



**Instituto de
Computação**

UNIVERSIDADE ESTADUAL DE CAMPINAS



MC102 – Aula 08

Funções

Algoritmos e Programação de Computadores

Zanoni Dias

2020

Instituto de Computação

Funções

Exemplos de Uso de Funções

Exercícios

Funções

Funções

- Um aspecto importante na resolução de um problema complexo é conseguir dividi-lo em subproblemas menores.
- Sendo assim, ao criarmos um programa para resolver um determinado problema, uma tarefa importante é dividir o código em partes menores, fáceis de serem compreendidas e mantidas.
- As funções nos permitem agrupar um conjunto de comandos, que são executados quando a função é chamada.
- Nas aulas anteriores vimos diversos exemplos de uso de funções (**range**, **sum**, **min**, **len**, etc).
- Agora vamos nos aprofundar no uso de funções e aprender a criar nossas próprias funções.

Por que Utilizar Funções?

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Permitir o reaproveitamento de códigos, implementados por você ou por outros programadores.
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, evitando inconsistências e facilitando alterações.

Definindo uma Função

- Para criar uma nova função usamos o comando **def**.
- Para os nomes das funções valem as mesmas regras dos nomes de variáveis.

```
1 def imprime_mensagem():  
2     print("Minha primeira função")  
3  
4 imprime_mensagem()  
5 # Minha primeira função
```

Definindo uma Função

- Precisamos sempre definir uma função antes de usá-la.

```
1 imprime_mensagem()  
2 # NameError: name 'imprime_mensagem' is not defined  
3  
4 def imprime_mensagem():  
5     print("Minha primeira função")
```

Redefinindo uma Função

- Uma função pode ser redefinida, para isso basta declararmos outra função utilizando o mesmo nome, mas não necessariamente com o mesmo comportamento.

```
1 def imprime_mensagem():  
2     print("Minha função")  
3  
4 def imprime_mensagem():  
5     print("Minha função foi redefinida")  
6  
7 imprime_mensagem()  
8 # Minha função foi redefinida
```


Escopo de uma Variável

- O escopo de uma variável é o local do programa onde ela é acessível.
- Quando criamos uma variável dentro de uma função, ela só é acessível nesta função. Essas variáveis são chamadas de locais.

```
1 def imprime_mensagem():
2     mensagem = "Variável local"
3     print(mensagem)
4
5 imprime_mensagem()
6 # Variável local
7 print(mensagem)
8 # NameError: name 'mensagem' is not defined
```

Escopo de uma Variável

- Quando criamos uma variável fora de uma função, ela também pode ser acessada dentro da função. Essas variáveis são chamadas de globais.

```
1 mensagem = "Variável global"
2 def imprime_mensagem():
3     print(mensagem)
4
5 imprime_mensagem()
6 # Variável global
7 print(mensagem)
8 # Variável global
```

Escopo de uma Variável

- Uma variável local com o mesmo nome de uma global, “esconde” a variável global.

```
1 a = 1
2 def imprime():
3     a = 5
4     print(a)
5
6 imprime()
7 # 5
8
9 print(a)
10 # 1
```

Escopo de uma Variável

- Uma variável local com o mesmo nome de uma global, “esconde” a variável global.

```
1 a = 1
2 def incrementa():
3     a = a + 1
4     print(a)
5
6 incrementa()
7 # UnboundLocalError: local variable 'a' referenced before
8   assignment
9
```

Escopo de uma Variável

- Uma variável local com o mesmo nome de uma global, “esconde” a variável global.

```
1 a = 1
2 def incrementa():
3     a = 12
4     a = a + 1
5     print(a)
6
7 incrementa()
8 # 13
9 print(a)
10 # 1
```

Argumentos

- Na medida do possível devemos evitar o uso de variáveis globais dentro de funções, que dificultam a compreensão, manutenção e reuso da função.
- Se uma informação externa for necessária, ela deve ser fornecida como argumento da função.
- Podemos definir argumentos que devem ser informados na chamada da função.

```
1 def imprime_mensagem(mensagem):  
2     print(mensagem)  
3  
4 bomdia = "Bom dia"  
5 imprime_mensagem(bomdia)  
6 # Bom dia
```

- O escopo dos argumentos é o mesmo das variáveis criadas dentro da função (variáveis locais).

