



## UNIDADE IV

---

### Introdução à Programação Estruturada

Prof. Me. Ricardo Veras

# Métodos de Manipulação de Listas

## LISTAS:

- **Lista** é uma das estruturas de dados mais utilizadas, por sua flexibilidade e sua praticidade.

## MÉTODOS DE MANIPULAÇÃO DE LISTAS:

- `copy()` :: retorna uma cópia da lista  
**`lista.copy()`**.

```
1  lst01 = [5, 1, 8, 3, -2, 0, 15, 10]
2  lst02 = lst01
3  lst03 = lst01.copy()
4  print("01A:", lst01)
5  lst01[2] = 0
6  print("01B:", lst01)
7  print("02:", lst02)
8  print("03:", lst03)
9  '''
10 Saída:
11 01A: [5, 1, 8, 3, -2, 0, 15, 10]
12 01B: [5, 1, 0, 3, -2, 0, 15, 10]
13 02: [5, 1, 0, 3, -2, 0, 15, 10]
14 03: [5, 1, 8, 3, -2, 0, 15, 10]
15 '''
```

# Métodos de Manipulação de Listas

- *append()* :: Adiciona um elemento ao final da lista

**lista.append(valor)**

```
1  lst01 = [5, 1, 8, 3, -2, 0, 15, 10]
2  num01 = 3
3  lst01.append(num01)
4  print(lst01)
5  lst02 = [-2, 7, 1, 12]
6  lst01.append(lst02)
7  print(lst01)
8  '''
9  Saída:
10 [5, 1, 8, 3, -2, 0, 15, 10, 3]
11 [5, 1, 8, 3, -2, 0, 15, 10, 3, [-2, 7, 1, 12]]
12 '''
```

- *clear()* :: Remove todos os elementos da lista

**lista.clear()**

```
1  lst01 = [5, 1, 8, 3, -2, 0, 15, 10]
2  lst01.clear()
3  print(lst01)
4  '''
5  Saída:
6  []
7  '''
```

Fontes: autoria própria.

# Métodos de Manipulação de Listas

- `extend()` :: Adiciona ao final da lista os elementos de uma outra lista (um a um):

**lista01.extend(lista02)**

```
1  lst01A = [1, 2, 3, 4, 5]
2  lst01B = [1, 2, 3, 4, 5]
3  lst02A = [6, 7, 8, 9]
4  lst02B = [6, 7, 8, 9]
5  # acrescentando com "append"
6  lst01A.append(lst02A)
7  # acrescentando com "extend"
8  lst01B.extend(lst02B)
9  print(lst01A)
10 print("Lista A com ", len(lst01A), "elementos")
11 print(lst01B)
12 print("Lista B com ", len(lst01B), "elementos")
13 '''
14 Saída:
15 [1, 2, 3, 4, 5, [6, 7, 8, 9]]
16 Lista A com 6 elementos
17 [1, 2, 3, 4, 5, 6, 7, 8, 9]
18 Lista B com 9 elementos
19 '''
```

# Métodos de Manipulação de Listas

- *count()* :: Retorna o número de elementos que são iguais ao valor especificado

**var = lista.count(valor)**

```
1  lst01 = [5, 1, 8, 5, -2, 5, 15, 10]
2  v01 = lst01.count(4)
3  v02 = lst01.count(8)
4  v03 = lst01.count(5)
5  print(v01, v02, v03)
6  '''
7  Saída:
8  0 1 3
9  '''
```

# Métodos de Manipulação de Listas

- `index()` :: Retorna o índice do primeiro elemento encontrado com o valor especificado  
**`var = lista.index(valor)`**

```
1  lst01 = ["João", "Maria", "Ana", "José", "Julia"]
2  # definindo valores iniciais
3  v01, v02 = -1, -1
4  # verificando posição
5  if ("Julia" in lst01): v01 = lst01.index("Julia")
6  if ("Mateus" in lst01): v02 = lst01.index("Mateus")
7  print(v01, v02)
8  '''
9  Saída:
10  4 -1
11  '''
```

Fonte: autoria própria.

- Obs.: caso o valor não exista na lista e se tente pegar o índice daquele valor, o sistema retorna um erro.

# Métodos de Manipulação de Listas

- *insert()* :: Adiciona um elemento na posição pretendida

**lista.insert(posicao, valor)**

```
1  lst01 = [1, 2, 3, 4, 5]
2  # acrescentando com "insert"
3  lst01.insert(2, 9)
4  print(lst01)
5  '''
6  Saída:
7  [1, 2, 9, 3, 4, 5]
8  '''
```

Fontes: autoria própria.

- *pop()* :: Remove o último elemento da lista, ou o elemento da posição pretendida.

**lista.pop()**

ou

**lista.pop(posicao)**

```
1  lst01 = [1, 2, 3, 4, 5, 6, 7]
2  print(lst01)
3  lst01.pop()
4  print(lst01)
5  lst01.pop(2)
6  print(lst01)
7  '''
8  Saída:
9  [1, 2, 3, 4, 5, 6, 7]
10 [1, 2, 3, 4, 5, 6]
11 [1, 2, 4, 5, 6]
12 '''
```

# Métodos de Manipulação de Listas

- `remove()` :: Remove o item com o valor especificado (apenas o primeiro que encontrar)  
**`lista.remove(valor)`**

```
1  lst01 = [1, 5, 2, 7, 5, 3, 4]
2  lst02 = ["João", "Maria", "Ana", "José", "Julia"]
3  lst01.remove(5)
4  lst02.remove("Maria")
5  # lst02.remove("Mateus") - resulta em erro
6  print(lst01)
7  print(lst02)
8  '''
9  Saída:
10 [1, 2, 7, 5, 3, 4]
11 ['João', 'Ana', 'José', 'Julia']
12 '''
```



