Sistema de monitoramente de casos de dengue em Feira de Santana

Adson Victor de Souza Alves

UEFS – Universidade Estadual de Feira de Santana
Av. Transnordestina, s/n, Novo Horizonte
Feira de Santana – BA, Brasil – 44036-900

21uefs@gmail.com

Resumo. *Em uma parceria entre o curso de Engenharia de Computação da Universidade Estadual de Feira de Santana e o sistema de vigilância de Feira de Santana foi elaborada o projeto de um sistema para monitoramente de casos de dengue na cidade visando o controle e redução de casos. O sistema realiza tanto os processo de visualização dos dados registrados, quanto permite novos registros. A elaboração do código foi em linguagem Python utilizando o IDE e editor de códigos Visual Studio Code. O programa atende aos requisitos solicitados de visualização e gravação de novos dados, a partir de cada formato solicitado e conta com funcionalidade extra para prover melhor experiência ao usuário.*

1. Introdução

Detectar mudanças significativas na ocorrência e distribuição das doenças, identificação quantificação e monitoramento das tendências e padrões do processo saúde-doença nas populações, além de observar as mudanças nos padrões de ocorrência dos agentes e hospedeiros são características fundamentais para o controle epidemiológico, visando alcançar o bem-estar de toda a população afetada ou que corre risco de ser impactada por doenças.

Em uma parceria entre o curso de Engenharia de Computação da Universidade Estadual de Feira de Santana (UEFS) com o sistema de vigilância de Feira de Santana, foi elaborado um projeto para desenvolvimento de um software chamado “Dengue Free Feira” que tem por objetivo monitorar a propagação da dengue, doença viral transmitida principalmente pelos mosquitos Aedes aegypti e Aedes albopictus, e a ocorrência de seus casos. Consequentemente, a coleta e registro desses dados têm por função o auxílio no controle da propagação da doença juntamente com a redução do número de casos.

A solução do programa é organizada a partir da construção de um menu para a navegação do usuário, que disponibilizará, a princípio, as opções para exibição das informações do problema (dengue), leitura e gravação de dados e encerramento do programa.

2. Metodologia

Nesta seção será abordado sobre o desenvolvimento do sistema associado aos requisitos do programa, de monitoramento e controle dos casos, bem como se deu todo o processo de codificação do software.

Para atingir os objetivos propostos, o sistema foi subdividido em funções: construção de um menu para navegação, exibir todos os dados, exibir os dados a partir de

uma data ou bairro informados pelo usuário, exibir os dados para um intervalo de datas solicitadas, exibição dos dados em formato percentual e atualização dos dados para novas datas informadas. Para todas as funções requeridas foram criadas funções a partir da declaração *def* , portanto, o código é devidamente modularizado. No código principal, onde estará a função *main()*, foram utilizadas as declarações *match case*, que funcionam semelhante a um bloco de condicionais *if else*, para fazer a diferenciação das opções informadas pelo usuário.

Para a criação do menu foi criada uma função que consiste na leitura de uma lista que conterá as opções para serem exibidas, um contador que atualiza a cada item lido por um *loop for* que percorre por todos os itens da lista inserida e um bloco de *try* e *except*, juntamente com um *input* que solicitará a opção do usuário e retorna esse valor a partir da declaração *return*.

```
#MENU

def menu(list):

    enum = 1
    print('\n')
    for item in list:
        print(f'[{enum}]. {item}')
        enum += 1

    try:
        option = int(input('\nInsira uma opção: '))
        return option
    except:
        print('\nOpção inválida!')
        return menu(list)
```

Figura 1 - Menu.

Para a leitura de dados foi utilizada a biblioteca **csv**, que realiza a leitura de um determinado arquivo, nesse caso, de uma tabela de dados iniciais, que consiste em um arquivo de extensão .csv (comma separated values, ou valores separados por vírgula) e também na gravação ou atualização de novos dados.

Foi utilizada a declaração *with open*, nativas da própria linguagem Python, que abre um ou mais determinado arquivo e garante a finalização de recursos adquiridos, para a execução do arquivo. Em sequência é utilizado a função *csv.reader()* para realizar a leitura do arquivo a partir de listas e também a função *next()* para obtenção dos nomes das colunas da tabela, que é uma outra maneira de iteração no Python. Por fim, os itens da tabela foram iterados por um *loop for* de forma abreviada, agrupados em tuplas de chave-valor a partir da função *zip()*, os valores de cada linha foram adicionados a diferentes dicionários e finalmente todos esses dicionários adicionados à uma lista para facilitar a leitura em necessidades futuras.

```python
with open('./PBL_DENGUE/dados.csv', 'r', encoding='utf-8', newline='') as archive:
    report = csv.reader(archive)
    headers = next(report) #Cabeçalho
    data = [{header: value for header, value in zip(headers, row)} for row in report]
```

Figura 2 – Leitura do arquivo.