Argumentos

- Uma função pode receber qualquer tipo de dado como argumento.

```
1 def imprime_soma(x, y):  
2     print(x + y)  
3  
4 imprime_soma(2, 3)  
5 # 5  
6 imprime_soma("2", "3")  
7 # 23  
8 imprime_soma(2, "3")  
9 # TypeError: unsupported operand type(s) for +: 'int' and  
   'str'
```

Argumentos

- Podemos escolher atribuir explicitamente os valores aos argumentos (**argumento = valor**), mas estas atribuições devem ser as últimas a serem feitas.

```
1 def imprime_subtração(x, y):
2     print(x - y)
3
4 imprime_subtração(1, 4)
5 # -3
6 imprime_subtração(1, y = 4)
7 # -3
8 imprime_subtração(y = 1, x = 4)
9 # 3
10 imprime_subtração(y = 1, 4)
11 # SyntaxError: positional argument follows keyword argument
```


Argumentos

- Quando não informamos o número correto de argumentos, obtemos um erro.

```
1 def imprime_soma(x, y):  
2     print(x + y)  
3  
4 imprime_soma(1)  
5 # TypeError: imprime_soma() missing 1 required positional  
   argument: 'y'  
6  
7 imprime_soma(1, 2, 3)  
8 # TypeError: imprime_soma() takes 2 positional arguments  
   but 3 were given
```

Argumentos

- Podemos informar valores padrões para alguns dos argumentos.

```
1 def imprime_soma(x, y = 0):  
2     print(x + y)
```

- Argumentos com valores padrões não precisam ser explicitamente passados na chamada da função.

```
1 imprime_soma(1)  
2 # 1  
3 imprime_soma(1, 0)  
4 # 1  
5 imprime_soma(1, 2)  
6 # 3
```

Argumentos

- Os argumentos funcionam como atribuições. Quando passamos variáveis associadas a tipos simples, qualquer alteração no argumento não altera a variável original.

```
1 def incrementa_argumento(x): # x = 1
2     x = x + 1
3     print(x)
4
5 a = 1
6 incrementa_argumento(a)
7 # 2
8 print(a)
9 # 1
```

- Assim como no caso de atribuições, quando os argumentos são estruturas mutáveis, como listas e dicionários, estamos apenas dando um novo nome para a mesma estrutura.

```
1 def duplica_ultimo(lista): # lista = numeros
2     lista.append(lista[-1])
3     print(lista)
4
5 numeros = [1, 2, 3, 4]
6 duplica_ultimo(numeros)
7 # [1, 2, 3, 4, 4]
8 print(numeros)
9 # [1, 2, 3, 4, 4]
```

- Assim como no caso de atribuições, se não queremos que a estrutura original seja modificada, podemos criar uma cópia da estrutura usando o método **copy**.

```
1 def duplica_ultimo(lista): # lista = [1, 2, 3, 4]
2     lista.append(lista[-1])
3     print(lista)
4
5 numeros = [1, 2, 3, 4]
6 duplica_ultimo(numeros.copy())
7 # [1, 2, 3, 4, 4]
8 print(numeros)
9 # [1, 2, 3, 4]
```

Valor de Retorno

- Uma função pode retornar um valor. Para determinar o valor retornado usamos o comando **return**.

```
1 def mensagem():  
2     return "Mais uma função"  
3  
4 x = mensagem()  
5 print(x, len(x))  
6 # Mais uma função 15
```