# Métodos de Manipulação de Listas

- `reverse()` :: Inverte a ordem dos valores da lista

**`lista.reverse()`**

- `sort()` :: Ordena os valores da lista (alfabeticamente/numericamente)

**`lista.sort()`**

```
1  lst01 = [1, 5, 2, 7, 5, 3, 4]
2  lst02 = ["João", "Maria", "Ana", "José", "Julia"]
3  lst01.reverse()
4  lst02.reverse()
5  print(lst01)
6  print(lst02)
7  lst01.sort()
8  lst02.sort()
9  print(lst01)
10 print(lst02)
11 '''
12 Saída:
13 [4, 3, 5, 7, 2, 5, 1]
14 ['Julia', 'José', 'Ana', 'Maria', 'João']
15 [1, 2, 3, 4, 5, 5, 7]
16 ['Ana', 'José', 'João', 'Julia', 'Maria']
17 '''
```

# Interatividade

Imagine que eu tenha duas variáveis, cada uma delas com a estrutura de lista, de modo que a “primeira” contém 5 elementos e a “segunda” contém 7 elementos. Se eu precisasse fazer com que a “segunda” adquirisse todos os elementos da “primeira”, de modo a ficar com 12 elementos ao todo, qual linha de código, dentre as opções a seguir, eu deveria utilizar?

- a) `segunda.append(primeira).`
- b) `segunda.extend(primeira).`
- c) `segunda.insert(primeira).`
- d) `primeira.append(segunda).`
- e) `primeira.extend(segunda).`

## Resposta

Imagine que eu tenha duas variáveis, cada uma delas com a estrutura de lista, de modo que a “primeira” contém 5 elementos e a “segunda” contém 7 elementos. Se eu precisasse fazer com que a “segunda” adquirisse todos os elementos da “primeira”, de modo a ficar com 12 elementos ao todo, qual linha de código, dentre as opções a seguir, eu deveria utilizar?

- a) `segunda.append(primeira).`
- b) `segunda.extend(primeira).`
- c) `segunda.insert(primeira).`
- d) `primeira.append(segunda).`
- e) `primeira.extend(segunda).`

# Dicionários

## DICIONÁRIO:

- É uma estrutura de dados similar às listas, mas com propriedades de acesso diferentes;
- É uma coleção de elementos, ordenada e mutável;
- Em Python, cria-se o dicionário utilizando “chaves” {};
- O dicionário não permite a inclusão de elementos duplicados;
- Um dicionário é composto por um conjunto de **chaves** (*keys*) e de **valores** (*values*).

PRODUTO	PREÇO
Alface	R\$ 2,50
Batata	R\$ 5,80
Tomate	R\$ 4,30
Feijão	R\$ 2,20

```
1  # variavel = {
2  #     "chave1": valor1,
3  #     "chave2": valor2,
4  #     ...
5  # }
6  tabela = {
7      "Alface":2.50,
8      "Batata":5.80,
9      "Tomate":4.30,
10     "Feijão":2.20,
11 }
```

Fonte: autoria própria.

# Dicionários

## DICIONÁRIO:

- Diferente das listas, onde o índice é um valor numérico, no caso do dicionário o índice será cada uma das “chaves”;
- Assim, cada elemento de um dicionário é acessado por meio de sua chave;

```
1  tabela = {  
2      "Alface":2.50,  
3      "Batata":5.80,  
4      "Tomate":4.30,  
5      "Feijão":2.20,  
6  }  
7  print(tabela["Batata"])  
8  ''  
9  Saída:  
10  5.8  
11  ''
```

Fonte: autoria própria.

- Dicionários são utilizados como estruturas de registros de base de dados.

# Dicionários

## DICIONÁRIO:

Quando se atribui um valor a determinada chave, duas possibilidades podem ocorrer:

- Se a chave já existir – O valor associado é alterado para o novo valor;
- Se a chave não existir – A nova chave será adicionada ao dicionário.

```
1  notaProva = {  
2      "Maria":8.5,  
3      "Ana":9.2,  
4  }  
5  print(notaProva)  
6  notaProva["Maria"] = 8.7  
7  print(notaProva)  
8  notaProva["José"] = 7.8  
9  print(notaProva)  
10 '''  
11 Saída:  
12 {'Maria': 8.5, 'Ana': 9.2}  
13 {'Maria': 8.7, 'Ana': 9.2}  
14 {'Maria': 8.7, 'Ana': 9.2, 'José': 7.8}  
15 '''
```

Fonte: autoria própria.

# Dicionários

## DICIONÁRIO:

CUIDADO ao acessar (ler) o valor de uma chave:

- Antes de acessar os dados, é necessário verificar se uma chave já existe, pois, caso a chave não exista, uma exceção do tipo **KeyError** é ativada (e, então, o programa é encerrado).

```
1  notaProva = {
2      "Maria":8.5,
3      "Ana":9.2,
4  }
5  print(notaProva["Maria"])
6  print(notaProva["José"])
7  '''
8  Saída:
9  8.5
10 Traceback (most recent call last):
11     File "c:/Python/Programas/Geral/programa.py", line 6, in <module>
12         print(notaProva["José"])
13     KeyError: 'José'
14 '''
```

# Dicionários

## DICIONÁRIO:

Para verificar se uma chave pertence a um dicionário, pode ser usado o operador *in*:

```
1  notaProva = {  
2      "Maria":8.5,  
3      "Ana":9.2,  
4  }  
5  if ("Maria" in notaProva): print(notaProva["Maria"])  
6  if ("José" in notaProva): print(notaProva["José"])  
7  if ("Ana" in notaProva): print(notaProva["Ana"])  
8  '''  
9  Saída:  
10  8.5  
11  9.2  
12  '''
```

Fonte: autoria própria.