A fim de cumprir o requisito de exibição de todos os dados, foi criada uma outra função com a entrada de um parâmetro do tipo *List*, nesse caso, da lista criada a partir da leitura do arquivo com todos os dados agrupados em dicionários. A princípio a função realiza a atribuição de todas as colunas em variáveis, a iteração de todos os dicionários a partir de um *loop for* e a cada ciclo é obtido cada um dos dados de cada coluna e linha por meio do método *get()*, que tem como parâmetro a chave de um dicionário e retorna o valor da chave informada. A partir de cada ciclo, os dados são exibidos no terminal organizadamente por funções *print()* e métodos de formatação de *strings*, como o que foi utilizado, *center()*, que centraliza uma *string*, preenchendo os espaços laterais excedentes com *whitespaces* ou espaços em branco.

```python
def getFullData(data):

    date = 'Data'
    district = 'Bairro'
    population = 'Habitantes'
    suspects = 'Casos Suspeitos'
    negatives = 'Casos Negativos'
    confirmeds = 'Casos Confirmados'

    print(f'| {date.center(10)} | | {district.center(20)} | | {population.center(10)} | | {suspects.center(15)} | | {negatives.center(15)} | | {confirmeds.center(17)} |')
    print('-----------------------------------------------------------------------------------------------------------\n')

    for item in data:
        data_date = item.get('Data')
        data_district = item.get('Bairro')
        data_population = item.get('Habitantes')
        data_suspects = item.get('Casos Suspeitos')
        data_negatives = item.get('Casos Negativos')
        data_confirmeds = item.get('Casos Confirmados')

        print(f'| {data_date.center(10)} | | {data_district.center(20)} | | {data_population.center(10)} | | {data_suspects.center(15)} | | {data_negatives.center(15)} | | {data_confirmeds.center(17)} |')
        print('-----------------------------------------------------------------------------------------------------------')
```

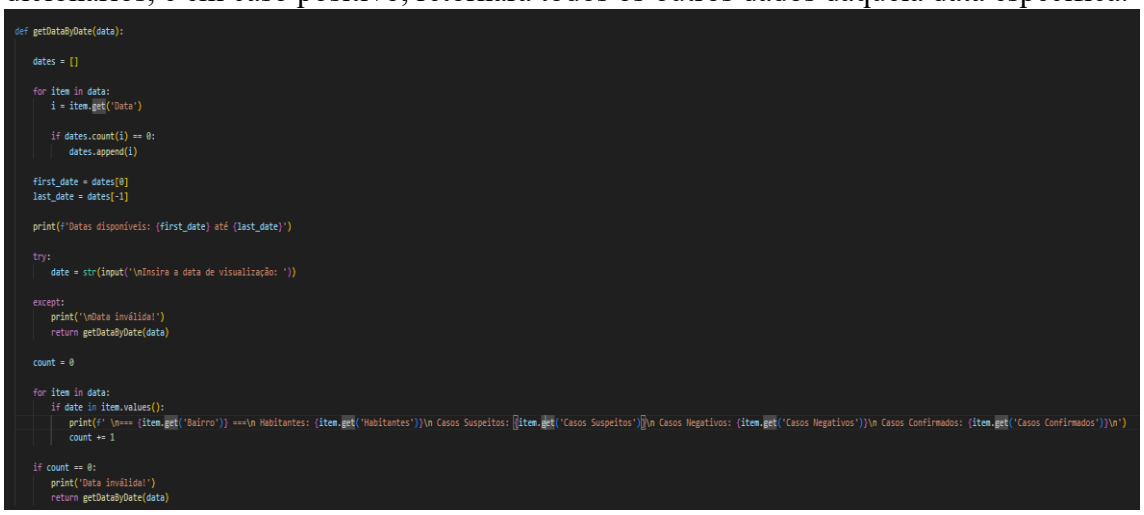
Figura 3 – Exibição dos dados.

Os dados das tabelas são distribuídos em 6 colunas: data, bairro, habitantes, casos suspeitos, casos negativos e casos confirmados, nesse mesma ordem.

