Universidade Federal do Vale do São Francisco

Minicurso de Linguagem Python

Edjair Aguiar Gomes Filho – Representante Discente pelo Centro Acadêmico de Engenharia da Computação

Aula 3 – Executando Arquivos Python/Listas

Sumário

1) ARQUIVOS PYTHON

Embora a opção de utilizar o modo interativo seja uma boa forma de aprender e praticar a sintaxe da linguagem Python, concordemos que não é a forma mais elegante de programar e rodar nossas aplicações.

Assim como em qualquer outra linguagem, é muito mais prático que rodemos nossos programas quando já estão finalizados, em formato de arquivo. Podemos utilizar uma IDE (como usamos o *DEV* ou o *CodeBlocks* para programar em .C, por exemplo), mas sempre existe a opção de primeiramente escrever o programa completo em um editor de texto e em seguida executá-lo no prompt de comando.

Por enquanto utilizaremos a opção de executar arquivos no prompt de comando, pois os ambientes de desenvolvimento Python serão abordados mais futuramente, em um capítulo à parte.

1.1) Executando arquivos .py

Assim como os módulos, a extensão de arquivo dos nossos códigos é *.py*.

Vamos escrever um programa simples. Podemos abrir o editor de texto e escrever um programa para simplesmente exibir uma saudação na tela e se despedir quando o usuário digitar qualquer coisa no teclado, como na Figura 1.

```
print("Hello World!")
input('Você já está indo? Tecle enter para sair!')
print("Bye World...")
quit()
```

Figura 1 – Arquivo contendo um programa simples em Python.

Salvamos nosso arquivo com o nome *hellobye.py*.

Se você já configurou o interpretador Python seu prompt de comando, agora basta especificar o caminho do seu arquivo no prompt. Isso mesmo: não há necessidade de chamar o interpretador digitando *python*. Simplesmente especifique o caminho do arquivo no seu promp de comando e o seu arquivo irá executar

[!] DICA: se você ficar com preguiça de digitar o caminho do arquivo, pode simplesmente arrastá-lo para o prompt e ele reconhecerá o caminho automaticamente.

Quando o caminho do arquivo estiver especificado no prompt, basta teclar enter e o código será executado, sem maiores problemas.

```
C:\Users\USUARIO>C:\Users\USUARIO\Documents\hellobye.py
Hello World!
Você já está indo? Tecle enter para sair!
Bye World...
```

Figura 2 – O código é executado em sua completude, sem necessidade de digitar comandos linha por linha no prompt. Muito mais elegante.

1.2) Importando módulos em arquivos de código

1.2.1) Módulos padrões

Para acessarmos módulos padrões de Python dentro dos nossos arquivos de código, ao invés de os chamarmos no prompt, devemos declará-los no cabeçalho do código, da mesma forma que as bibliotecas em linguagem C. Por exemplo: suponhamos o código da Figura 3, que calcula a raiz quadrada de um número que o usuário insere.

```
import math

x = int(input('Digite um número qualquer: '))
resultado = math.sqrt(x)
print('A raiz quadrada desse número é', resultado)
```

Figura 3 – Programa para calcular raiz quadrada. Observe que o *import* precisou ser chamado no início do script.

[1] LEMBRE-SE: A função input() deve ser usada dentro da função conversão int(). Lembra da primeira aula? O input lê a entrada do usuário como uma string, mas precisamos que a entrada seja um inteiro para operar a raiz quadrada.

[2] LEMBRE-SE: Para acessarmos as funções que existem em um módulo, precisamos especificar de qual módulo está vindo a função, no formato nomeDoMódulo.nomeDaFunção().

Agora podemos simplesmente chamar nosso programa no prompt de comando e executá-lo.

```
C:\Users\USUARIO>C:\Users\USUARIO\Documents\raiz.py
Digite um número qualquer: 56
A raiz quadrada desse número é  7.483314773547883
```

Figura 4 – Execução do código da Figura 3

1.2.2) Módulos criados pelo programador

Para importamos dentro do código um módulo que nós mesmos criamos, basta que os dois arquivos estejam dentro da mesma pasta.

