

TTI103

Lógica de Programação

Aula T09

Modularização

Visão geral

- Programação em módulos
 - técnica de desenvolvimento de software que enfatiza a separação das funcionalidades de um programa em módulos independentes e reutilizáveis
- Cada módulo deve conter tudo o que é necessário para executar apenas um único aspecto da funcionalidade desejada



Definições

- Modularizar significa que um sistema complexo é dividido em partes ou componentes menores, ou seja, módulos.
- Esses componentes podem ser criados e testados independentemente e, em muitos casos, podem ser usados em outros sistemas.
- Esse conceito não é novo, desde os anos 60 as linguagens tinham subprogramas, macros, sub-rotinas, procedimentos, módulos, funções etc.
- No paradigma orientação a objetos, tem-se os métodos que implementam comportamentos bem definidos de uma classe
- Módulos são importantes porque permitem dividir programas grandes e complexos em partes menores e gerenciáveis:
 - Como são menores, é possível concentrar no que é necessário que façam e testá-los para garantir que funcionem corretamente

Princípio número 1 do desenvolvimento de software: alta coesão - baixo acoplamento

- O objetivo da quebra é obter módulos sem ou com poucas dependências de outros módulos
 - a minimização das dependências é o objetivo
- Ao criar um sistema modular, vários módulos são construídos separadamente e mais ou menos de forma independente
- Vários aplicativos diferentes e funcionais poderão ser criados a partir da reutilização deles

Módulos em Python?

- Em poucas palavras: todo arquivo com a extensão `.py` e código Python adequado, pode ser um módulo!
- Não há sintaxe especial necessária para tornar esse arquivo um módulo.
- Um módulo pode conter objetos arbitrários, classes ou atributos.
 - Todos esses objetos podem ser acessados após uma importação.
 - Existem diferentes maneiras de importar um módulo.

Algumas funções definidas dentro do próprio **Python** que você já utilizou

- Entrada e saída
 - input e print
- Conversão de tipo
 - int, float e str
- Cálculos matemáticos
 - abs, math.sqrt, math.cos, math.ceil e math.pow

Criando nossas próprias funções

- Objetivo
 - reaproveitar, sempre que necessário, o mesmo **bloco de operações**.
- As principais vantagens no uso de funções são
 - Códigos mais legíveis, especialmente para programas grandes
 - Manutenções e modificações no código são mais fáceis, com menor chance de prejudicar o que já foi feito
 - Possibilidade de reaproveitamento da mesma estrutura em diversas seções do código-fonte

Criando nossas próprias funções

- Funções são blocos que tomam argumentos de entrada (input), realizam operações e retornam um valor de saída (output)
- As entradas e saídas podem ser variáveis de qualquer tipo
- Para que o Python reconheça um bloco como uma função, ele precisa estar organizado e endentado.
- O comando para definir funções é *def*

Sintaxe - parte 1

```
def <nome da função> (<argumentos>):  
    comando 1  
    comando 2  
    ...  
    return <resultado>  
    ...  
comando n  
return <resultado>
```

A palavra **def** determina que será definida uma função, que será composta de:

- **<nome da função>**: assim como as variáveis criadas pelo programador, as funções precisam ter um nome.
 - As regras definidas são as mesmas das variáveis
- **<argumentos>**: determinam nomes de variáveis internas da função, automaticamente criadas quando a função é chamada, responsáveis por receber os valores que devem ser fornecidos como entrada para a função.
 - Uma função pode não receber argumento algum, ou um número arbitrário de argumentos
- A utilização dos parênteses delimitando os argumentos e do caractere “:” após o fechamento dos parênteses é obrigatória, assim como a endentação do código que representa o corpo da função.

Sintaxe - parte 2


```
def <nome da função> (<argumentos>):  
    comando 1  
    comando 2  
    ...  
    return <resultado>  
    ...  
    comando n  
    return <resultado>
```

A palavra **def** determina que será definida uma função, que será composta de:

- **o corpo da função:** determina a sequência de comandos que realiza a tarefa necessária e gera os resultados esperados. É composto por
 - **comandos**, que realizam a tarefa
 - cláusulas **return**, que encerram a execução da função e retornam os resultados de saída gerados
 - Esses resultados são repassados para o *chamador* da função. Se a função não precisa retornar nada, a cláusula **return** não precisa ser usada, a menos que se deseje explicitamente definir o término da sua execução em algum momento.
 - Quando essa cláusula não é usada, o término da execução ocorre da mesma forma que em um programa, ou seja, quando seu último comando é executado, e nada é retornado

