Curso de Computação 1
Introdução à Programação em Python

AULA 11

Modularização

Chegamos enfim ao final deste primeiro curso de programação. É chegado o momento de refletir sobre como os conceitos, técnicas e recursos de programação que vimos ao longo do curso podem nos ser úteis para construir soluções computacionais para problemas. Veremos um recurso simples para fazer com que um programa em Python seja executado automaticamente, e revisaremos os passos para construção de soluções computacionais usando a estratégia de programação modular e o padrão de arquitetura em camadas.

Chamada da função *main*

Talvez você já tenha percebido que podemos colocar a chamada de uma função diretamente dentro de um arquivo `.py`. Quando este arquivo for interpretado, a função será chamada diretamente. Em geral, fazemos isso com a função `main`, para que ela seja executada diretamente quando mandamos executar um arquivo na interface do editor do IDLE. Existe porém uma maneira mais adequada para escrever a chamada da função `main` dentro de um arquivo. Veja como isso é feito no código abaixo:

```
def soma(numero1,numero2):
  """Funcao que soma dois numeros inteiros
  Parametro de entrada: int,int
  Valor de retorno: int"""
  return numero1 + numero2

def main():
  """Funcao Principal"""

  print(str.format("A soma de 2 e 3 e: {0:3d}",soma(2,3)))

if __name__ == "__main__":
  main()
```

Neste exemplo, temos a definição da função `soma` (nossa velha conhecida) e também de uma função `main`, que faz uso da função `soma`. Repare nas duas linhas de código que estão após a função `main`.

```
if __name__ == "__main__":
  main()
```

Temos a chamada da função `main` dentro de uma estrutura `if`. Essa é uma forma padrão de escrita de chamada de função `main` no Python. Os identificadores que começam com dois

caracteres sublinhado (underline) tem um papel especial no Python. Vamos tentar entender qual é o efeito desse padrão específico.

Temos que lembrar que um código Python pode ser usado como uma biblioteca, ou seja, um módulo, que pode ser usado por outros programas. Sim, seu código pode vir a ser "importado" como um módulo. O padrão `__name__ == "__main__"` é usado para diferenciar estas duas situações: (1) a execução direta, e (2) a importação. Ele é uma maneira de se fazer a pergunta: *Quem está executando o código ? É o interpretador diretamente, ou é outro módulo?*

Vamos seguir com exemplos:

Considere que você salvou as funções `soma` e `main` no arquivo `MeuPrograma.py`

Suponha que agora você fez a função `vezes5`, salvando-a no arquivo `Auxiliar.py`. Note que ao invés de copiar a função *soma* do arquivo `MeuPrograma.py` para `Auxiliar.py`, apenas importamos as funções de `MeuPrograma.py` para `Auxiliar.py`

```
from MeuPrograma import *

def vezes5(numero1,numero2):

  """Multiplica por 5 o resultado da funcao soma
  Parametros de entrada: int,int
  Valor de retorno: int"""

  return 5*soma(numero1,numero2)
```

De forma que:

- Ao definir a função `main` em `MeuPrograma.py`, estamos requerendo que os comandos dentro dela sejam executados ao executarmos este arquivo.
- O comando `if __name__ == "__main__":` serve para verificar se estamos ou não rodando `MeuPrograma.py` diretamente.
- Se estivermos, o teste do `if` será **`True`** e a função `main` será executada.
- É isso que ocorre quando executamos `MeuPrograma.py`.
- Já quando estamos executando `Auxiliar.py`, importamos as funções do arquivo `MeuPrograma.py`.
- Quando importamos `MeuPrograma.py` durante a execução de `Auxiliar.py`, o teste `if __name__ == "__main__":` não será satisfeito (será **`False`**), uma vez que quem está sendo executado é `Auxiliar.py` e não `MeuPrograma.py`.
- Logo a função main definida em `MeuPrograma.py` não será executada.

Curso de Computação 1
Introdução à Programação em Python

Projeto e desenvolvimento de aplicações modulares

A programação permeia muitas áreas atualmente. O nível de sofisticação esperado é elevado. Programar não é uma tarefa fácil, o que significa que, para construir código, há um investimento sendo feito (do tempo do programador, que pode ser apenas você ou toda uma equipe). Para justificar esse investimento, um trecho de código deve ser reutilizável. Por ser entendida como uma prática importante e eficiente, atualmente várias pessoas compartilham seus códigos, criando comunidades.