Vamos primeiramente criar um módulo simples que define uma função que calcula o dobro de um número inserido pelo usuário. Chamaremos esse módulo de *vezesdois.py*.

```
def dobro(x):
    print('O dobro desse número é', x*2)
```

Figura 5 – Módulo criado por nós, vezesdois.py

Agora escrevemos um programa que solicita ao usuário que insira um número. Importamos nosso módulo *vezesdois*, pedimos um número ao usuário e chamamos a função *dobro* de *vezesdois*, passando o valor do usuário como parâmetro da função.

```
import vezesdois

num = input('Insira um valor para que seja calculado seu dobro: ')
x = int(num)
resultado = vezesdois.dobro(x)
```

Figura 6 – Nosso programa, que utiliza a função *dobro* do módulo *vezesdois*

Salvamos nosso programa como *boraver.py,* na mesma pasta que nosso módulo.

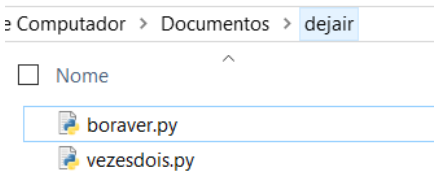

Figura 7 – Os arquivos devem estar na mesma pasta

Executamos agora nosso programa *boraver.py* normalmente no prompt de comando.

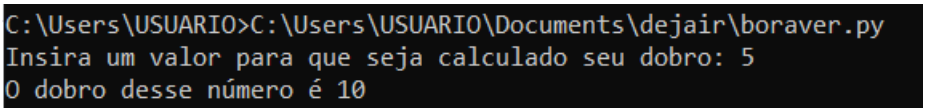

Figura 8 – Executando o código que criamos, que chama o módulo que criamos. Dá muito trabalho ser um programador tão autossuficiente...

2) LISTAS

A gente aprende no curso de Computação que a definição de lista é *uma estrutura de dados composta por itens organizados de forma linear, na qual cada um pode ser acessado a partir de um índice, começando do zero, onde cada índice representa sua posição na lista...* etc.

O que não te contam nas disciplinas regulares do curso (ou contam, muito superficialmente) é que, diferentemente da linguagem C, por exemplo, as listas em Python não-obrigatoriamente armazenam somente objetos do mesmo tipo. Isso significa que podemos armazenar simultaneamente strings, floats, inteiros... simplificando: em Python, nossas listas podem armazenar basicamente *qualquer coisa*. Inclusive outras listas.

2.1) Representação

Nossa lista em Python vai ser representada como uma sequência de objetos separados por vírgula, dentro de colchetes. Assim, uma lista vazia, por exemplo, pode ser definida pelos colchetes vazios. A Tabela 1 mostra uma listagem de algumas possibilidades de criação de listas. Na primeira coluna temos a declaração da lista e na segunda coluna temos o que é armazenado nessa declaração.

univasf = []	
univasf = [‘lab’, ‘ru’, 123, 111]	lab, ru, 123, 111
nova_uni = [‘tristeza’, univasf]	tristeza, [lab, ru, 123, 111]

Tabela 1 – Possibilidades de listas em Python

2.2) Operadores

i) Acesso por índices: Uma lista é armazenada por índices, começando a partir do zero. Portanto, podemos acessar os objetos de nossa lista através dos seus índices. Por exemplo, a lista chamada *univasf* da Tabela 1 contém quatro elementos, cujos índices variam de zero a três. Dessa forma, podemos acessar cada um dos objetos através dos seus índices.

univasf[0]	lab
univasf[1]	ru
univasf[2]	123
univasf[3]	111

ii) Comprimento: O comprimento de uma lista é devolvido através da função *len()*. Por exemplo, se chamarmos a função len() passando como parâmetro nossa lista *univasf,* será nos devolvido o tamanho quatro. Se passarmos como parâmetro a *nova_uni*, obteremos o tamanho dois, pois *univasf* é um único objeto dentro da lista *nova_uni*.

len(univasf)	4
len(nova_lista)	2

iii) Concatenação e Multiplicação: Podemos concatenar listas usando a adição e multiplicar listas por um inteiro, que vai gerar várias cópias dos seus itens.

univasf+nova_uni	lab, ru, 123, 111, tristeza, [lab, ru, 123, 111]
lista*3	lab, ru, 123, 111, lab, ru, 123, 111, lab, ru, 123, 111,

iv) Verificação de itens: Outra funcionalidade interessante é que podemos procurar um objeto específico em uma lista. Usaremos o operador *in*, que deve ser usado da seguinte forma

‘nomeDoObjeto’ in nomeDaLista

Esse operador vai retornar um valor booleano (*True* ou *False*), que obviamente vai indicar se o objeto está ou não contido na lista.