Chamada de Funções

- Para chamar uma função no **Python**, basta escrever o nome dela dentro do código, com os argumentos entre parêntesis.

 chamada.py > ...

```
1  # Definição da função
2  def uma_funcao(x):
3      y = 2*x + 3
4      return y
5
6  # Programa principal
7  y1 = uma_funcao(10)
8  y2 = uma_funcao(20)
9  y3 = uma_funcao(30)
10 print(f'y1 = {y1}, y2 = {y2}, y3 = {y3}!')
```

Escopo de variáveis - parte 1

- Variáveis criadas no programa principal e dentro das funções possuem o que se chama de escopos diferentes
- O escopo de uma variável é onde ela existe, ou seja, onde ela pode ser reconhecida e utilizada
- Os argumentos de uma função e as variáveis criadas dentro dela estão exclusivamente dentro do escopo da função, ou seja, só existem e podem ser referenciadas ali dentro, pois não existem fora dela
 - Denomina-se esse escopo como local, pois as variáveis só existem localmente na função onde foram definidas
- As variáveis definidas no programa principal possuem escopo global, ou seja, existem em todo o programa, incluindo as funções definidas dentro dele.
 - **Lembre-se: uma variável só passa a existir após ter algum valor atribuído a ela**

Escopo de variáveis - parte 2

- Boa prática de programação: evitar o uso de variáveis globais dentro de funções. Embora acessíveis, recomenda-se que
 - Valores necessários para a execução de uma função sejam passados como argumentos de entrada
 - Valores definidos dentro da função, que sejam tratados como valores de retorno da função
 - Não se acessa ou se define valores de variáveis globais dentro de uma função
- Cuidado: Variáveis de escopos diferentes podem possuir o mesmo nome, sem embaralhar os seus respectivos valores
 - Variáveis globais coexistem com variáveis locais de mesmo nome
 - Porém, dentro de uma função prevalecem as variáveis locais, ou seja, as variáveis globais de mesmo nome não são acessíveis.

Exemplo para ilustrar o escopo de variáveis

 exemplo_escopo1.py > ...

```
1
2 def soma(num1, num2):
3     resultado = num1 + num2
4     return resultado
5
6 def subtracao(num1, num2):
7     resultado = num1 - num2
8     return resultado
9
10 # Programa principal
11 num1 = float(input('Digite o primeiro valor: '))
12 num2 = float(input('Digite o segundo valor: '))
13 resultado = soma(num1, num2)
14 print(f'A soma é {resultado}')
15 resultado = subtracao(num1, num2)
16 print(f'A subtração é {resultado}')
```

- Temos inúmeras variáveis de mesmo nome, mas escopos diferentes
 - Temos 3 variáveis que se chamam num1, no programa principal e nas duas funções.
 - Isso significa que em alguns momentos do fluxo de execução do programa, teremos duas variáveis num1
 - Isso porque as variáveis locais deixam de existir quando as funções que as declararam são finalizadas. Porém, temos a variável global num1, que poderá então coexistir com uma das variáveis de mesmo nome das duas funções.
 - O mesmo acontece com a variável resultado

Retorno de vários valores

- Eventualmente é necessário que uma função retorne mais que um valor como resultado
- Para fazer isso em Python, basta colocar todos os valores separados por vírgula na cláusula *return* e considerar essa questão na chamada da função
- O exemplo clássico: ler os coeficientes *a*, *b*, *c* de uma equação do segundo grau, calcular e exibir as suas raízes.

```
import math
```

```
def delta(a, b, c):  
    d = (b**2) - (4*a*c)  
    return d
```

```
def raizes(a, b, c):  
    d = delta(a,b,c)  
    if (d < 0):  
        return None, None  
    else:  
        x1 = (-b + (math.sqrt(d)))/(2*a)  
        x2 = (-b - (math.sqrt(d)))/(2*a)  
        return x1, x2
```

```
# principal - entrada  
a = input('Digite o valor do coeficiente a: ')  
b = input('Digite o valor do coeficiente b: ')  
c = input('Digite o valor do coeficiente c: ')  
# principal - processamento  
r1, r2 = raizes(a,b,c)  
# principal - saída  
if (r1 != None):  
    print(f'Raízes: {r1:.2f} e {r2:.2f}')
```

```
else:  
    print('Não existem raízes reais')
```

Vamos praticar!!!

TTI103

Lógica de Programação

Aula T09

Modularização