# Dicionários

## DICIONÁRIO:

Outro forma de retornar uma lista com todas as chaves ou, mesmo, uma lista com todos os valores do dicionário é utilizando os métodos:

- <dicionário>.keys() – Retorna as chaves que pertencem ao dicionário;
- <dicionário>.values() – Retorna os valores que pertencem ao dicionário.

```
1  notaProva = {
2      "Maria":8.5,
3      "Ana":9.2,
4  }
5  lstCh = notaProva.keys()
6  lstVl = notaProva.values()
7  print(lstCh)
8  print(lstVl)
9  '''
10 Saída:
11 dict_keys(['Maria', 'Ana'])
12 dict_values([8.5, 9.2])
13 '''
```

Fonte: autoria própria.

# Dicionários

## DICIONÁRIO:

- Para acessar os elementos de um dicionário podemos utilizar um comando de repetição **for**.

```
1  notaProva = {  
2      "Maria":8.5,  
3      "Ana":9.2,  
4  }  
5  lstCh = notaProva.keys()  
6  lstVl = notaProva.values()  
7  for ch in lstCh:  
8      print(ch)  
9  for vl in lstVl:  
10     print(vl)  
11     ''  
12 Saída:  
13 Maria  
14 Ana  
15 8.5  
16 9.2  
17 ''
```

```
1  notaProva = {  
2      "Maria":8.5,  
3      "Ana":9.2,  
4  }  
5  for chv in notaProva:  
6      print (chv,"::",notaProva[chv])  
7  ''  
8 Saída:  
9 Maria :: 8.5  
10 Ana :: 9.2  
11 ''
```

Fontes: autoria própria.

# Dicionários

## DICIONÁRIO:

- Para acessar os elementos de um dicionário podemos também utilizar um comando **for** associado ao método **items()**.

```
1  tabela = {  
2      "Alface":2.50,  
3      "Batata":5.80,  
4      "Tomate":4.30,  
5      "Feijão":2.20 }  
6  for x, y in tabela.items():  
7      print(x,"::",y)  
8  ''  
9  Saída:  
10 Alface :: 2.5  
11 Batata :: 5.8  
12 Tomate :: 4.3  
13 Feijão :: 2.2  
14 ''
```

# Dicionários

## DICIONÁRIO:

- Para apagar um item do dicionário, utiliza-se o comando ***del***.

```
1  notaProva = {
2      "Maria":8.5,
3      "Ana":9.2,
4      "José":7.8,
5  }
6  print(notaProva)
7  del notaProva["Ana"]
8  print(notaProva)
9  '''
10 Saída:
11 {'Maria': 8.5, 'Ana': 9.2, 'José': 7.8}
12 {'Maria': 8.5, 'José': 7.8}
13 '''
```

# Dicionários

## DICIONÁRIOS COM LISTAS:

- Em Python, é possível ter um dicionário nos quais cada chave está associada a uma lista de valores ou, até mesmo, a outro dicionário.

Exemplo: imagine uma tabela de estoque de mercadorias na qual se tem a informação da quantidade em estoque e o preço individual. Neste caso, o nome do produto poderá ser a chave, e a lista (uma lista por chave) conterá os seus valores onde o primeiro elemento dessa lista poderá ser a quantidade em estoque, e o segundo, o preço do produto.

```
1  estoquePreco = {
2      "Canetas" : [100, 3.50],
3      "Borrachas" : [200, 1.80],
4      "Cadernos" : [80, 8.00],
5      "Pastas" : [70, 2.50] }
6  print("Qtdd. de Borrachas:")
7  print(estoquePreco["Borrachas"][0])
8  print("Preço da Borracha (R$):")
9  print(estoquePreco["Borrachas"][1])
10 ''
11 Saída:
12 Qtdd. de Borrachas:
13 200
14 Preço da Borracha (R$):
15 1.8
16 ''
```

Fonte: autoria própria.

# Interatividade

Tem-se o dicionário a seguir (ed\_dic) montado da seguinte forma:

```
ed_letr = [[["X","D"],["U","W"],["N"],["K","A"],[["P","F"],["Z"]]]
ed_plan = [["Mercúrio", "Vênus"],["Terra", ["Marte", "Júpiter"]],["Saturno", "Urano", "Netuno"],["Plutão"]
ed_dias = ["segunda", "terça", "quarta", "quinta", "sexta", "sábado", "domingo"]
ed_medi = [[3.9201, 2.3800, 6.1932], 2.7618, [1.7260, 0.1720, 5.5000]]
ed_dic = {"Letras":ed_letr, "Planetas":ed_plan, "Medidas":ed_medi, "Semana":ed_dias}
x = ????
```

Qual dos comandos a seguir faz com que se consiga atribuir a uma variável “x”, o valor “Júpiter” do dicionário anterior?