‘ru’ in univasf	True
‘lolzinho’ in univasf	False

v) Mínimo, máximos e somatório: Caso declaremos uma lista numérica, podemos usar as funções *min()*, *max()* e *sum()*, que devolvem, respectivamente, o menor valor, o maior valor e a soma dos elementos da lista. Por exemplo, declaramos uma lista chamada números:

numeros = [14.55, 67, 89.88, 10, 21.5]

Poderemos utilizar as funções descritas, que nos retornarão:

min(numeros)	10
max(numeros)	89.88
sum(numeros)	202.93

2.3) Métodos

A diferença entre uma operação e um método é que a operação nos retornam um resultado, enquanto o método efetua alguma alteração na estrutura da lista. Os métodos são utilizados com a seguinte sintaxe:

nomeDaLista.método(parâmetro)

Vejamos abaixo alguns exemplos de métodos.

i) append(objeto): Caso declaremos uma lista chamada *para_ler:*

para_ler = [‘1984’, ‘Admirável Mundo Novo’, ‘A Revolução dos Bichos’, ‘Fahrenheit 451’]

Podemos adicionar um novo elemento no final dessa lista usando o método *append* e passando qual objeto queremos acrescentar. Teremos:

para_ler.append(‘Laranja Mecânica’)	
para_ler	[‘1984’, ‘Admirável Mundo Novo’, ‘A Revolução dos Bichos’, ‘Fahrenheit 451’, ‘Laranja Mecânica’]

ii) insert(índice, objeto): Enquanto o método *append* adiciona um objeto apenas no final da lista, o *insert* permite adicionar um novo objeto em qualquer índice que escolhermos.

para_ler.insert(0, 'A Guimba')	
para_ler	['A Guimba', '1984', 'Admirável Mundo Novo', 'A Revolução dos Bichos', 'Fahrenheit 451', 'Laranja Mecânica']

iii) pop(indice): Esse método remove um objeto da lista **através do seu índice** e retorna qual foi o objeto removido.

para_ler.pop(1)	'1984'
para_ler	['A Guimba', 'Admirável Mundo Novo', 'A Revolução dos Bichos', 'Fahrenheit 451', 'Laranja Mecânica']

iv) remove(objeto): Também remove um objeto da lista, mas ao invés de utilizar o índice, utiliza o nome do objeto. É um método silencioso, não retorna o objeto removido.

para_ler.remove('A Guimba')	
para_ler	['Admirável Mundo Novo', 'A Revolução dos Bichos', 'Fahrenheit 451', 'Laranja Mecânica']

v) sort(): O sort organiza a lista em ordem crescente para números e em ordem alfabética (lexicográfica, para ser chique) para string.

para_ler.sort()	
para_ler	['Admirável Mundo Novo', 'A Revolução dos Bichos', 'Fahrenheit 451', 'Laranja Mecânica']

vi) reverse(): Inverte as posições dos objetos da lista.

para_ler.reverse()	
para_ler	[‘Laranja Mecânica’, ‘Fahrenheit 451’, ‘A Revolução dos Bichos’, ‘Admirável Mundo Novo’]

vii) count(objeto): Esse método devolve o número de ocorrências de um de um determinado objeto passado como parâmetro. Suponha a lista *filmes_vi_hj* abaixo:

filmes_vi_hj = [‘Harry Potter 3’, ‘Frozen’, ‘Vingadores 1, ‘Vingadores Sei Lá O Penúltimo, ‘Frozen’]

Com o método *count,* podemos obtere os seguintes resultados:

filmes_vi_hj.count(‘Harry Potter 3’)	1
filmes_vi_hj.count(‘Frozen)	2

3) EXERCÍCIOS

1. Declare uma lista com 5 números inteiros a seu gosto. Faça com que o programa exiba essa lista em ordem inversa.
2. Crie uma lista das suas séries favoritas. Podem ser quantas você quiser! Lembre-se que as variáveis em Python são dinâmicas. Em seguida, faça com que o programa exibe as séries em ordem alfabética.
3. Faça um programa que leia quatro notas, armazene numa lista e imprima a média. Dica: crie uma lista vazia e utilize o método *append* para armazenar as entradas do usuário. Depois utilize a operação *sum* para ajudar a calcular a média das notas.
4. Utilize uma lista e faça um programa que pergunte cinco questões de ‘sim’ ou ‘não’ para um criminoso. As cinco perguntas são:
   a) Você telefonou para as vítimas?
   b) Esteve no local do crime?
   c) Mora perto da vítima?
   d) Você é agiota?

   Armazene essas respostas na lista criada e imprima a quantidade de vezes que o criminoso respondeu ‘sim’ e quantas vezes respondeu ‘não’.

   Lembre-se que agora já podemos escrever nossos programas em arquivos!