

Profa. Dra. Raquel C. de Melo-Minardi  
Departamento de Ciência da Computação  
Instituto de Ciências Exatas  
Universidade Federal de Minas Gerais

---



# MÓDULO 2 – PROGRAMAÇÃO

## Modularização de código

# MODULARIZAÇÃO DE CÓDIGO

- ▶ A **programação modular** é um conjunto de técnicas de projeto e organização de código que visa **separar as diferentes funcionalidades** de um sistema em **módulos independentes e intercambiáveis**
- ▶ Normalmente pensamos nos módulos para que contenham todo o código necessário para implementação de uma funcionalidade
- ▶ Esse conceito é normalmente relacionado aos conceitos de programação estruturada e programação orientada por objetos que definimos a seguir
- ▶ **Programação estruturada**: estruturação do fluxo de controle do programa tendo em vista a codificação (nível um pouco mais baixo)
- ▶ **Programação orientada por objetos**: uso de objetos de dados, um tipo de estrutura de dados

**Programação modular** é a decomposição em alto nível de um software em partes. Ela se diferencia da programação estruturada por essa quebra em módulos que normalmente são organizados internamente de forma estruturada. A programação orientada por objetos é modular mas nem toda programação modular é orientada por objetos

- ▶ Em Python, podemos modularizar código usado três diferentes mecanismos que permitem organizar o sistema em bibliotecas e funcionalidades

**Biblioteca** é uma coleção de implementações em alguma linguagem de programação e que pode ser utilizada por múltiplos programas que não necessariamente tenham relação uns com os outros possibilitando o que chamamos de reuso

**Funcionalidade** é um conceito mais restrito é define uma tarefa indivisível do sistema e que normalmente possui uma interface bem definida composta por argumentos de entrada e valor(es) de retorno ou saída

## VANTAGENS

- ▶ **Menos código** a ser escrito
- ▶ Código escrito será mais **simples e compreensível**
- ▶ O **escopo** das variáveis é **limitado** e pode ser facilmente controlado
- ▶ O código fica **organizado** em arquivos separados de acordo com sua função
- ▶ Maior facilidade de **projeto** e **divisão de trabalho** entre equipes de desenvolvimento
- ▶ Uma funcionalidade em desenvolvimento pode ser **reusada**, eliminando a necessidade de se escrever o código múltiplas vezes
- ▶ Os **erros** são mais facilmente identificados, pois são localizados dentro de subrotinas
- ▶ O software é mais fácil de **manter** e **estender**

# COMO IMPLEMENTAR MODULARIZAÇÃO EM PYTHON

- ▶ Em Python, implementamos uma biblioteca através dos seguintes mecanismos
  - ▶ Módulo
  - ▶ Subrotinas
  - ▶ Classes



**Módulo** que é um arquivo de código fonte, normalmente contendo diversas funcionalidades de semântica relacionada

**Exemplos:** módulo de cálculos de medidas de média estatística, módulo de tratamento de sequências, módulo de sobreposição de estruturas, etc



**Subrotinas**, também denominadas **procedimentos** ou **funções**, consistem em blocos de código que podem ser ativados (desviando o fluxo de execução do programa para si) quando necessário

**Exemplos:** subrotina para cálculo de média aritmética de um conjunto de valores, subrotina para alinhamento de um par de sequências, subrotina para obtenção de uma matriz de translação

Uma subrotina é um termo genérico que pode se diferenciar segundo sua estrutura e utilidade:

- ▶ **Procedimento:** normalmente realiza uma tarefa mas não retorna um valor
- ▶ **Função:** usualmente retorna um valor ou um conjunto de valores. É análoga a uma função matemática na qual um valor é recebido na entrada e outro, gerado na saída

# SUBROTINAS

- ▶ Veja a seguir um exemplo de definição e respectiva chamada a subrotina em Python:

```
# Definição da função
def minhaFuncao(a, b, c):
    print(a, b, c)

# Chamada da função
minhaFuncao(1, 2, 3)
```

- ▶ Essa subrotina recebeu o nome de “minhaFuncao” e recebe 3 argumentos na chamada, imprimindo-os. Ela não retorna nenhum valor como resultado. Note que a sub-rotina recebe três parâmetros: a, b e c.