- a) `x = ed_dic["Planetas"][1][1][1].`
- b) `x = ed_dic[1][1][1][1].`
- c) `x = ed_dic["Júpiter"].`
- d) `x = ed_dic["Planetas"][4].`
- e) `x = ed_dic["Planetas"][2][2].`

# Resposta

Tem-se o dicionário a seguir (ed\_dic) montado da seguinte forma:

```
ed_letr = [[["X","D"],["U","W"],["N"],["K","A"],[["P","F"],"Z"]]  
ed_plan = [["Mercúrio", "Vênus"],["Terra", ["Marte", "Júpiter"]],["Saturno", "Urano", "Netuno"],"Plutão"]  
ed_dias = ["segunda", "terça", "quarta", "quinta", "sexta", "sábado", "domingo"]  
ed_medi = [[3.9201, 2.3800, 6.1932], 2.7618, [1.7260, 0.1720, 5.5000]]  
ed_dic = {"Letras":ed_letr, "Planetas":ed_plan, "Medidas":ed_medi, "Semana":ed_dias}  
x = ????
```

Qual dos comandos a seguir faz com que se consiga atribuir a uma variável “x”, o valor “Júpiter” do dicionário anterior?

- a) **x = ed\_dic["Planetas"][1][1][1].**
- b) x = ed\_dic[1][1][1][1].
- c) x = ed\_dic["Júpiter"].
- d) x = ed\_dic["Planetas"][4].
- e) x = ed\_dic["Planetas"][2][2].

# Introdução aos Módulos

## MÓDULOS:

- Quando estamos desenvolvendo programas complexos com várias funcionalidades e funções, é interessante que separemos os tipos de funções em arquivos diversos, organizando o nosso programa em partes administráveis de fácil acesso e rápida manutenção em futuras implementações;
- Assim, um módulo é o que podemos chamar de uma “**Biblioteca de Códigos ou Instruções Prontas**” (e de constantes), ou seja, um ou mais arquivos contendo uma série de funções (ou “definições”) e valores que iremos utilizar em nosso aplicativo, e que podem ser reutilizados por várias partes do mesmo ou, ainda, por outros aplicativos que venhamos a desenvolver no futuro;
  - Para construir um módulo basta criar um arquivo com as suas funções e códigos em Python, e salvá-lo com o nome do módulo, e com a extensão.py (**nomeDoModulo.py**).



# Introdução aos Módulos

## MEU PRIMEIRO MÓDULO:

- Vamos criar um módulo (uma biblioteca de funções ou definições simples), e ver como utilizamos este módulo em outros programas;
- Para criar um módulo, vamos criar um arquivo .py (programa Python) como sempre criamos ao longo deste curso. (nomeDoModulo.py).

Exemplo:

Vamos criar o arquivo *meuModulo.py* e, nele, vamos criar o dicionário e a função ao lado:



```
Python > meuModulo.py > ...
1  dicionario1 = {
2      "chave1": "valor1",
3      "chave2": "valor2",
4      "chave3": "valor3"
5  }
6
7  def imprime(texto):
8      print(texto)
9
```

# Introdução aos Módulos

## MEU PRIMEIRO MÓDULO (continuação):

Agora, vamos criar o programa (programa.py) que irá se utilizar do módulo criado e rodar a função nele criada:



```
meuModulo.py  programa.py X

Python > programa.py
1  import meuModulo
2  # rodando a função "imprime" daquele módulo
3  # ..enviando um texto qualquer.
4  meuModulo.imprime("Rodando a Função 'imprime'...")
5  meuModulo.imprime(meuModulo.dicionario1["chave2"])
6  '''
7  Saída:
8  Rodando a Função 'imprime'...
9  valor2
10  '''
```

Fonte: autoria própria.

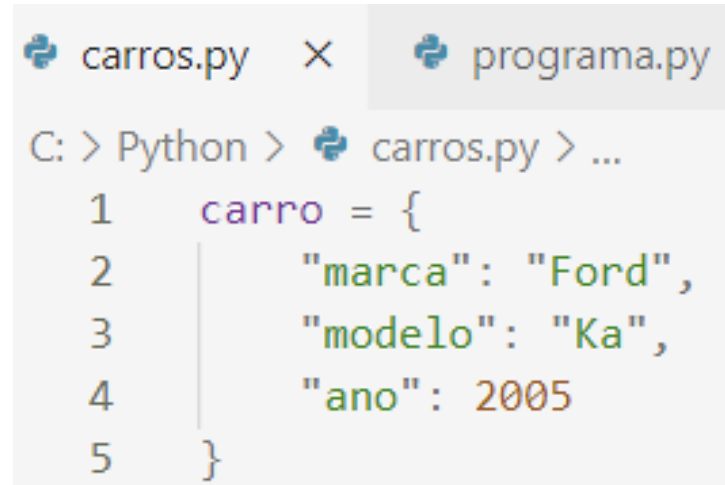
- Obs.: neste exemplo, os dois arquivos (meuModulo.py e programa.py) estão no mesmo diretório.

# Introdução aos Módulos

## APELIDOS DE MÓDULOS:

- É possível dar “apelidos” aos módulos usando a instrução `as`, que pode ser um nome menor, o que, de certa forma, economiza e agiliza a digitação da programação, caso o módulo seja muito utilizado.