Atenção! A **organização** e a **legibilidade** do código são requisitos essenciais, tão importantes quanto eficiência e eficácia. Ter seu código bem documentado também é importante para que você mesmo consiga utilizá-lo futuramente. É muito fácil esquecer o que um código faz, e gasta-se tempo para tentar entendê-lo novamente.

Para atender às demandas atuais, cada vez mais complexas e sofisticadas, o programador deve pesquisar sobre bibliotecas disponíveis que o auxiliem. Tão importante quanto saber programar tudo o que precisa, é saber aproveitar o que está disponível, ou fica inviável atender às demandas em tempo aceitável.

É importante ser capaz de ler e entender a documentação de tais bibliotecas. Um bom conhecimento da linguagem de programação em que a biblioteca foi escrita ajuda muito.

Uma vez que se produza código reutilizável e de boa qualidade, é de bom tom compartilhar! A comunidade agradece e reconhece o esforço.

Vamos explorar a seguir alguns exemplos. Fique atento às opções de projeto de cada um deles, e como essas opções impactam na organização e legibilidade do código produzido.

Curso de Computação 1
Introdução à Programação em Python

Exemplo 1: Sorteador

Na aula anterior, vimos o exemplo do programa Sorteador, que realizava um sorteio e mostrava seus resultados ao usuário. Na versão vista anteriormente, o programa possuía 2 funções. A primeira delas, a função *sortear*, recebia 2 parâmetros N e X, onde N e X são números inteiros que representam, respectivamente, quantos números deveriam ser sorteados e qual o máximo do intervalo [1,X], ou seja, até que número poderia ser sorteado. A outra função era a função principal, *main*, que se encarregava de realizar toda a interação com o usuário. Nela, era pedido que o usuário informasse os valores de N e X, que eram então repassados como argumentos à função *sortear*, que por sua vez retornava uma lista com os números sorteados, e esses números eram então formatados para serem apresentados ao usuário como resultado do sorteio. Veja o código da função *main* abaixo:

```
def main():
        '''Funcao principal: Programa Sorteador
        None -> None'''

        #Solicitando definicoes do sorteio ao usuario
        print("-------- Programa Sorteador --------")
        n = int(input("Quantos numeros deverao ser sorteados? \n"))
        x = int(input("Ate que numero pode ser sorteado (De 1 a ...)? \n"))

        #Chamada da funcao que faz o sorteio
        lista_sorteados = sortear(n,x)

        #Impressao dos resultados do sorteio
        print("-------- Resultado do sorteio --------")
        print(str.format("Foram sorteados {} numeros", len(lista_sorteados)))

        for i in range(len(lista_sorteados)):
                print(str.format("O {}o numero sorteado foi: {}", i+1, lista_sorteados[i]))
```

Observe que só são solicitados dados ao usuário antes da chamada da função *sortear*. Da mesma forma, somente após a execução da função *sortear* é que é realizado o tratamento e formatação dos números sorteados para serem apresentados ao usuário. A partir do conceito de modularização, podemos reescrever a função *main* ao se criar duas novas funções: uma para a solicitação de dados ao usuário, e outra para a impressão dos resultados do sorteio. Dessa forma, cada funcionalidade do programa será executada por uma função específica, e a função *main* será responsável apenas por orquestrar a chamada de cada uma das funções nos momentos apropriados. Caso seja desejado alterar alguma funcionalidade do programa, será necessário fazer a alteração apenas na função responsável por essa funcionalidade, não impactando o código das outras funções. Veja abaixo o código completo da versão modularizada do programa Sorteador (código também disponível no material da aula na turma):