# SUBROTINAS

- ▶ Subrotinas também podem **retornar um valor** como no exemplo a seguir

```
def quadrado(num):  
    return num**2  
  
print(quadrado(8)) # Imprimirá "64"
```

- ▶ A função ao lado recebe um número como argumento e retorna o seu quadrado que é impresso

## ESCOPO

- ▶ O escopo de uma variável indica sua visibilidade – ou seja, a partir de onde, no código, a variável é acessível
- ▶ Temos dois escopos para variáveis em Python:
  - ▶ Local
  - ▶ Global

## ESCOPO LOCAL

- ▶ Escopo **local**: criada dentro de uma função, existe apenas dentro da função onde foi declarada
  - ▶ As variáveis locais são inicializadas a cada nova chamada à função
  - ▶ Não é possível acessar seu valor fora da função onde ela foi declarada
  - ▶ Para que possamos interagir com variáveis locais, passamos parâmetros e retornamos valores nas funções

## ESCOPO GLOBAL

- ▶ Escopo **global**: declarada (criada) fora das funções e pode ser acessada por todas as funções presentes no módulo onde é definida
- ▶ Variáveis globais também podem ser acessadas por outros módulos, caso eles importem o módulo onde a variável foi definida
- ▶ Uma aplicação útil de variáveis globais é o armazenamento de valores constantes no programa, acessíveis a todas as funções
- ▶ Se for atribuído valor a ela, será na verdade criada uma nova variável, local, com o mesmo nome da global
- ▶ Dificultam o entendimento do código e violam o encapsulamento das funções, podendo serem alteradas por qualquer função, sem que seja simples saber quem a alterou

# MÓDULOS

- ▶ Para criar um módulo em Python, basta criar um arquivo com extensão “.py”
- ▶ Esse módulo será posteriormente incluído através de “import <nome módulo>” em outros scripts “.py” que você desenvolver
- ▶ Após incluir o módulo, você pode chamar funções que estejam implementadas no arquivo que foi incluído



# MÓDULOS

```
# Arquivo sequencia.py
def conteudoGC(sequencia):
    s = list(sequencia)
    if len(s) == 0:
        print('Sequência vazia.')
        return -1

    cC = cG = 0
    for n in s:
        if n == 'C':
            cC += 1
        elif n == 'G':
            cG += 1

    return (cG+cC)/len(s)
```

# PYTHON X PERL

```
# Arquivo sequencia.py
def conteudoGC(sequencia):
    s = list(sequencia)
    if len(s) == 0:
        print('Sequência vazia.')
        return -1
    cC = cG = 0
    for n in s:
        if n == 'C':
            cC += 1
        elif n == 'G':
            cG += 1
    return (cG+cC)/len(s)
```

```
sub conteudoGC{
    my ($sequencia) = @_;
    my (@s) = split(//, $sequencia);
    my ($n); my ($cG, $cC);
    if (scalar(@s) == 0){
        die ("Sequência \"$sequencia\" vazia.\n");
    }
    $cG = $cC = 0;
    foreach $n (@s){
        if ($n eq 'C'){
            $cC++;
        }elseif ($n eq 'G'){
            $cG++;
        }
    }
    return ($cG + $cC)/scalar(@s);
}
return 1;
```

# MÓDULOS

- ▶ Para usar a função em outro programa, você deve usar o comando `import` como no exemplo a seguir

```
import sequencia  
cGC = sequencia.conteudoGC( 'ACGTAGGGATGGCGTAGGAAAATGCGGGATGGCTGAGGCT' )  
print( '%0.4f' % cGC)
```

## COMO DIVIDIR O SOFTWARE EM MÓDULOS

- ▶ **Não há uma regra bem definida** para guiar essa divisão de um programa em módulos e rotinas
- ▶ Uma recomendação é **segmentar o máximo que for possível**
  - ▶ Sempre que você identificar uma porção do seu algoritmo que possa ser implementada dentro de uma rotina, você deve fazê-lo

## PASSAGEM DE PARÂMETROS...

- ▶ Agora que você já foi introduzido aos principais conceitos de modularização, subrotinas e módulos, entraremos em detalhes sobre as duas formas existentes de passagem de parâmetros para subrotinas
  - ▶ Passagem por valor
  - ▶ Passagem por referência