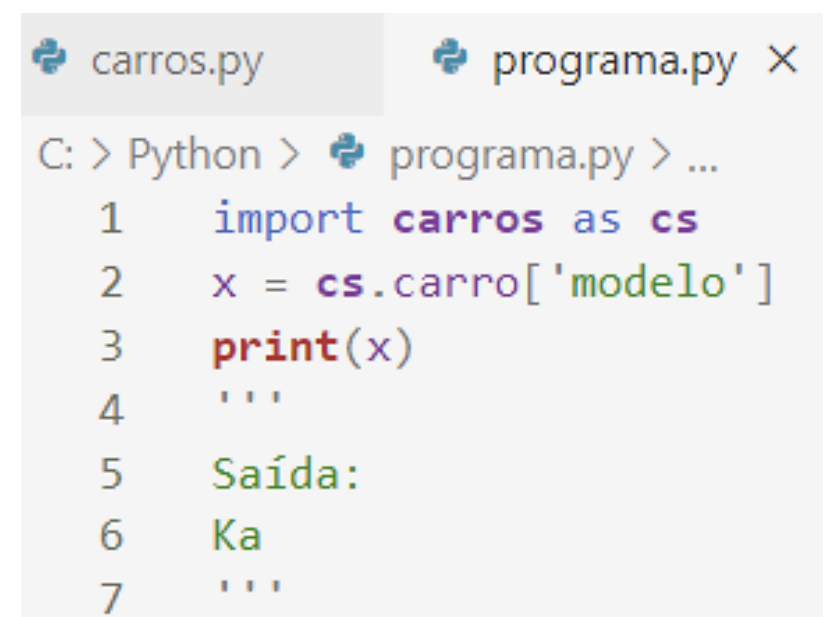
Como exemplo, vamos criar os arquivos “carros.py” e “programa.py” (no mesmo diretório) de acordo com o código a seguir:



The screenshot shows a code editor with two tabs: 'carros.py' and 'programa.py'. The 'carros.py' tab is active. The code in 'carros.py' is as follows:

```
C: > Python > carros.py > ...  
1  carro = {  
2      "marca": "Ford",  
3      "modelo": "Ka",  
4      "ano": 2005  
5  }
```

Fontes: autoria própria.



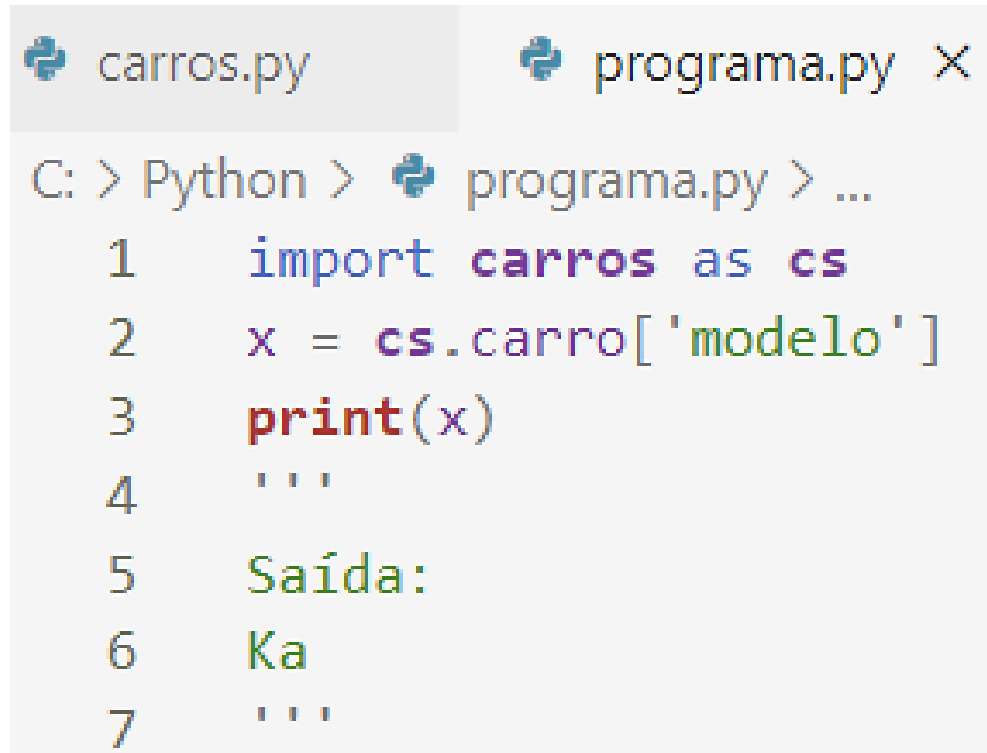
The screenshot shows the same code editor with the 'programa.py' tab active. The code in 'programa.py' is as follows:

```
C: > Python > programa.py > ...  
1  import carros as cs  
2  x = cs.carro['modelo']  
3  print(x)  
4  ''  
5  Saída:  
6  Ka  
7  ''
```

# Introdução aos Módulos

## APELIDOS DE MÓDULOS:

- Percebam que no arquivo “programa.py” importamos o módulo “carros” e demos a este módulo um apelido: “cs”;
- Desta forma, será a partir deste apelido que deveremos acionar e acessar qualquer elemento (variável ou função) daquele módulo (como no exemplo dado).



```
C: > Python > programa.py > ...  
1  import carros as cs  
2  x = cs.carro['modelo']  
3  print(x)  
4  '''  
5  Saída:  
6  Ka  
7  '''
```

# Introdução aos Módulos

## *From... Import:*

- Podemos escolher importar somente partes de um módulo; isso ajuda ao Python otimizar o que vai ou não vai para a memória da máquina. É muito comum, também, utilizar-se, somente, uma ou duas funções de um módulo num programa. Neste caso, uma boa prática é utilizar as instruções *from...import*.

