliente for novo, o programa vai solicitar o tipo do cliente, mas caso contrário ele é redirecionado diretamente para a elaboração do pedido. Ele escolhe a base por meio dos números do menu e depois escolhe os complementos da mesma forma, com a diferença de responder sim (s) ou não (n) para adicionar novo complemento. O valor do pedido é exibido e se houver cashback, o

usuário escolhe com sim (s) ou não (n) se quer utilizar. Depois, o resumo final do pedido é exibido e há uma pergunta para processar novo pedido ou não, também com a lógica do sim (s) ou não (n).

Esse trecho já foi citado na introdução, mas somente reforçando que “em relação às entradas do programa, pode-se afirmar que as mesmas estão presentes na identificação do cliente, na afirmação do tipo de cliente, na escolha das bases e complementos do pedido, na adição de complementos e na escolha de processar um novo pedido. No viés contrário, ou seja, tratando das saídas, tem-se a impressão das frases de orientação para a continuação do programa, o menu/cardápio e a saída que apresenta toda a descrição do pedido.”

3.1. Testes





Figuras 1 e 2. Verificação da existência do cliente.

Como pode ser visto, a funcionalidade de verificação de existência de um determinado cliente funciona.

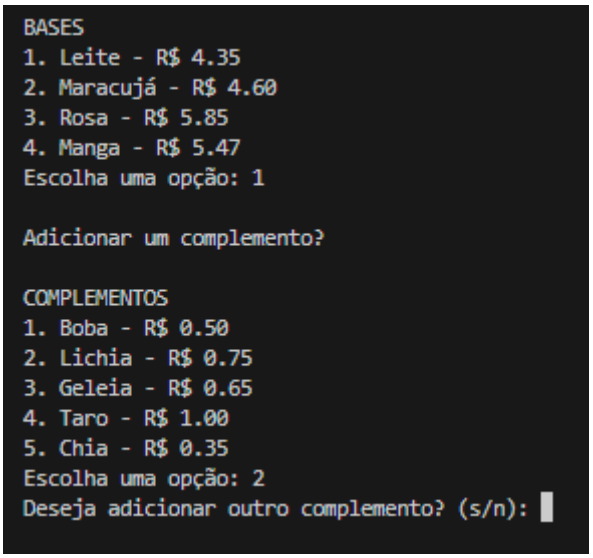


Figura 3. Adição de bases e complementos.

Como pode ser visto, a funcionalidade de adicionar base e complementos funciona normalmente.

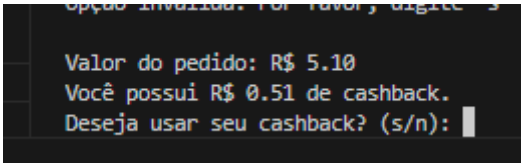


Figura 4. Verificação da existência de cashback.

Como pode ser visto, a funcionalidade de procurar cashback também funciona, lembrando que nessa imagem o cashback é da compra anterior.

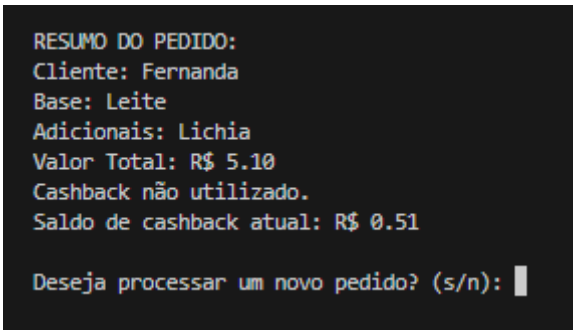


Figura 5. Verificação do resumo do pedido e novo processamento.

Como pode ser visto, a verificação do resumo do pedido e novo processamento funciona.

3.2. Erros

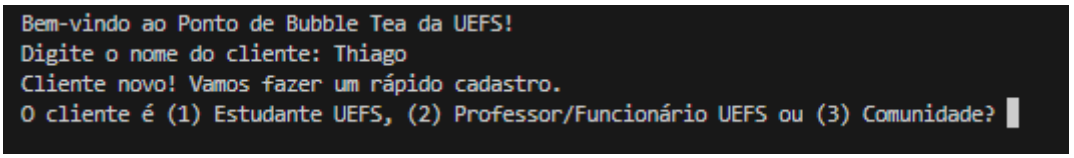


Figura 1. Erro de cadastrar dois clientes como o mesmo cliente.

Analisando, é possível perceber que se duas pessoas colocarem Thiago, o sistema vai considerar como a mesma pessoa.

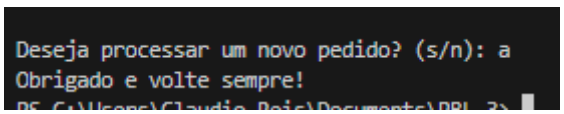


Figura 2. Erro onde o processamento aceita qualquer coisa.

Teoricamente, faltou a verificação de entrada correta.

4. Conclusão

Analisando todo o trabalho, acredito que a criação do “Sistema de Gerenciamento de Clientes e Pedidos para um Point de Bubble Tea” foi bem-sucedida e quase todos os requisitos foram cumpridos, apenas o histórico de pedidos não foi feito e há a existência de pequenos erros no código e um tratamento de erros imperfeito, mas nada que interfira na função principal do programa.

Além disso, como possíveis melhorias para o programa, poderiam ser desenvolvidas as seguintes funcionalidades: interface do programa, melhor otimização do código, histórico de pedidos, tratamento de erros mais eficiente e correção dos pequenos erros.

4.1. Detalhe Importante

O programa foi criado com o auxílio de Inteligência Artificial (IA), mais especificamente o Google AI Studio, disponível em: https://aistudio.google.com.

5. Referências Bibliográficas

DOWNEY, Allen B. Pense em Python: pense como um cientista da computação. 2. ed. São Paulo: Novatec, 2016. Disponível em: http://penseallen.github.io/PensePython2e/. Acesso em 17 de Março de 2025.