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
6   foi redefinida")
7
8 imprime_mensagem()
  # 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

```
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
7 global
8 print(mensagem)
  # 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)
1    # 1
0
```

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
      assignment
8
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)
1    # 1
0
```

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 funcão.

```
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, 2)
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 funcão.

```
1 imprime_soma(1)
2   # 1
3 imprime_soma(1,
4 0)
5 # 1
6 imprime_soma(1,
  2)
  # 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
5a = 1
6 incrementa_argumento(a)
7   # 2
8 print(a)
9   # 1
```

Argumentos

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

Argumentos

- 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
5numeros = [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
```

Valor de Retorno

- 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,
1  resposta2)    # None None
0
```

Valor de Retorno

- Os comandos depois de um `return` são desconsiderados.

```
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
4    função2()
5
6  def
7  função2():
8    print("Ex
9    ecução
1    da função
0    2")
1
1
1  def
2  função1():
1    print("Ex
3    ecução
     da função
     1")
```

Exemplo de Uso de Funções

Números Primos

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

```
n = int(input("Entre com um número inteiro positivo: "))
 primo = True

for divisor in range(2, int(n**0.5)+1):
    if n % divisor == 0:
     primo = False
      break

if primo:
   print ("Primo")
  else:
     print("Compo
   sto")
```

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):  if n % divisor == 0:
4        primo = False break
5    return primo
6
7  n = int(input("Entre com um número inteiro
8  positivo: "))
9
1  if testa_primo(n):
0    print("Primo")  else:
1      print("Composto")
1
2
1
3
1
4
```

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):  if n % divisor == 0:
3        return False return True
4
5
6
7      n = int(input("Entre com um número inteiro
8  positivo: "))
9
1      if testa_primo(n):
0    print("Primo")  else:
1      print("Composto")
1
2
1
3
1
4
```

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
```

Números Primos

- 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`.

Números Primos

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

```
  horas = segundos_totais // 3600
   resto = segundos_totais % 3600
   minutos = resto // 60
  segundos = resto % 60
  print('{:02d}:{:02d}:{:02d}'.format(horas, minutos,
        segundos))

converte_tempo_segundos(65135)
# 18:05:35
```

Horas, Minutos e Segundos

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

```
def converte_tempo(total, unidade):
  if unidade == "segundos":
    converte_tempo_segundos(total)
  elif unidade == "minutos":
    converte_tempo_segundos(total * 60)
  elif unidade == "horas":
    converte_tempo_segundos(total * 3600)
  else:
   print("Unidade inválida")


converte_tempo(35135,
"segundos")
# 09:45:35
converte_tempo(539, "minutos")
# 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")
1
0
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
10 def converte_tempo(total, unidade =
"s
11   # ...
12
13 converte_tempo(1000000)
14 # 11 dias, 13 horas, 46 minutos e 40
segundos
```

Exercícios

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.

Exercícios

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

APS 03 – Em grupo

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.

Desafio - Algoritmo de Euclides

- O Algoritmo de Euclides (300 a.C.) calcula o Máximo Divisor Comum (MDC) de dois números inteiros, 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:
  - $MDC(21, 15) = MDC(15, 21 \% 15) = MDC(15, 6)$
  - $MDC(15, 6) = MDC(6, 15 \% 6) = MDC(6, 3)$
  - $MDC(6, 3) = MDC(3, 6 \% 3) = MDC(3, 0)$
  - $MDC(3, 0) = 3$

MDC de Dois Números

- Possível Resposta:

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

MDC de Dois Números

- 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:

```
def mmc2(x, y):



  resultado = max(x, y)
  while resultado % min(x, y) != 0:
    resultado = resultado + max(x, y)
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:

```
def mmc2(x, y):
  resultado = 1
  divisor = 2
  while (x !=
  1) or (y !=
  1):
    if (x % divisor == 0) or (y % divisor == 0):
       resultado = resultado * divisor
      if x % divisor == 0:
          x = x / divisor
      if y % divisor == 0:
          y = y / divisor
  else:
    divisor = divisor + 1
return resultado
```

Combinação

- Possível Resposta:

```
def fatorial(x):
  fat = 1
  for i in range(1, x + 1):
      fat = fat * i
  return fat

    def combinacao(m,
            n):
  return
  fatorial(m) /
  (fatorial(m - n) *
  fatorial(n))
```