

CC1612 - Fundamentos de Algoritmos

Centro Universitário FEI

Prof. Danilo H. Perico

Modularização: Funções

Funções

- Funções são blocos de código que realizam determinadas tarefas que normalmente precisam ser executadas diversas vezes dentro da mesma aplicação
- Assim, tarefas muito utilizadas costumam ser agrupadas em funções, que, depois de definidas, podem ser utilizadas / chamadas em qualquer parte do código somente pelo seu nome

Funções - **def**

- Podemos criar nossas próprias funções no Python utilizando a palavra-chave **def**
- Sintaxe:

```
def <nome da função>():  
    # tarefas que serão realizadas dentro da função
```

- Exemplo:

```
def imprimeOla():  
    print("Olá")
```

Funções com e sem parâmetros

- As funções podem ou não ter parâmetros, que são valores enviados às funções no momento em que elas são chamadas
- Exemplos:

Sem parâmetros:

```
def soma():  
    a = 9  
    b = 8  
    print(a+b)
```

Chamada da
função

soma()

17

Com parâmetros:

```
def soma(a, b):  
    print(a+b)
```

soma(3,4)

Chamada da
função

7

Funções - **return**

- Além dos parâmetros, as funções podem ou não ter um valor de retorno
- O retorno é definido pela palavra-chave **return**
- Exemplos:

Sem parâmetros:

```
def soma():  
    a = 9  
    b = 8  
    return(a+b)  
  
print(soma())
```

17

Com parâmetros:

```
def soma(a, b):  
    return(a+b)  
  
print(soma(3,4))
```

7

Funções

- Exemplo: Fazer uma função que retorne *True* ou *False* para a verificação de números pares.
 - Precisa de parâmetros? Sim ou Não?
 - É melhor usar ou não o *return*?

Funções

- Exemplo: Fazer uma função que retorne *True* ou *False* para a verificação de números pares.

```
def par(num):  
    return(num % 2 == 0)
```

```
print(par(3))  
print(par(4))  
print(par(67))
```

```
False  
True  
False
```


Funções

- Exemplo: Se precisarmos de uma função que retorne a string “par” ou “ímpar”, podemos reutilizar a função par e criar uma função nova:

```
def par(num):  
    return(num % 2 == 0)  
  
def parOuImpar(x):  
    if par(x) == True:  
        return "par!"  
    else:  
        return "ímpar!"  
  
print(parOuImpar(4))  
print(parOuImpar(3))  
print(parOuImpar(56))
```

```
par!  
ímpar!  
par!
```

Exercícios

31. Escreva uma função que retorne o maior de dois números. A função deve se chamar *maximo(x, y)*.
32. Escreva uma função chamada *multiplo(x, y)* que receba dois números e retorna *True* se o primeiro for múltiplo do segundo número.
33. Escreva uma função que receba a base e a altura de um triângulo e retorne sua área ($A = \text{base} * \text{altura} / 2$).

Funções - escopo das variáveis: locais vs. globais

- Quando usamos funções, trabalhamos com variáveis internas, que pertencem somente à função.
- Estas variáveis são as *variáveis locais*
- Não podemos acessar os valores das variáveis locais fora da função a que elas pertencem
- É por isso que passamos parâmetros e retornamos valores nas funções. Os parâmetros e o *return* possibilitam a troca de dados no programa

Funções - escopo das variáveis: locais vs. globais

- Por sua vez, as **variáveis globais** são definidas fora das funções e podem ser vistas e acessadas por todas as funções e pelo “código principal”, que não está e função nenhuma

- Exemplo:

```
# variável global:  
a = 5  
  
def alteraValor():  
    # variável local da função alteraValor():  
    a = 7  
    print("Dentro da função 'a' vale: ", a)  
  
print("'a' antes da chamada da função: ", a)  
alteraValor()  
print("'a' depois da chamada da função", a)
```

```
'a' antes da chamada da função: 5  
Dentro da função 'a' vale: 7  
'a' depois da chamada da função 5
```

Funções - escopo das variáveis: locais vs. globais

- Exemplo: Se quisermos modificar a variável global dentro da função, devemos utilizar a palavra-chave *global*

```
# variável global:  
a = 5  
  
def alteraValor():  
    # dizemos para a função que a variável 'a' é global:  
    global a  
    a = 7  
    print("Dentro da função 'a' vale: ", a)  
  
print("'a' antes da chamada da função: ", a)  
alteraValor()  
print("'a' depois da chamada da função", a)
```

```
'a' antes da chamada da função: 5  
Dentro da função 'a' vale: 7  
'a' depois da chamada da função 7
```

Exercícios

34. Escreva uma função que receba uma string e uma lista como parâmetros. A função deve comparar a string passada com os elementos da lista e retornar **True** se a string for encontrada na lista, e **False**, caso contrário.