Curso de Computação 1
Introdução à Programação em Python

```python
# Programa sorteador:  Versão modularizada
def define_sorteio():
        '''Funcao que solicita ao usuario quantos numeros devem ser
        sorteados e ate qual numero pode ser sorteado.
        None -> tuple(int, int)'''

        print("-------- Programa Sorteador --------")
        n = int(input("Quantos números deverão ser sorteados? \n"))
        x = int(input("Até que número pode ser sorteado (De 1 a ...)? \n"))

        return n, x


from random import randint
def sortear(n, x):
        '''Funcao que sorteia n numeros aleatorios entre 1 e x
        int, int -> list'''

        lista_sorteados = []
        while len(lista_sorteados) < n:
                numero = randint(1,x)
                if numero not in lista_sorteados:
                        list.append(lista_sorteados, numero)

        return lista_sorteados


def imprime_resultado(lista_sorteados):
        '''Funcao que imprime o resultado do sorteio.
        list -> None'''

        print("-------- Resultado do sorteio --------")
        print(str.format("Foram sorteados {} numeros", len(lista_sorteados)))

        for i in range(len(lista_sorteados)):
                print(str.format("O {}° numero sorteado foi: {}", i+1, lista_sorteados[i]))


def main():
        '''Funcao principal do Programa Sorteador
        None -> None'''

        n, x = define_sorteio()
        lista_sorteados = sortear(n, x)
        imprime_resultado(lista_sorteados)


if __name__ == '__main__':
        main()
```

Apesar de este programa ser simples, repare que esta versão ficou um pouco mais organizada e legível que a anterior.

Curso de Computação 1
Introdução à Programação em Python

Exemplo 2: Laboratório de física

Na aula passada desenvolvemos um programa para nos ajudar a construir um relatório sobre um experimento de Física.

Inicialmente desenvolvemos algumas funções para cálculo de valores esperados e cálculo dos erros entre os valores observados e valores esperados. As funções `velocidade_esperada`, `calcula_esperadas`, `erro_do_experimento` e `erro_ao_logo_do_tempo`, cuidam dessa parte do trabalho e devem ficar agrupadas em um só arquivo.

Em uma primeira versão, criamos uma função `main` que coordenava a execução dessas funções para a obtenção de uma lista contendo os erros de cada observação em cada instante. Mas observamos que essa solução não era satisfatória pois obrigava e definir explicitamente na função principal todos os dados do experimento, sendo necessário editá-la para quaisquer alterações.

Para resolver esse problema, propusemos uma nova versão, completada por você, onde os dados de entrada do experimento e os valores observados eram solicitados do usuário a cada execução do programa. Além disso, apresentamos o resultado de uma maneira mais amigável, com informações sobre as velocidades esperadas em cada instante, as observadas, e os erros correspondentes. Toda essa comunicação com o usuário foi implementada diretamente na função principal. A versão completa da função principal com toda a interação com o usuário se encontra na figura a seguir.

Curso de Computação 1
Introdução à Programação em Python

```
def main():
    ''' programa principal para realização dos cálculos do laboratório de cinemática'''

    # dados do experimento
    # pedir para o usuário fornecer os dados do experimento
    #Fornecimento da velocidade inicial
    velocidade_inicial = float(input('Velocidade inicial (m/s): '))

    #Fornecimento da aceleração
    aceleracao = float(input('Aceleração constante (m/s²): '))

    tempos = []
    tempo_str = '...'
    print('Tempos (Digite enter quando terminar):')
    while tempo_str != '':
        tempo_str = input(' - Tempo (s): ')
        if tempo_str:
            list.append(tempos, float(tempo_str))
    qtd_observacoes = len(tempos)

    # Calcula a velocidade esperada
    velocidades_esperadas = calcula_esperadas(velocidade_inicial, aceleracao, tempos)

    # resultados observados
    # pedir para o usuário fornecer os resultados observados
    velocidades_observadas = []
    print(str.format('Velocidades observadas (Espera-se {} velocidades):', qtd_observacoes))
    i = 0
    while i < qtd_observacoes:
        velocidade_str = input(str.format(' - Velocidade (m/s) em {} s: ', tempos[i]))
        if velocidade_str !='':
            list.append(velocidades_observadas, float(velocidade_str))
            i += 1
        else:
            print(str.format('Ainda faltam {} velocidades.', qtd_observacoes - i))

    # cálculo dos erros
    erros = erro_ao_longo_do_tempo(velocidades_esperadas, velocidades_observadas, tempos)

    # Saída para o usuário
    strings_velocidades_esperadas = []
    for velocidade in velocidades_esperadas:
        string_velocidade_esperada = str.format('{:8.2f}', velocidade)
        list.append(strings_velocidades_esperadas, string_velocidade_esperada)

    strings_velocidades_observadas = []
    for velocidade in velocidades_observadas:
        string_velocidade_observada = str.format('{:8.2f}', velocidade)
        list.append(strings_velocidades_observadas, string_velocidade_observada)

    strings_erros_em_porcentagem = []
    for erro in erros:
        erro_em_porcentagem, tempo = erro
        string_erro_em_porcentagem = str.format('{:7.2f}%', erro_em_porcentagem)
        list.append(strings_erros_em_porcentagem, string_erro_em_porcentagem)

    print(str.format('\n\nComparação de velocidades (v0 = {} m/s; a = {} m/s²):', velocidade_inicial, aceleracao))
    print(str.format(' - Velocidades esperadas  (m/s): [{}]', str.join(', ', strings_velocidades_esperadas)))
    print(str.format(' - Velocidades observadas (m/s): [{}]', str.join(', ',strings_velocidades_observadas)))
    print(str.format(' - Erros                   (%): [{}]', str.join(', ', strings_erros_em_porcentagem)))
```