Para a exibição desses dados por data e/ou por bairro informado, foi criada uma outra função com entrada de parâmetro também do tipo *List*. Inicialmente é criada uma lista que conterá todas as datas da tabela e preenchida a partir de um *loop for* que iterará a lista de parâmetro e fará uma verificação de todas as datas utilizando o método *get()* para obter o valor, o método *count()* associado à uma condicional *if else* para verificar se o item já existe dentro da nova lista, e se for inexistante, será adicionado. Após a verificação de todos os itens, retornará a primeira data e a última a partir do acesso padrão de itens de uma lista e serão exibidas ao usuário esse intervalo para saber quais datas estão disponíveis para pesquisa.

Após o usuário informar uma data dentro desse intervalo, essa data será armazenada em uma variável e utilizada para busca dentro da lista de parâmetro novamente a partir de um outro *loop for*. Sequencialmente, um outro bloco de

condicionais *if else* fará a verificação se a data é correspondente a cada um dos dicionários, e em caso positivo, retornará todos os outros dados daquela data específica.



```
def getDataByDate(data):

    dates = []

    for item in data:
        i = item.get('Data')

        if dates.count(i) == 0:
            dates.append(i)

    first_date = dates[0]
    last_date = dates[-1]

    print(f'Datas disponíveis: {first_date} até {last_date}')

    try:
        date = str(input('\nInsira a data de visualização: '))

    except:
        print('\nData inválida!')
        return getDataByDate(data)

    count = 0

    for item in data:
        if date in item.values():
            print(f' \n=== {item.get('Bairro')} ===\n Habitantes: {item.get('Habitantes')}\n Casos Suspeitos: {item.get('Casos Suspeitos')}\n Casos Negativos: {item.get('Casos Negativos')}\n Casos Confirmados: {item.get('Casos Confirmados')}\n')
            count += 1

    if count == 0:
        print('Data inválida!')
        return getDataByDate(data)
```

Figura 4 – Exibição a partir de uma data

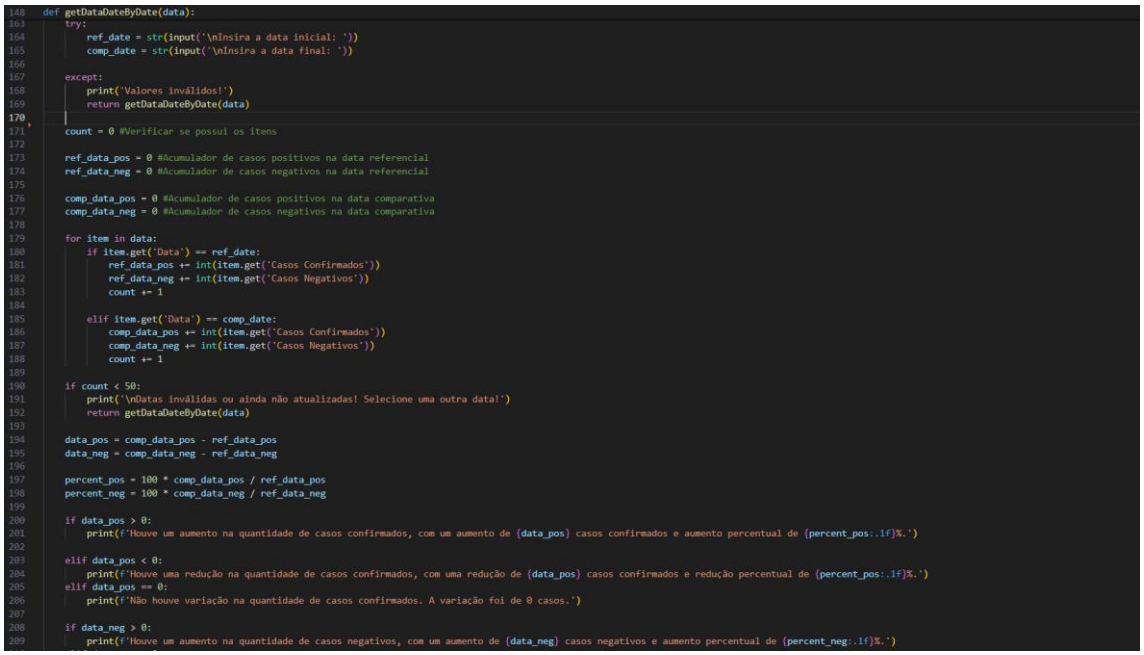
A exibição dos dados a partir de um bairro informado funciona de maneira semelhante, apenas alterando a forma como os bairros disponíveis para busca são informados para o usuário. Diferentemente da exibição dos dados por data, no qual foi utilizado a função *count()* para determinar se uma data é repetida ou não, nessa função utiliza-se a função *len()* que, para esse caso, retorna a quantidade de itens em uma lista, previamente criada, e como já é sabido a quantidade de bairros monitorados, pode-se apenas comparar entre si e interromper o iterador (*loop for*) quando forem iguais, indicando que a lista já foi preenchida com todos os bairros.

```
def getDataByDistrict(data):

    districts = []

    for item in data:
        districts.append(item.get('Bairro'))

        if len(districts) == 25:
            break

    print(f'Bairros disponíveis para visualização: {districts}')

    try:
        district = str(input('\nInsira o bairro para visualização: '))
        district = district.title()

    except:
        print('\nBairro inválido!')
        return getDataByDistrict(data)

    count = 0

    for item in data:
        if district in item.values():
            print(f' \n=== {item.get('Bairro')} === \n Data: {item.get('Data')} \n Habitantes: {item.get('Habitantes')}\n Casos Suspeitos: {item.get('Casos Suspeitos')}\n
            count += 1
```