Funções - parâmetros opcionais

- Podemos ainda criar funções que podem ou não receber argumentos.
- Exemplo:

Chamada sem
argumento:
Neste caso, soma
assume valores 1
para *a* e *b*

```
def soma(a=1, b=1):  
    print(a+b)
```

```
soma()  
soma(2,3)
```

2
5

Chamada com
argumentos:
Neste caso, soma
assume valores 2
para *a* e 3 para *b*

Funções *lambda*

- No Python podemos criar funções simples, sem nome, chamadas de *lambda*
- Podemos pensar em *lambdas* como funções de uma só linha
- Exemplo: função *lambda* que recebe um parâmetro *x* e que retorna o quadrado deste número. *Lambda* cria uma função, que é atribuída a variável *a*

```
a = lambda x: x**2  
print(a(3))
```

9

Funções *lambda*

- Exemplo: função *lambda* que calcula o aumento, dado o valor inicial e a porcentagem de aumento.

```
aumento = lambda a,b : a*b/100  
aumento(100,5)
```

```
5.0
```

Exercícios

35. Neste exercício, você escreverá uma função que determina se uma senha é ou não é boa. Para ser boa, uma senha tem que ter pelo menos 8 caracteres, conter pelo menos uma letra maiúscula, pelo menos uma letra minúscula e pelo menos um número. Sua função deve retornar *True* se a senha for boa. Caso contrário, deve retornar *False*.
36. Uma data é considerada mágica quando o dia multiplicado pelo mês é igual ao ano de dois dígitos. Por exemplo, 10 de junho de 1960 é uma data mágica porque junho é o sexto mês e 6 vezes 10 é 60, o que equivale ao ano de dois dígitos. Escreva uma função que determine se uma data é ou não uma data mágica. Use sua função em um programa principal que deve encontrar e exibir todas as datas mágicas do século XX.

Arquivos

Arquivos

- Utilizar arquivos é uma forma de garantir o armazenamento permanente dos dados que são importantes no seu programa, pois nenhuma variável, nem mesmo a lista, continua existindo depois que o programa termina.
- Então, utilizar um arquivo é uma maneira excelente de trabalhar com a entrada e a saída de dados para os programas.
- Arquivos são linhas de texto, normalmente salvos com a extensão *.txt* ou *.dat*

Arquivos

- Na programação, assim como na nossa interação com o computador, o primeiro passo para acessar um arquivo é abri-lo.
- Para abrir o arquivo, utilizamos a função *open*
- Sintaxe:

```
arquivo = open("teste.dat", "w")
```

- A variável *arquivo* salva o arquivo em si. É por meio desta variável que executaremos as funções de escrita e leitura.
- *open()* tem dois parâmetros: nome do arquivo e modo de acesso

Arquivos

- O modo de acesso pode ser:

modo	operação
r	leitura
w	escrita
a	escrita, preservando o conteúdo existente
b	modo binário
+	atualização (leitura e escrita)

```
arquivo = open("teste.dat", "w")
```

Arquivos - Escrita

- Para escrever no arquivo, utilizamos o método *write*, que vai ser chamado pela variável *arquivo*:

```
arquivo.write("texto a ser escrito no arquivo")
```

- O método *write* funciona de maneira similar que o *print* com marcadores (%d, %f, %s), porém precisamos sempre incluir o “\n” quando queremos ir para a próxima linha.

Arquivos - Escrita

- Depois que escrevemos no **arquivo**, precisamos fechá-lo, utilizando o método **close**

```
arquivo.close()
```

- É sempre importante fechar o arquivo para informar ao Sistema Operacional que não vamos mais utilizá-lo.
- Muitas vezes, o Sistema Operacional salva as informações que queremos escrever em uma memória auxiliar e deixa a operação de escrever realmente no arquivo só quando informamos que vamos fechá-lo.
- Então, se não fechamos, corremos o risco de perder o que gostaríamos de escrever.

Arquivos - Escrita

- Exemplo completo de escrita em arquivo texto:

```
arquivo = open("teste.dat", "w")

for linha in range(1,101):
    arquivo.write("Linha %d\n" % linha)
arquivo.close()
```

teste.dat:

```
Linha 1
Linha 2
Linha 3
Linha 4
Linha 5
Linha 6
Linha 7
Linha 8
Linha 9
Linha 10
Linha 11
Linha 12
Linha 13
Linha 14
Linha 15
```

Arquivos - Leitura

- Para ler do arquivo, precisamos seguir o mesmo procedimento:
 - Abrir o arquivo em modo leitura “r”

```
arquivo = open("teste.dat", "r")
```

- Utilizar um método para ler o arquivo
- Fechar o arquivo com o método *close*

Arquivos - Leitura

- Para ler do arquivo, podemos utilizar o método *readlines()*
- Exemplo completo:

```
arquivo = open("teste.dat", "r")  
  
for linha in arquivo.readlines():  
    print(linha)  
arquivo.close()
```