- Podemos usar tuplas para retornar múltiplos valores.

```
1 def soma_e_subtração(x, y):  
2     return (x + y, x - y)  
3  
4 soma, subtração = soma_e_subtração(4, 1)  
5 print(soma, subtração)  
6 # 5 3
```

- Quando não utilizamos o comando **return** ou não informamos nenhum valor para o **return** a função retorna o valor **None**.

```
1 def soma(x, y):  
2     z = x + y  
3 def subtração(x, y):  
4     z = x - y  
5     return  
6  
7 resposta1 = soma(2, 3)  
8 resposta2 = subtração(2, 3)  
9 print(resposta1, resposta2)  
10 # None None
```

Valor de Retorno

- O comando **return** interrompe a execução da função, ou seja, nenhum outro comando da função é executado.
- Uma função pode ter múltiplos comandos **return**, mas apenas um deles será executado.

```
1 def retorna_soma(x, y):  
2     z = x + y  
3     return z  
4     print("Esta mensagem não será impressa")  
5  
6 print(retorna_soma(2, 3))  
7 # 5
```


A Função `main`

- Para manter o código bem organizado, podemos separar todo o programa em funções.
- Neste caso, a ultima linha do código contém uma chamada para a função principal (por convenção chamada de `main`).

```
1 def main():  
2     print("Execução da função main")  
3  
4 main()  
5 # Execução da função main
```

A Função main

- Como a chamada da função `main` fica no final do código, não precisamos nos preocupar com a ordem em que as outras funções são definidas.

```
1 def main():
2     função1()
3     função2()
4
5 def função2():
6     print("Execução da função 2")
7
8 def função1():
9     print("Execução da função 1")
10
11 main()
12 # Execução da função 1
13 # Execução da função 2
```

Exemplos de Uso de Funções

- Em aulas anteriores, vimos como testar se um número é primo:

```
1 n = int(input("Entre com um número inteiro positivo: "))
2 primo = True
3
4 for divisor in range(2, int(n**0.5)+1):
5     if n % divisor == 0:
6         primo = False
7         break
8
9 if primo:
10     print("Primo")
11 else:
12     print("Composto")
```

Números Primos

- Vamos criar uma função que realiza este teste.

```
1 def testa_primo(n):
2     primo = True
3     for divisor in range(2, int(n**0.5)+1):
4         if n % divisor == 0:
5             primo = False
6             break
7     return primo
8
9 n = int(input("Entre com um número inteiro positivo: "))
10
11 if testa_primo(n):
12     print("Primo")
13 else:
14     print("Composto")
```

Números Primos

- Vamos criar uma função que realiza este teste.

```
1 def testa_primo(n):
2     for divisor in range(2, int(n**0.5)+1):
3         if n % divisor == 0:
4             return False
5     return True
6
7
8
9 n = int(input("Entre com um número inteiro positivo: "))
10
11 if testa_primo(n):
12     print("Primo")
13 else:
14     print("Composto")
```

Números Primos

- Usando esta função vamos escrever um programa que imprima os n primeiros números primos.

```
1 def testa_primo(n):  
2     # ...  
3  
4     n = int(input("Numero de primos a serem calculados: "))  
5     candidato = 2  
6  
7     while n > 0:  
8         if testa_primo(candidato):  
9             print(candidato)  
10            n = n - 1  
11            candidato = candidato + 1
```

- As funções aumentam a clareza do código.
- Também tornam mais simples as modificações no código.
- Exemplo: melhorar o teste de primalidade.
 - Testar se o candidato é um número par.
 - Se for ímpar, testar apenas divisores ímpares (3, 5, 7, etc).
- O uso de funções facilita a manutenção do código.
- Neste caso, basta alterar a função `testa_primo`.


```
1 def testa_primo(n):  
2     if n % 2 == 0:  
3         return n == 2  
4     for divisor in range(3, int(n**0.5)+1, 2):  
5         if n % divisor == 0:  
6             return False  
7     return True
```