Figura 5 – Exibição a partir de um bairro.

Para a comparação de dados em um intervalo de duas datas, primeiro é informado para o usuário as datas disponíveis utilizando a mesma execução mostrada anteriormente, na exibição por data solicitada (Figura 4). Na sequência serão solicitados para o usuários duas datas: uma data referencial (data inicial) e uma data comparativa (data final). Após informadas as datas, serão definidos acumuladores para casos positivos e casos negativos

nas duas datas informadas. Novamente será utilizado um *loop for* para iteração dos dados, atribuição e somatória nos acumuladores, utilizando de uma condicial para identificar os itens que correspondem as datas informadas. Após todos os dados coletados, executará uma expressão algébrica para calcular os valores percentuais de variação entre as duas datas informadas e por fim, exibidas ao usuário.

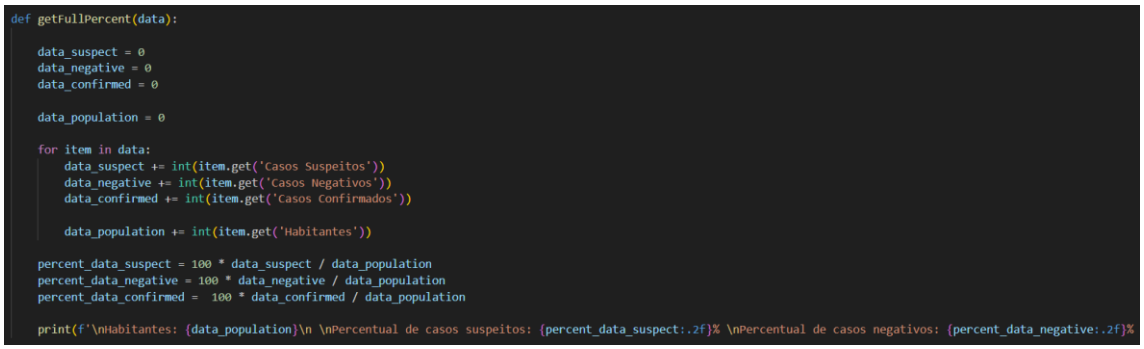


```
def getDataDateByDate(data):
    try:
        ref_date = str(input('\nInsira a data inicial: '))
        comp_date = str(input('\nInsira a data final: '))

    except:
        print('Valores inválidos!')
        return getDataDateByDate(data)

    count = 0 #Verificar se possui os itens

    ref_data_pos = 0 #Acumulador de casos positivos na data referencial
    ref_data_neg = 0 #Acumulador de casos negativos na data referencial

    comp_data_pos = 0 #Acumulador de casos positivos na data comparativa
    comp_data_neg = 0 #Acumulador de casos negativos na data comparativa

    for item in data:
        if item.get('Data') == ref_date:
            ref_data_pos += int(item.get('Casos Confirmados'))
            ref_data_neg += int(item.get('Casos Negativos'))
            count += 1

        elif item.get('Data') == comp_date:
            comp_data_pos += int(item.get('Casos Confirmados'))
            comp_data_neg += int(item.get('Casos Negativos'))
            count += 1

    if count < 50:
        print('\nDatas inválidas ou ainda não atualizadas! Selecione uma outra data!')
        return getDataDateByDate(data)

    data_pos = comp_data_pos - ref_data_pos
    data_neg = comp_data_neg - ref_data_neg

    percent_pos = 100 * comp_data_pos / ref_data_pos
    percent_neg = 100 * comp_data_neg / ref_data_neg

    if data_pos > 0:
        print(f'Houve um aumento na quantidade de casos confirmados, com um aumento de {data_pos} casos confirmados e aumento percentual de {percent_pos:.1f}%.')

    elif data_pos < 0:
        print(f'Houve uma redução na quantidade de casos confirmados, com uma redução de {data_pos} casos confirmados e redução percentual de {percent_pos:.1f}%.')
    elif data_pos == 0:
        print(f'Não houve variação na quantidade de casos confirmados. A variação foi de 0 casos.')

    if data_neg > 0:
        print(f'Houve um aumento na quantidade de casos negativos, com um aumento de {data_neg} casos negativos e aumento percentual de {percent_neg:.1f}%.')
```