Linha 1

Linha 2

Linha 3

Linha 4

Linha 5

Linha 6

Arquivos

- Exemplo gerar e gravar números pares e ímpares em arquivos separados. Números de 0 a 999.

```
impares = open("impares.txt", "w")  
  
pares = open("pares.txt", "w")  
  
for n in range(1000):  
    if n % 2 == 0:  
        pares.write("%d\n" % n)  
    else:  
        impares.write("%d\n" % n)  
impares.close()  
pares.close()
```

Arquivos

- Podemos realizar diversas operações com os arquivos
- Por exemplo:
 - Ler
 - Processar
 - Gerar novos arquivos

Arquivos

- Exemplo: Utilizando o arquivo “*pares.txt*” gerado no último exemplo, vamos criar outro arquivo que deve conter somente os números múltiplos de 4.

```
multiplos4 = open("multiplos_4.txt", "w")

pares = open("pares.txt", "r")

for linha in pares.readlines():
    if int(linha) % 4 == 0:
        multiplos4.write(linha)
pares.close()
multiplos4.close()
```

Exercício

37. Crie um programa que inverta a ordem das linhas do arquivo *pares.txt*. A primeira linha deve conter maior número e a última o menor. Salve o resultado em outro arquivo, chamado *pares_invertido.txt*.

Arquivos

- Até agora, estamos utilizando somente um dado por linha
- Porém, podemos salvar informações correlatas na mesma linha
- Exemplo: Criar um arquivo com o nome e o telefone de pessoas, conforme são digitados pelo usuário. O programa deve funcionar em loop até que o nome digitado seja vazio.

Arquivos - Escrita de dois dados na mesma linha

```
1 contatos = open("contatos.dat", "w")
2 nome = input("Nome: ")
3 telefone = input("Telefone: ")
4
5 while nome != "":
6     contatos = open("contatos.dat", "a")
7     contatos.write("%s %s\n" % (nome, telefone))
8     contatos.close()
9     nome = input("Nome: ")
10    telefone = input("Telefone: ")
11
```

```
Nome: fulano
Telefone: 123456
Nome: sicrano
Telefone: 9876543
Nome: beltrano
Telefone: 5555555
```

Arquivos - Leitura de dois dados na mesma linha

- Entendo melhor o `readlines()`
 - O `readlines()` retorna uma lista onde cada uma das linhas ocupa uma posição/ índice:

```
1 contatos = open("contatos.dat", "r")
2
3 conteudo_do_arquivo = contatos.readlines()
4
5 print(conteudo_do_arquivo)
```

['fulano 123456\n', 'sicrano 9876543\n', 'beltrano 5555555\n']

Método `split(x)`

- Divide as informações no caractere informado como parâmetro
- Exemplo:

```
nome, telefone = input("Entre com o nome e o telefone: ").split(" ")  
  
print(nome)  
print(telefone)
```

```
Entre com o nome e o telefone: fulano 1234567  
fulano  
1234567
```

Arquivos

- Como ler uma linha com duas informações?

```
1 contatos = open("contatos.dat", "r")
2
3 contato = []
4
5 for linha in contatos.readlines():
6     linha_separada = linha.split(" ")
7     contato.append(linha_separada)
8
9 print(contato)
10 print(contato[0])
11 print(contato[0][0])
12 print(contato[0][1])
```

Lista de listas

```
[['fulano', '123456\n'], ['sicrano', '9876543\n'], ['beltrano', '5555555\n']]
['fulano', '123456\n']
fulano
123456
```

Exercício

38. Uma empresa de 500 funcionários, está tendo problemas de espaço em disco no seu servidor de arquivos. Para tentar resolver este problema, o administrador de rede precisa saber qual o espaço ocupado pelos usuários, e identificar os usuários com maior espaço ocupado. Através de um programa, baixado da Internet, ele conseguiu gerar o seguinte arquivo, chamado "usuarios.txt":

alexandre	456123789
anderson	1245698456
antonio	123456456
carlos	91257581
cesar	987458
rosemary	789456125

Exercícios

A partir deste arquivo, você deve criar um programa que gere um relatório, chamado "relatório.txt", no seguinte formato:

Uso do espaço em disco pelos usuários			

--			
Nr.	Usuário	Espaço utilizado	% do uso
1	alexandre	434,99 MB	16,85%
2	anderson	1187,99 MB	46,02%
3	antonio	117,73 MB	4,56%
4	carlos	87,03 MB	3,37%
5	cesar	0,94 MB	0,04%
6	rosemary	752,88 MB	29,16%
Espaço total ocupado: 2581,57 MB			
Espaço médio ocupado: 430,26 MB			