Mas se observamos bem essa função principal, vemos que toda a parte de cálculo dos erros fica perdida no meio de tantos detalhes de interação com o usuário. Isso acontece porque

esses dois tipos de ação estão sendo tratados em níveis de abstração diferente. Se criarmos funções também para cuidar da interação, como criamos para os cálculos, o algoritmo para a solução de nosso problema de preparação de um relatório ficará mais visível. Podemos então criar um novo módulo apenas para cuidar das ações de entrada e saída de dados. Nosso objetivo é ficar com a seguinte função principal:

```
def main():
    ''' programa principal para realização dos cálculos do laboratório de cinemática'''

    # dados do experimento
    # pedir para o usuário fornecer os dados do experimento
    velocidade_inicial = pede_float('Velocidade inicial (m/s): ')
    aceleracao = pede_float('Aceleração (m/s²): ')
    tempos = pede_instantes_do_experimento()
    qtd_observacoes = len(tempos)

    # cálculo das velocidades esperadas em cada instante
    velocidades_esperadas = calcula_esperadas(velocidade_inicial, aceleracao, tempos)

    # resultados observados
    # pedir para o usuário fornecer os resultados observados
    velocidades_observadas = pede_velocidades_observadas(tempos)

    # cálculo dos erros
    erros = erro_ao_longo_do_tempo(velocidades_esperadas, velocidades_observadas, tempos)

    # Saída para o usuário
    mostra_relatorio(velocidade_inicial, aceleracao, tempos, velocidades_esperadas, velocidades_observadas, erros)
```

Fica bem mais fácil de ver e entender o que está acontecendo, não é?

Para que essa nova função principal tenha o mesmo comportamento da função original, vamos agora criar esse módulo de entrada e saída e completar o programa. Precisamos definir as três funções que solicitam dados do usuário:
`pede_float`,
`pede_instantes_do_experimento`, e
`pede_velocidades_observadas`.
E precisamos definir a função que constrói o relatório final: `mostra_relatorio`.

A seguir apresentamos as funções de entrada. Observe como em alguns casos podemos criar parâmetros para poder reaproveitar uma mesma função em mais de um ponto do programa, como foi o caso da função `pede_float`. Observe também que todas elas possuem um retorno. Afinal de contas, o papel delas é obter informação do usuário para uso pelo programa, como é o caso da função `input`. Então, é essencial que a função retorne o valor ou valores lido(s), eventualmente após algum tipo de validação ou transformação, como fizemos em `pede_float`.