Figura 6 – Intervalo de datas

Para a exibição dos dados em valor percentual foram criadas duas funções, além da função citada anteriormente, que também exibe valores percentuais e variação. A primeira para exibir o valor percentual para todos os dados e a segunda para exibir o valor percentual de um único bairro informado pelo usuário.

A primeira função consiste, inicialmente, na criação de variáveis que serão utilizadas como acumuladores, para os dados de casos suspeitos, casos negativos, casos confirmados e o número total de habitantes. Criadas as variáveis, será utilizado novamente um *loop for* para percorrer por todos os dicionários, obter os valores desejados e então realizar a somatório dentro dos acumuladores. Após coletar todos os dados, tem-se apenas uma expressão algébrica para determinar os valores percentuais. E por último, serão imprimidos para o usuário esses valores calculados.



```
def getFullPercent(data):

    data_suspect = 0
    data_negative = 0
    data_confirmed = 0

    data_population = 0

    for item in data:
        data_suspect += int(item.get('Casos Suspeitos'))
        data_negative += int(item.get('Casos Negativos'))
        data_confirmed += int(item.get('Casos Confirmados'))

        data_population += int(item.get('Habitantes'))

    percent_data_suspect = 100 * data_suspect / data_population
    percent_data_negative = 100 * data_negative / data_population
    percent_data_confirmed =  100 * data_confirmed / data_population

    print(f'\nHabitantes: {data_population}\n \nPercentual de casos suspeitos: {percent_data_suspect:.2f}% \nPercentual de casos negativos: {percent_data_negative:.2f}%
```

Figura 7 – Dados percentuais totais.

Para a segunda função, utiliza-se, basicamente, de uma mescla entre as funções de exibição dos dados por bairro e de valores percentuais totais. A função é iniciada a partir da exibição dos bairros disponíveis para o usuário, conforme a figura 5. Após exibidos os bairros, o usuário informa qual bairro deseja visualizar os valores percentuais e novamente utiliza-se de um *loop for* para percorrer os dicionários e condicionais para obter os dados desejados. Por fim, será executada novamente uma expressão algébrica para cálculo dos valores percentuais e exibidos para o usuário.

```
def getPercentByDistrict(data):
    try:
        district = str(input('\nInsira o bairro para visualização: '))
        district.capitalize()

    except:
        print('\nBairro inválido!')
        return getDataByDistrict(data)

    data_suspect = 0
    data_negative = 0
    data_confirmed = 0

    count = 0

    for item in data:
        if item.get('Bairro') == district:
            data_suspect += int(item.get('Casos Suspeitos'))
            data_negative += int(item.get('Casos Negativos'))
            data_confirmed += int(item.get('Casos Confirmados'))

            data_population = int(item.get('Habitantes'))

            count += 1

    if count == 0:
        print('\nBairro inválido!')
        return getPercentByDistrict(data)

    percent_data_suspect = 100 * data_suspect / data_population
    percent_data_negative = 100 * data_negative / data_population
    percent_data_confirmed =  100 * data_confirmed / data_population

    print(f'\nBairro selecionado: {district} \nHabitantes: {data_population} \nPercentual de casos suspeitos: {percent_data_suspect:.2f}% \nPercentual de casos negativos
```

Figura 8 – Dados percentuais por bairro.

Em conclusão, tem-se a função para atualização de novos dados. A leitura dos dados é feita internamente para reutilização do cabeçalho (chave de cada coluna) e a fim de garantir o fechamento e abertura do arquivo para atualização a partir de métodos internos registrados dentro da própria função.