Exemplo:

```
C: > Python > prog2.py > ...
```

```
1  import math
2  x = math.cos(3.141592653589793)
3  print(x)
4  '''
5  Saída:
6  -1.0
7  '''
```

Fontes: autoria própria.

```
C: > Python > prog2.py > ...
```

```
1  from math import cos, sin, pi
2  x = cos(pi)
3  y = sin(pi)
4  print("x =", x, " : y =", y)
5  '''
6  Saída:
7  x = -1.0   : y = 1.2246467991473532e-16
8  '''
```

# O Módulo SYS

## SYS:

- Existem vários módulos padrão que já vêm com a instalação do Python;
- Um deles é o módulo `sys`, que permite o acesso a algumas variáveis usadas ou mantidas pelo interpretador do Python, e também fornece as funções e as variáveis usadas para manipular diferentes partes do ambiente de “tempo de execução” do Python;
- (Obs.: o módulo `sys` é diferente do módulo `os`, este último utilizado para manipular o sistema operacional);
- O módulo `sys` contém funções que permitem saber, entre outras coisas: qual a plataforma do dispositivo que está rodando o código; obter os caminhos de sistema que o interpretador Python utiliza; quais módulos foram importados para o programa; qual a versão do Python etc.;
  - Ele também dispõe de acesso à saída e à entrada de informação, e reconhecimento de erros padrão (*stdin*, *stdout* e *stderr*), além de ter uma função para encerrar a execução de programas `sys.exit()`.

# O Módulo SYS

## SYS (continuação):

- Muitas vezes, alguns programas em Python precisam processar argumentos que são passados via terminal, ao acionarmos uma aplicação. A lista de argumentos passados na linha de comandos fica armazenada numa variável chamada “argv” que pode ser acessada pelo módulo sys.

### Exemplo:

Suponha que tenhamos o programa ao lado (cujo arquivo é “prog04.py”), e a partir do “*Prompt* de Comando” executemos o seguinte comando:

```
python prog04.py 30 40 50
```

```
1  import sys
2  x = sys.argv
3  print(type(x))
4  for n in x:
5      print(n)
6  ...
7  <class 'list'>
8  prog04.py
9  30
10 40
11 50
12 ...
```

Fonte: autoria própria.

# Instalando Módulos

## Módulos:

- A comunidade Python é muito ativa no desenvolvimento de módulos novos, além da manutenção nos módulos já existentes;
- Existe uma infinidade de módulos prontos em Python, como uma grande biblioteca de módulos e funções (uma biblioteca Python é uma coleção de módulos);
- Vamos rever o processo de instalação de módulos através da ferramenta pip (*Package Installer for Python*). Nas versões mais novas, ela vem embutido na instalação do pacote padrão do Python.

Observação: antes de usar o pip, é recomendável se atualizar o programa (Python) usando o comando no terminal:

```
python -m pip install --upgrade pip
```

Exemplo de comando para a instalação de módulos, como o da biblioteca *Arrow* (com funções de data e horário):

```
pip install arrow
```



# Arquivos Python compilados

## Arquivos “*byte-compiled*” (.pyc):

- O Python é uma linguagem interpretada, isso quer dizer que ele lê, em tempo real, os programas fontes escritos em texto, e os traduz para o código de máquina no momento da execução;
- Importar um módulo muito grande e traduzi-lo em código de máquina, no momento da execução, é uma tarefa que pode durar um tempo considerável e que pode prejudicar o tempo de processamento do programa;
- Para minimizar esse problema de desempenho, o Python permite algumas otimizações criando um arquivo chamado *byte-compiled*, que é um arquivo com a extensão .pyc que contém um código intermediário já otimizado, que é mais fácil de ser traduzido para o código de máquina;
  - Os arquivos .pyc são independentes de sistemas operacionais (ou plataformas – podem ser movidos do Windows para o Linux, por exemplo);
  - Esses arquivos são criados, automaticamente, na primeira execução de cada programa, caso eles, ainda, não existam.

# Interatividade

O que é um módulo pronto da linguagem Python?

- a) É um conjunto de funções que um programador cria, especificamente, para determinado sistema e que pode ser utilizado por qualquer função daquele sistema.
- b) É todo módulo que importamos em um programa utilizando o comando “*import*”.
- c) É qualquer arquivo em Python pré-compilado que agiliza (torna mais rápida) a execução de sistemas.
- d) É um conjunto de arquivos que geramos em Python e que, juntos, completam todo o sistema.
- e) É uma biblioteca de códigos, ou de instruções pré-programadas, que já existe com a instalação do Python.

# Resposta

O que é um módulo pronto da linguagem Python?

- a) É um conjunto de funções que um programador cria, especificamente, para determinado sistema e que pode ser utilizado por qualquer função daquele sistema.
- b) É todo módulo que importamos em um programa utilizando o comando “*import*”.
- c) É qualquer arquivo em Python pré-compilado que agiliza (torna mais rápida) a execução de sistemas.
- d) É um conjunto de arquivos que geramos em Python e que, juntos, completam todo o sistema.
- e) É uma biblioteca de códigos, ou de instruções pré-programadas, que já existe com a instalação do Python.

# Banco de Dados

## BANCO DE DADOS (BD):

- Banco de Dados – sinônimo de SGBD (Sistema de Gerenciamento de Banco de Dados), como por exemplo: *MySQL*, *Oracle*, *SQL Server* etc.;
- SGBD – sistema que gerencia vários Bancos de Dados (que, por sua vez, é uma coleção de tabelas, índices, procedimentos etc.);
- Tabelas – dentro das tabelas estão os dados (as informações guardadas), manipulados por uma linguagem chamada SQL (*Structured Query Language* – Linguagem Estruturada de Consulta).