Horas, Minutos e Segundos

- Vamos criar uma função que recebe um valor em segundos e imprime este valor em horas, minutos e segundos.

```
1 def converte_tempo_segundos(segundos_totais):
2     horas = segundos_totais // 3600
3     resto = segundos_totais % 3600
4     minutos = resto // 60
5     segundos = resto % 60
6     print('{:02d}:{:02d}:{:02d}'.format(horas, minutos,
7                                         segundos))
8
9 converte_tempo_segundos(65135)
# 18:05:35
```

Horas, Minutos e Segundos

- Se quisermos receber o tempo em minutos podemos usar a função anterior.

```
1 def converte_tempo_segundos(segundos_totais):  
2     # ...  
3  
4 def converte_tempo_minutos(minutos_totais):  
5     converte_tempo_segundos(minutos_totais * 60)  
6  
7 converte_tempo_minutos(539)  
8 # 08:59:00
```

Horas, Minutos e Segundos

- O mesmo vale para receber o tempo em horas.

```
1 def converte_tempo_segundos(segundos_totais):  
2     # ...  
3  
4 def converte_tempo_horas(horas_totais):  
5     converte_tempo_segundos(horas_totais * 3600)  
6  
7 converte_tempo_horas(5)  
8 # 05:00:00
```

Horas, Minutos e Segundos

- Podemos criar uma única função que recebe a unidade como argumento.

```
1 def converte_tempo(total, unidade):
2     if unidade == "segundos":
3         converte_tempo_segundos(total)
4     elif unidade == "minutos":
5         converte_tempo_segundos(total * 60)
6     elif unidade == "horas":
7         converte_tempo_segundos(total * 3600)
8     else:
9         print("Unidade inválida")
10
11 converte_tempo(35135, "segundos")
12 # 09:45:35
13 converte_tempo(539, "minutos")
14 # 08:59:00
```

Horas, Minutos e Segundos

- Podemos criar uma única função que recebe a unidade como argumento.

```
1 def converte_tempo(total, unidade = "segundos"):
2     if unidade == "segundos":
3         converte_tempo_segundos(total)
4     elif unidade == "minutos":
5         converte_tempo_segundos(total * 60)
6     elif unidade == "horas":
7         converte_tempo_segundos(total * 3600)
8     else:
9         print("Unidade inválida")
10
11 converte_tempo(35135)
12 # 09:45:35
13 converte_tempo(539, "minutos")
14 # 08:59:00
```

Dias, Horas, Minutos e Segundos

- Se quisermos agora imprimir o tempo em dias, basta modificar a função `converte_tempo_segundos`.

```
1 def converte_tempo_segundos(segundos_totais):
2     dias = segundos_totais // (3600 * 24)
3     segundos_do_dia = segundos_totais % (3600 * 24)
4     horas = segundos_do_dia // 3600
5     resto = segundos_do_dia % 3600
6     minutos = resto // 60
7     segundos = resto % 60
8     print("{} dias, {} horas, {} minutos e {} segundos".
9           format(dias, horas, minutos, segundos))
10
11 def converte_tempo(total, unidade = "segundos"):
12     # ...
13
14 converte_tempo(1000000)
# 11 dias, 13 horas, 46 minutos e 40 segundos
```

Exercícios

1. Escreva uma função que, dados dois números inteiros positivos, calcule e retorne o Máximo Divisor Comum (MDC) entre os dois.
2. Escreva uma função que, dados dois números inteiros positivos, calcule e retorne o Mínimo Múltiplo Comum (MMC) entre os dois.
3. Escreva uma função que, dada uma lista de dois ou mais números inteiros positivos, calcule e retorne o Máximo Divisor Comum (MDC) entre eles.
4. Escreva uma função que, dada uma lista de dois ou mais números inteiros positivos, calcule e retorne o Mínimo Múltiplo Comum (MMC) entre eles.