A função inicia-se justamente com a leitura dos dados, da mesma maneira como realizada anteriormente (Figura 2). Posteriormente, iniciará um *loop while* que fará com que a função siga funcionando até o usuário decidir não registrar mais nenhum dado. Dentro do *loop* será solicitado para o usuário a confirmação de todos os dados de cada coluna (data, bairro, habitantes, casos suspeitos, casos negativos e casos confirmados). Todos os dados são tratados a partir de um bloco *try* e *except* impedindo qualquer possível erro de quebrar o funcionamento do código.

```python
def dataUpdate():
    option = menu(['Registrar novos dados', 'Encerrar'])

    while option != 2:

        match option:

            case 1:

                try:
                    date = str(input('Data: '))
                    district = str(input('Bairro: '))
                    population = str(input('Habitantes: '))

                    if population.isnumeric() == False:
                        print('\n"Habitantes" deve conter apenas números!\n')
                        return dataUpdate()

                    suspects = int(input('Casos Suspeitos: '))
                    confirmeds = int(input('Casos Negativos: '))
                    negatives = int(input('Casos Confirmados: '))

                except:
                    print('\nValor inválido!\n')
                    return dataUpdate()
```

Figura 9 – *Loop while* e solicitação de dados.

Tendo os valores inseridos pelo usuário devidamente verificados, será realizada a atualização dos casos suspeitos a partir dos casos suspeitos, casos confirmados e casos negativos inseridos pelo usuário. Essa atualização baseia-se na estrutura de iteração dos dicionários buscando a última linha da tabela e obtendo a quantidade de casos suspeitos, para em sequência ser realizada a atualização do valor de casos suspeitos.

```
def dataUpdate():
                #ATUALIZAÇÃO DO NÚMERO DE SUSPEITOS

                sumConNeg = confirmeds + negatives

                for item in data:
                    if item.get('Bairro') == district:
                        suspects_2 = int(item.get('Casos Suspeitos'))
                        total_suspects = suspects_2 + suspects

                suspects = total_suspects - sumConNeg

            case 2:
                print('\nRegistro encerrado!')

            case _:
                print('\nOpção inválida!\n')
```

Figura 10 – Atualização de casos suspeitos.

Após todos os valores terem sido tratados, serão inseridos na tabela utilizando um modo alternativo de abertura de arquivos da biblioteca **csv**. Esse modo é o *Append*, inserido dentro dos parâmetros da declaração *with open* (mode = 'a'), que permite que uma nova linha seja simplesmente adicionada. A biblioteca fará a leitura de um dicionário com os novos valores a ser inseridos a partir da função *csv.DictWriter*, recebendo como parâmetros o arquivo aberto no modo *Append* e o cabeçalho e por fim, é utilizada a função *writerow()*, que fará de fato a inserção dos itens informados no arquivo de extensão .csv, sendo os parâmetros chave-valor que serão inseridos.

```
def dataUpdate():
    #ATUALIZAR NOVOS DADOS

    with open('./PBL_DENGUE/dados.csv', 'a', encoding='utf-8', newline='') as dataAppend:
        newData = csv.DictWriter(dataAppend, fieldnames=headers)
        newData.writerow({'Data': date, 'Bairro': district, 'Habitantes': population, 'Casos Suspeitos':suspects, 'Casos Confirmados': confirmeds, 'Casos Negativos': negatives})
```

Figura 11 – Adicionando novos dados.

2.1. Função extra

Nesta subseção será abordado sobre uma funcionalidade extra implementada no programa, que não consta dentro dos requisitos, porém visa oferecer uma melhor experiência de utilização do programa pelo usuário e tratar acidentes que podem ou não serem recorrentes.

Essa função baseia-se na alteração de um dado específico dentre os que já foram preenchidos e registrados no arquivo. Seguindo o padrão de modularização do código, é realizada a declaração de uma nova função, que tem início a partir da exibição de um menu, solicitando qual coluna ele deseja realizar a alteração (data, bairro, habitantes, casos suspeitos, casos negativos e casos confirmados). Após informado, será lido internamente o arquivo, portanto, não é necessário nenhum parâmetro nessa função. Lido o arquivo e computada a opção do usuário, tem-se em sequência um bloco de *try* e *except*,

que solicitará para o usuário a linha do arquivo na qual deseja realizar a alteração e qual o novo valor que deverá ser inserido.

```python
def editData():
    opt_edit = menu(['Data', 'Bairro', 'Habitantes', 'Casos Suspeitos', 'Casos Negativos', 'Casos Confirmados'])

    with open('./PBL_DENGUE/dados.csv', 'r', encoding='utf-8', newline='') as archive:
        report = csv.reader(archive)
        headers = next(report) #Cabeçalho
        data = [{header: value for header, value in zip(headers, row)} for row in report]

    column = headers[(opt_edit - 1)]

    try:
        line = int(input('\nInsira a linha que deseja alterar: '))
        value = str(input('Insira o dado atualizado: '))

    except:
        print('\nValor inválido!\n')
        return editData()
```

Figura 12 – Primeira parte da função

Na sequência, utiliza-se um bloco *match case*, para validar a opção inserida pelo usuário durante o primeiro menu exibido e entender qual coluna deseja-se fazer a atualização ou correção.