### Observação:

- Nosso objetivo, aqui, é entender o básico para que seja possível uma conexão entre um programa em Python e o SGBD *MySQL*. Não iremos entrar em detalhes e nem em conceitos, sobre o Banco de Dados Relacionais ou sobre a Linguagem SQL (obs.: existem outras disciplinas, neste curso, focadas, somente, no ensino de Banco de Dados e de SQL).

# Banco de Dados

## BANCO DE DADOS (BD) – continuação:

- O Python possui módulos específicos para tratar as conexões com o Banco de Dados. Neste curso, vamos utilizar o *MySQL*, que é um SGBD com uma versão gratuita e que está em crescente utilização pelas empresas.

Antes de começar a manipular o banco de dados *MySQL* com o Python, temos que completar 3 passos:

- Instalar o *MySQL*;
- Instalar o módulo conector do Python;
- Construir um programa que se conecta ao SGBD.

# Banco de Dados

## BANCO DE DADOS (BD) – continuação:

- A instalação do MySQL depende do sistema operacional;
- Faça o *download* da última versão no *site* oficial do MySQL;
- Obs.: escolha a versão gratuita, que tem o nome de “*MySQL Community*”, no *link*: <https://dev.mysql.com/downloads/>;
- Siga o processo de instalação, escolhendo uma senha para o usuário “*root*” (atenção: anote a senha para não esquecer).

# Banco de Dados

## BANCO DE DADOS (BD) – continuação:

- A instalação do módulo conector é feita pelo utilitário *pip*.

Digite no terminal (*Prompt* de Comando):

```
pip install mysql-connector-python
```

# Banco de Dados

## ACESSANDO UM SGBD:

- Construção de um programa que se conecta ao *MySQL*;
- A criação de um Banco de Dados, das tabelas, e a manipulação de dados, pode ser feita, diretamente, via código em Python;
- Para acessar o SGBD *MySQL*, gera-se uma conexão com o *MySQL* onde se indica qual é o servidor (o *host*), o usuário (normalmente, o usuário *root*), a senha e o nome da Base de Dados (quando esta já existir).



# Banco de Dados

## ACESSANDO UM SGBD:

Exemplo de um código para acessar o *MySQL* (mas não um BD específico):

```
C: > Python > acessoBD.py > ...
1  import mysql.connector
2  # Observações:
3  # - localhost = sua máquina local
4  # - root = usuário administrador do banco
5  # - password = sua senha do MySQL
6  bd = mysql.connector.connect(
7      host="localhost",
8      user="root",
9      password="<sua senha>"
10 )
11 print(bd)
12 '''
13 Saída:
14 <mysql.connector.connection.MySQLConnection object at 0x034BFC10>
15 '''
```

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD:

Para iniciar a leitura, gravação, alteração ou exclusão de dados, em um BD, devemos, antes, realizar algumas tarefas utilizando a conexão que já vimos como fazer e, então:

- Criar o Banco de Dados;
- Conectar o programa ao Banco de Dados;
- Criar a tabela (ou as tabelas).

Observação:

- Nestes exemplos estaremos utilizando o Usuário Administrador do MySQL (*root*) para fazer essa conexão;
  - No entanto, profissionalmente, isso não é uma boa prática de segurança. Normalmente, nas aplicações profissionais são criados “Usuários de Serviço”, gerados, somente, com as permissões necessárias, diminuindo, assim, os riscos de problemas com a segurança nas Bases de Dados das empresas.

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Exemplo de criação de um BD, cujo nome será: primeirobanco:

```
C: > Python >  acessoBD2.py > ...  
1  import mysql.connector  
2  bd = mysql.connector.connect(  
3      host="localhost",  
4      user="root",  
5      password="<sua senha>"  
6  )  
7  bd_cursor = bd.cursor()  
8  bd_cursor.execute("CREATE DATABASE primeirobanco")
```


Fonte: autoria própria.

- Obs.: o objeto *MySQLCursor()* interage, diretamente, com a conexão criada no *MySQL*, o que permitirá executar os comandos SQL no BD.

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Exemplo para listar os Bancos de Dados existentes no SGBD:

```
C: > Python >  acessoBD3.py > ...  
1  import mysql.connector  
2  bd = mysql.connector.connect(  
3      host="localhost",  
4      user="root",  
5      password="<sua senha>"  
6  )  
7  bd_cursor = bd.cursor()  
8  # o comando SQL: SHOW DATABASES lista os bancos  
9  bd_cursor.execute("SHOW DATABASES")  
10 for banco in bd_cursor:  
11     print(banco)  
12 '''  
13 Saída:  
14 ('information_schema',)  
15 ('mysql',)  
16 ('performance_schema',)  
17 ('primeirobanco',)  
18 ('sys',)  
19 '''
```

Fonte: autoria própria.

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Existem duas formas mais comuns de se conectar com uma Base de Dados específica para criar os seus elementos (como as suas tabelas e os seus índices):

- Utilizando o comando do *MySQL* “USE <nome\_da\_base\_de\_dados>”;
- Adicionando no comando de conexão (no comando “*connect*” do *mysql.connector*), o argumento (ou parâmetro) “*database*”.
- É esta segunda forma que vamos utilizar em nossos exemplos.