Curso de Computação 1
Introdução à Programação em Python

```
def pede_float(pedido):
    '''pede um número float até que ele seja dado
    str --> str'''
    while True:
        float_string = str.strip(input(pedido))
        if float_string != '':
            return float(float_string)

def pede_instantes_do_experimento():
    '''solicita do usuário os instantes onde foram realizadas as observações e constrói uma lista
    com essa informação
    None --> list'''
    tempos = []
    tempo_str = '...'
    print('Tempos (Digite apenas ENTER quando terminar):')
    while tempo_str != '':
        tempo_str = str.strip(input(' - Tempo (s): '))
        if tempo_str != '':
            list.append(tempos, float(tempo_str))
    return tempos

def pede_velocidades_observadas(tempos):
    '''solicita do usuário as n velocidades observadas em cada instante e constrói uma lista
    com essa informação.
    int --> list'''
    velocidades_observadas = []
    n = len(tempos)
    print(str.format('Velocidades observadas (Espera-se {} velocidades):', n))

    i = 0
    while i < n:
        velocidade_str = input(str.format(' - Velocidade (m/s) em {} s: ', tempos[i]))
        if velocidade_str != '':
            list.append(velocidades_observadas, float(velocidade_str))
            i += 1
        else:
            print(str.format('Ainda faltam {} velocidades.', len(tempos) - i))

    return velocidades_observadas
```

Agora vamos ver a função `mostra_relatório`. Ela recebe como argumentos todos os dados do problema para usá-los na apresentação e não retorna nada. Apenas faz a formatação e apresentação do relatório. Para tal, usa duas funções de formatação das linhas do relatório. Como as partes de formatação das linhas correspondentes às velocidades eram idênticas, uma mesma função atende os dois casos. A formatação dos erros era ligeiramente diferente, ficando com uma função própria,

Curso de Computação 1
Introdução à Programação em Python

```
def formata_velocidades(velocidades):
    '''formata todas as velocidades para a impressão do relatório.
    list --> list'''
    strings_velocidades = []
    for velocidade in velocidades:
        string_velocidade = str.format('{:8.2f}', velocidade)
        list.append(strings_velocidades, string_velocidade)
    return strings_velocidades

def formata_erros(erros):
    '''formata todos os erros em porcentagem para a impressão do relatório.
    list --> list'''
    strings_erros = []
    for erro in erros:
        erro_em_porcentagem, tempo = erro
        string_erro_em_porcentagem = str.format('{:7.2f}%', erro_em_porcentagem)
        list.append(strings_erros, string_erro_em_porcentagem)
    return strings_erros

def mostra_relatorio(velocidade_inicial, aceleracao, tempos, velocidades_esperadas, velocidades_observadas, erros):
    '''apresenta os dados do problema por linhas, procurando manter um formato tabulado, para facilicar a
    leitura do relatório
    float, float, list, list, list, list --> None'''
    strings_velocidades_esperadas = formata_velocidades(velocidades_esperadas)
    strings_velocidades_observadas = formata_velocidades(velocidades_observadas)
    strings_erros_em_porcentagem = formata_erros(erros)

    print(str.format('\n\nComparação de velocidades (v0 = {} m/s; a = {} m/s²):', velocidade_inicial, aceleracao))
    print(str.format(' - Velocidades esperadas  (m/s): [{}]', str.join(', ', strings_velocidades_esperadas)))
    print(str.format(' - Velocidades observadas (m/s): [{}]', str.join(', ',strings_velocidades_observadas)))
    print(str.format(' - Erros                   (%): [{}]', str.join(', ', strings_erros_em_porcentagem)))
```

A solução final então, fica com 3 arquivos: um primeiro contendo a parte dos cálculos, um segundo contendo as funções de entrada e saída de dados, e um terceiro, que importa os 2 primeiros e contém apenas a função principal.

Digamos agora que queremos um relatório mais sofisticado, podemos trocar importar uma nova função `mostra_relatorio` que apresente os resultados de maneira diferente. Para fazer essa mudança precisamos apenas:

- Ter a definição da nova função `mostra_relatorio` (definida por você mesmo ou por outra pessoa), e importar o arquivo que a contenha.
- Caso se queira usar uma função com outro nome, basta mudar a exata linha onde a função `mostra_relatorio` é chamada dentro da função `main`.

Nos códigos disponibilizados junto com este roteiro está uma versão alternativa para esta função, que mostra o relatório em um formato de tabela. Faça o teste para ver como o relatório fica diferente!