Para alteração das colunas data, bairro, habitantes e casos suspeitos utiliza-se uma abordagem simples, na qual é utilizado um *loop for* juntamente com a função *enumerate()*, que valida também a posição de cada item iterado, para percorrer o arquivo lido e encontrar a linha e coluna que o usuário deseja realizar a alteração, após definida (se existente), será salva em uma outra variável para que seja feita a atualização fora do *loop*. Quando encontrada a linha, é utilizada também uma declaração *break* para encerrar a busca, tornando mais eficaz o tempo de resposta. A alteração é refletida a partir do método *update()*, para atualizações de dicionários e por fim, como todos os dicionários estão dentro de uma lista, é feita a atualização da linha a partir do indíce, que é obtido no *loop for* usando a função *enumerate()*.

```
def editData():
    match opt_edit:

        case 1 | 2 | 3 | 4:

            for pos, item in enumerate(data):
                if pos + 2 == line:
                    line_to_update = pos
                    toReplace = item
                    break

            try:

                toReplace.update({column:value})
                data[line_to_update] = toReplace

            except:
                print('\nValor inválido!\n')
                return editData()
```

Figura 13 – *Match Case* e alteração simples

Para a alteração de casos confirmados e casos negativos, tem-se uma abordagem um pouco diferente e talvez um pouco mais complexa, já que é necessário fazer também a atualização dos casos suspeitos. Inicialmente, é utilizado o mesmo princípio da alteração dos outros dados, porém, para a linha em que deseja-se realizar a alteração, deve-se buscar também o número de casos suspeitos.

Após a obtenção de todos os dados necessários (dicionário que será realizado a alteração, números de casos negativos ou positivos – a depender da escolha do usuário – e o número de casos suspeitos, é realizada a diferença dos casos atuais e número de casos informado, em sequência, uma condicional para verificar se essa diferença é superior ao número de casos suspeitos, pois se for superior retornará um valor negativo para a quantidade de casos suspeitos e por fim é realizada a alteração nas duas colunas, da mesma maneira que para os outros dados.

```
def editData():
        case 5:

            greater = 0
            minor = 0

            for pos, item in enumerate(data):
                if pos + 2 == line:
                    line_to_update = pos
                    toReplace = item
                    negatives = int(item.get('Casos Negativos'))
                    suspects = int(item.get('Casos Suspeitos'))
                    break

            try:
                greater = max(negatives, int(value))
                minor = min(negatives, int(value))
                negatives_diff = greater - minor

                if negatives_diff > suspects:
                    print('\nValor para casos negativos inválido!\n')
                    return editData()

                else:
                    try:
                        suspects = suspects - negatives_diff
                        toReplace.update({column:value})
                        toReplace.update({'Casos Suspeitos':suspects})
                        data[line_to_update] = toReplace
                    except:
                        print('\nValor inválido!\n')
                        return editData()

            except ValueError:
                print('\nCasos negativos deve conter apenas números!\n')
                return editData()
```

Figura 14 – Tratamento e alteração dos valores

Em conclusão utiliza-se a declaração *close()* para fechar o arquivo, antes aberto em modo apenas para leitura, reaberto em modo para gravação e logo em seguida alterado os dados no arquivo e novamente utilizando *close()* para fechar e garantir o salvamento da alteração.

```python
def editData():
    archive.close()

    with open('./PBL_DENGUE/dados.csv', 'w', encoding='utf-8', newline='') as f:
        writer = csv.DictWriter(f, fieldnames=headers)
        writer.writeheader()
        writer.writerows(data)

        f.close()
```

Figura 15 – Modo de gravação e fechamento.

3. Resultados

Nesta seção de resultados será abordado como o usuário pode utilizar o programa, os devidos dados de entrada e respectivos dados de saída, bem como testes e erros encontrados e solucionados ao longo do processo de codificação do programa.

A navegação e visualização do sistema bem como de suas funções ocorre por meio apenas no terminal, ou seja, sem apresentar uma interface gráfica, que, embora esse modelo de interface não esteja presente, o programa ainda é bem intuitivo para navegação dado a forma como as opções são exibidas para o usuário e como funciona a interação entre ele (usuário) e o código (máquina).