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Exemplo com argumento “*database*”  
no comando de conexão:

Neste exemplo, vamos criar uma  
tabela chamada “alunos”, para que,  
depois, possamos inserir  
informações nela:

```
C: > Python > acessoBD4.py > ...
1  import mysql.connector
2  bd = mysql.connector.connect(
3      host="localhost",
4      user="root",
5      password="<sua senha>",
6      database="primeirobanco"
7  )
8  bd_cursor = bd.cursor()
9  strSQL = "CREATE TABLE alunos "
10 strSQL += "(id INT AUTO_INCREMENT PRIMARY KEY, "
11 strSQL += "nome VARCHAR(255), "
12 strSQL += "endereco VARCHAR(255))"
13 bd_cursor.execute(strSQL)
14 # Para vermos quais as Tabelas Criadas
15 bd_cursor.execute("SHOW TABLES")
16 for tabela in bd_cursor:
17     print(tabela)
18 '''
19 Saída:
20 ('alunos',)
21 '''
```

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

- Com o Banco de Dados criado, vamos fazer uma inclusão de informações na tabela “alunos”;
- Obs.: no exemplo ao lado, a saída do *print* do programa será:  
4 registros inseridos.

```
C: > Python > acessoBD5.py > ...
1  import mysql.connector
2  bd = mysql.connector.connect(
3      host="localhost",
4      user="root",
5      password="<sua senha>",
6      database="primeirobanco"
7  )
8  bd_cursor = bd.cursor()
9  sql = "INSERT INTO alunos (nome, endereco) VALUES (%s, %s)"
10 valores = [
11     ('Ana', 'Rua Joao do Pulo, 76'),
12     ('Beatriz', 'Alameda Jau, 897'),
13     ('Carlos', 'Avenida Rita Lia, 472'),
14     ('Daniel', 'Rua Oculta, sem numero'),
15 ]
16 bd_cursor.executemany(sql, valores)
17 # Importante: o comando commit() é necessário para salvar
18 # ..as alterações de informações no banco e dados
19 bd.commit()
20 # Comando que imprime o que acabou de ser feito
21 print(bd_cursor.rowcount, "registros inseridos.")
```

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Exemplo de leitura (consulta)  
no Banco de Dados:

- Obs.: o comando “*fetchall*” do *MySQLCursor()* retorna uma lista de tuplas, com os dados existentes na tabela.

```
C: > Python > acessoBD6.py > ...
1  import mysql.connector
2  bd = mysql.connector.connect(
3      host="localhost",
4      user="root",
5      password="<sua senha>",
6      database="primeirobanco"
7  )
8  bd_cursor = bd.cursor()
9  bd_cursor.execute("SELECT * FROM alunos")
10 alunos = bd_cursor.fetchall()
11 for aluno in alunos:
12     print(aluno)
13 '''
14 Saída:
15 (1, 'Ana', 'Rua Joao do Pulo, 76')
16 (2, 'Beatriz', 'Alameda Jau, 897')
17 (3, 'Carlos', 'Avenida Rita Lia, 472')
18 (4, 'Daniel', 'Rua Oculta, sem numero')
19 '''
```

Fonte: autoria própria.



# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Exemplo de um *update*  
(alteração de dados já  
existentes na tabela):

```
C: > Python > acessoBD6.py > ...
1  import mysql.connector
2  bd = mysql.connector.connect(
3      host="localhost",
4      user="root",
5      password="<sua senha>",
6      database="primeirobanco"
7  )
8  bd_cursor = bd.cursor()
9  sql = "UPDATE alunos "
10 sql += "SET endereco = 'Rua das Camélias, 50' "
11 sql += "WHERE nome = 'Ana'"
12 bd_cursor.execute(sql)
13 bd.commit()
14 print(bd_cursor.rowcount, "registro alterado.")
15 '''
16 Saída:
17 1 registro alterado.
18 '''
```

# Banco de Dados

## LEITURA, GRAVAÇÃO, ALTERAÇÃO E EXCLUSÃO EM UM BD (continuação):

Exemplo de exclusão ou deleção de um registro:

```
C: > Python > acessoBD7.py > ...
1  import mysql.connector
2  bd = mysql.connector.connect(
3      host="localhost",
4      user="root",
5      password="<sua senha>",
6      database="primeirobanco"
7  )
8  bd_cursor = bd.cursor()
9  sql = "DELETE FROM alunos WHERE nome = 'Daniel'"
10 bd_cursor.execute(sql)
11 bd.commit()
12 print(bd_cursor.rowcount, "registro deletado.")
13 '''
14 Saída:
15 1 registro deletado.
16 '''
```

# Interatividade

Quando executamos uma leitura com a instrução “*select*” do SQL, em um Banco de Dados, a partir de um “cursor” (uma estrutura de controle que permite percorrer sobre os registros lidos de um Banco de Dados) qual é o comando da linguagem Python que retorna uma lista de tuplas com os dados obtidos do Banco de Dados?

- a) `lista = fetchall(cursor).`
- b) `lista = cursor.getlist().`
- c) `lista = cursor.fetchall().`
- d) `lista = getlist(cursor).`
- e) `lista = cursor.getall().`

## Resposta

Quando executamos uma leitura com a instrução “*select*” do SQL, em um Banco de Dados, a partir de um “cursor” (uma estrutura de controle que permite percorrer sobre os registros lidos de um Banco de Dados) qual é o comando da linguagem Python que retorna uma lista de tuplas com os dados obtidos do Banco de Dados?

- a) `lista = fetchall(cursor).`
- b) `lista = cursor.getlist().`
- c) `lista = cursor.fetchall().`
- d) `lista = getlist(cursor).`
- e) `lista = cursor.getall().`

# Referências

- Imagens das telas do VS *Code*: autoria própria.

**ATÉ A PRÓXIMA!**