Exemplo 3: Matrizes

Neste exemplo, temos um pequeno programa construído para somar ou subtrair duas matrizes dadas pelo usuário e o código do mesmo foi apresentado na aula passada em um único arquivo (`Matrizes.py`). Este arquivo foi novamente fornecido com o roteiro desta aula. Baixe e abra o arquivo para acompanhar a descrição fornecida abaixo.

A organização proposta foi criar cinco funções, além da função principal, cada uma cobrindo uma tarefa específica, sendo que duas dessas funções seriam funções responsáveis pela entrada - `escreveMatriz` - e saída - `imprimeMatriz`. Perceba que ambas funções são chamadas mais de uma vez no código.

```
def escreveMatriz(matriz):
    '''escreve cada elemento da matriz a partir de entrada do usuario
    list->None'''
    for i in range(len(matriz)):
        for j in range(len(matriz[i])):
            matriz[i][j] = float(input('Valor de a('+str(i)+','+str(j)+') = '))

def imprimeMatriz(matriz):
    '''imprime matriz na saida padrao
    list->None'''
    for linha in matriz:
        print('|',end='\t')
        for aij in linha:
            print(str.format('{:.2f}',aij),end='\t')
        print('|')
```

Isso ilustra como blocos recorrentes são bons candidatos a serem transformados em função. A transformação desses blocos em funções é benéfica para a organização do código porque melhora a legibilidade, além de evitar possíveis erros propagados por meio de ações de copiar e colar. A manutenção do código também fica mais simples, já que quando um erro é detectado, há apenas um lugar onde o código deve ser alterado.

```
def criaMatriz(linhas,colunas):
    '''cria uma matriz nula com dimensoes linhas x colunas
    int,int->list'''
    matriz = []
    for i in range(linhas):
        list.append(matriz,colunas*[0])
    return matriz
```

Das outras três funções restantes, uma delas é a função `criaMatriz`. Esta é chamada em quatro oportunidades: para criar cada matriz de entrada e também a matriz resultado da soma e da subtração. E completando, existem as duas funções de `soma` e `subtração` que implementam o cerne das funcionalidades que planejamos ter quando pensamos neste programa. Conforme vimos anteriormente, separar as funções de serviço das funções de interface é uma boa estratégia de organização.

```
def somaMatriz(A,B):
    '''retorna a soma A+B de duas matrizes
    list,list->list'''
    soma = criaMatriz(len(A),len(A[0]))
    for i in range(len(soma)):
        for j in range(len(soma[i])):
            soma[i][j] = A[i][j] + B[i][j]
    return soma

def subtraiMatriz(A,B):
    '''returna a subtracao A-B de duas matrizes
    list,list->list'''
    sub = criaMatriz(len(A),len(A[0]))
    for i in range(len(sub)):
        for j in range(len(sub[i])):
            sub[i][j] = A[i][j] - B[i][j]
    return sub
```

Por fim, a última função deste exemplo é a função principal que segue uma organização em três blocos: inicialização de variáveis, opções de menu e execução da opção escolhida. Eventualmente, alguns destes blocos poderiam ser candidatos a se tornarem funções, em especial o caso do bloco que mostra as opções de menu. Entretanto foi feita uma opção por não criar tais funções, seja porque não são blocos recorrentes em código ou porque seria difícil imaginar um contexto de reuso da função que implementasse um desses blocos. Em casos assim, a escolha entre criar ou não criar uma função varia de acordo com as preferências do programador sobre como melhor organizar seu código.

Atividade (sem entrega): divida o código do exemplo de soma e subtração de matrizes, de maneira a criar um módulo apenas com as funções de serviço. Você deve fazer a importação desse módulo no arquivo que contém a função principal. Faça esta alteração sem que seja necessário alterar o código da função principal. Após esta reorganização do código, faça um teste de maneira a garantir que a função principal continua executando corretamente.

Prática em Programação

Após concluir as etapas anteriores deste roteiro, faça as atividades práticas desta aula, disponíveis no Google Classroom da turma.
Esperamos que tenha gostado do curso e que as habilidades que você desenvolveu lhe sejam úteis!