Após iniciado o sistema, será exibido para o usuário 3 opções para início de navegação – “Sobre a Dengue”, “Dados” e “Sair”. A primeira opção exibe informações acerca do problema dengue para que o usuário entenda um pouco mais sobre do que se trata o *software*.

Na segunda opção, “Dados”, é a seção onde o usuário poderá visualizar os dados que foram gravados e atualizados até o momento, bem como ele pode atualizar esses dados. Logo após iniciar nessa sessão, serão exibidos para ele novas opções: “Visualizar Dados”, “Atualizar Dados” e “Voltar”. A última opção retorna para o menu inicial, enquanto a primeira opção abrirá um novo leque de opções, sendo elas “Visualização completa”, responsável por visualizar todos os dados registrados até o presente momento, incluindo os valores percentuais, “Visualização por data”, na qual o usuário pode visualizar os dados a partir de uma data informada por ele ou a partir de um intervalo de datas, “Visualização por bairro”, semelhante a visualização por data, porém para essa opção o usuário informa um bairro que deseja visualizar, que será exibida também a opção de exibição de valores percentuais por um bairro também informado. Terá também a opção “Voltar” que também retorna ao menu inicial.

Ainda dentro da sessão de “Dados”, a opção “Atualizar Dados” permite que o usuário faça o registro de novos dados (nova data, número de habitantes, bairro e quantidade de cada tipo de caso), bem como a alteração e/ou correção de algum dado já preenchido anteriormente no arquivo.

Por fim, a opção “Sair” do menu inicial encerra o programa e se despede do usuário.

Ao longo de todo o processo de codificação foi comum deparar-se com erros, que em grande maioria, foram erros de sintaxe e facilmente corrigidos. Outros erros que poderiam acontecer durante a execução do programa, como durante as opções solicitadas

para o usuário, foram tratados a partir de blocos com *try* e *except* e/ou funções para tratar os dados antes solicitados e obter o que realmente era esperado.

Sobre a entrada e saída de dados, a única variação relevante, em um cenário macro, é referente ao arquivo de extensão .csv, que após sofrer as atualizações e/ou novos registros, será permanentemente alterado, embora qualquer dado inserido corretamente possa ser corrigido em um segundo momento, a partir da função extra de edição de dados, abordada na subseção 2.1.

4. Conclusão

Esse relatório apresentou uma solução de *software* para o monitoramento dos casos de dengue, a partir da leitura e registro de novos dados. O sistema foi elaborado a partir do IDE e editor de códigos-fonte Visual Studio Code (VSCode), na versão 1.89.1, utilizando a linguagem imperativa Python na versão 3.12.3, com interface a partir do próprio terminal e sendo os dados armazenados em memória. A solução compreende o problema proposta e apresenta funcionalidade extra que torna sua utilização mais prática, a partir da utilização de linhas de comandos imperativas e estruturas, com definição de variáveis, comandos de entrada e saída, estruturas de repetição, estruturas de condicionais, contadores, acumuladores, importação de bibliotecas, manipulação de arquivos e processamentos a partir de cálculos e atribuições.

O relatório apresenta descrição de alto nível de todo o processo de codificação e das estruturas que o compõe, além de anexos ao longo de toda explanação que facilita e auxilia no entendimento. A solução baseia-se em funções que atendam a cada um dos requisitos solicitados durante a elaboração do projeto (abordados na introdução, seção 1 e na metodologia, seção 2), cada uma com sua própria especificidade para atingir seus respectivos objetivos atribuídos.

O programa atende ao problema proposto e às especificações solicitadas. Entretanto, apresenta uma grande limitação devido ao armazenamento dos dados ser em memória, além das necessidades de ler, sobrescrever e gravar novos arquivos, que são cumpridas também em memória e pode gerar maior tempo de resposta ou lentidão se a base de dados carregada for muito extensa. A partir de versões futuras pode-se adicionar mais flexibilidade e menor tempo de resposta, como a partir da implementação de um banco de dados para tornar o armazenamento e carregamento de informações mais eficaz, bem como a utilização de uma interface gráfica, que torna a interação entre o usuário e máquina mais eficiente e satisfatória.

Esse relatório apresenta robustez em sua estruturação devido a organização detalhada do problema em funções, testes e tratamento de erros. Ao final desta explanação pode também ter ocorrido problemas de redação que reduzam a clareza e a concisão com que as informações são apresentadas.

Referências

John V. Guttag. Introduction to Computation and Programming Using Python. Revised and expanded edition. MIT Press, 2013.