5. Escreva uma função que dado um número inteiro ($n > 1$), retorne uma lista com os fatores primos de n .
6. Implemente uma função para calcular o número de combinações possíveis de m elementos em grupos de n elementos ($n \leq m$), dado pela fórmula de combinação:

$$\frac{m!}{(m - n)!n!}$$

7. Implemente uma função que, dada uma lista, retorne uma outra lista, com os elementos da lista original, sem repetições.
8. Implemente uma função que, dada uma lista, retorne a moda da lista, ou seja, uma lista com o(s) elemento(s) mais frequente(s) da lista original.

9. Implemente uma função que, dadas duas listas representando dois conjuntos, retorne uma lista que represente a união dos dois conjuntos.
10. Implemente uma função que, dadas duas listas representando dois conjuntos, retorne uma lista que represente a interseção dos dois conjuntos.
11. Implemente uma função que, dadas duas listas representando dois conjuntos, retorne uma lista que represente a diferença entre os dois conjuntos.
12. Implemente uma função que, dadas duas listas representando dois conjuntos, verifique se o primeiro é um subconjunto do segundo.

Algoritmo de Euclides

- O Algoritmo de Euclides (300 a.C.) calcula o Máximo Divisor Comum (MDC) de dois números inteiros positivos, sendo pelo menos um deles diferente de zero.
- O algoritmo usa dois fatos:
 - $MDC(x, 0) = x$
 - $MDC(x, y) = MDC(y, x \% y)$
- Exemplo:
 - $MDC(21, 15) = MDC(15, 21 \% 15) = MDC(15, 6)$
 - $MDC(15, 6) = MDC(6, 15 \% 6) = MDC(6, 3)$
 - $MDC(6, 3) = MDC(3, 6 \% 3) = MDC(3, 0)$
 - $MDC(3, 0) = 3$

- Possível Resposta:

```
1 def mdc2(x, y):  
2     while (y != 0):  
3         r = x % y  
4         x = y  
5         y = r  
6     return x
```

- Possível Resposta:

```
1 def mdc2(x, y):  
2     while (y != 0):  
3         (x, y) = (y, x % y)  
4     return x
```

MMC de Dois Números

- Possível Resposta:

```
1 def mmc2(x, y):  
2  
3  
4     resultado = 1  
5     while (resultado % x != 0) or (resultado % y != 0):  
6         resultado = resultado + 1  
7     return resultado
```

MMC de Dois Números

- Possível Resposta:

```
1 def mmc2(x, y):  
2  
3  
4     resultado = x  
5     while resultado % y != 0:  
6         resultado = resultado + x  
7     return resultado
```


MMC de Dois Números

- Possível Resposta:

```
1 def mmc2(x, y):  
2  
3  
4     resultado = max(x, y)  
5     while resultado % min(x, y) != 0:  
6         resultado = resultado + max(x, y)  
7     return resultado
```

MMC de Dois Números

- Possível Resposta:

```
1 def mmc2(x, y):  
2     if (x < y):  
3         (x, y) = (y, x)  
4     resultado = x  
5     while resultado % y != 0:  
6         resultado = resultado + x  
7     return resultado
```

MMC de Dois Números

- Possível Resposta:

```
1 def mmc2(x, y):  
2     resultado = 1  
3     divisor = 2  
4     while (x != 1) or (y != 1):  
5         if (x % divisor == 0) or (y % divisor == 0):  
6             resultado = resultado * divisor  
7             if x % divisor == 0:  
8                 x = x / divisor  
9             if y % divisor == 0:  
10                y = y / divisor  
11        else:  
12            divisor = divisor + 1  
13    return resultado
```

- Possível Resposta:

```
1 def mmc2(x, y):  
2     return int((x * y) / mdc2(x, y))
```

Combinações

- Possível Resposta:

```
1 def fatorial(x):  
2     fat = 1  
3     for i in range(1, x + 1):  
4         fat = fat * i  
5     return fat  
6  
7 def combinacoes(m, n):  
8     return fatorial(m) / (fatorial(m - n) * fatorial(n))
```