FIAP ON

PYTHON

LISTAS E FUNÇÕES

HUMBERTO DELGADO DE SOUSA

3

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

3 LISTAS E FUNÇÕES

Neste capítulo, abordaremos dois assuntos que são importantíssimos para o desenvolvimento Python, são eles: as listas e as funções. Você verá como as listas maximizam a captura dos dados e permitem, com as variáveis e os laços, o desenvolvimento de uma aplicação que atenda às reais necessidades de qualquer usuário final, tornando o seu código dinâmico.

Já as funções serão as responsáveis por nos acompanhar durante todo o restante do curso, já utilizamos algumas funções próprias do Python, como: input() e print(), por exemplo. Entretanto, as funções possuem muito mais poderes. Podemos utilizar funções de outros programadores, importando-as dentro dos nossos projetos e criar as nossas próprias funções (assunto que será abordado neste capítulo) para que outros programadores possam utilizá-las.

Enfim, vamos embarcar nesta jornada, pois muitas novidades virão e um mundo repleto de possibilidades seguirá se abrindo... print (“Let’s go...”)!

3.1 Projeto

Antes de iniciarmos o nosso assunto sobre listas, vamos primeiramente abrir o nosso projeto criado no PyCharm (MeusProjetos_Python). Para isso, abra o PyCharm, e logo surgirá a seguinte caixa de mensagem:

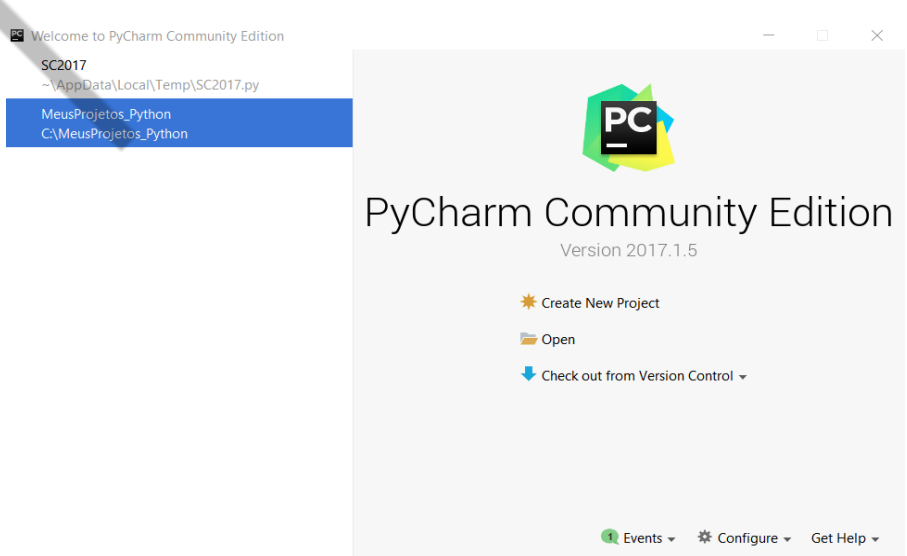

Figura 3.1 – Tela de abertura do PyCharm
Fonte: Elaborado pelo autor (2017)

Clique sobre o seu projeto, conforme destacado em azul na imagem acima, e logo deverá aparecer a janela abaixo, com o código que criamos no final do capítulo anterior, demonstrando, assim, que o PyCharm está pronto para ser utilizado.

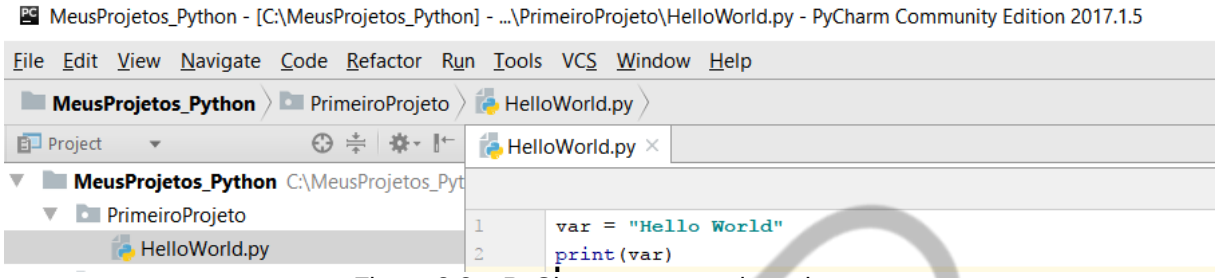

Figura 3.2 – PyCharm com o projeto aberto
Fonte: Elaborado pelo autor (2017)

3.2 Variável, precisa de uma ajuda?

Como já foi dito anteriormente, todas as linguagens possuem formas particulares para armazenamento de dados voláteis (dados que ficam na memória do computador enquanto o sistema estiver aberto ou não forem sobrescritos). Falando de Python especificamente, vimos no capítulo anterior uma dessas formas, as variáveis. Entretanto, iremos passar por situações em que a variável não será o recurso mais indicado, pensando em desenvolver um código limpo e reduzido.

Por exemplo, iremos pesquisar em um conjunto de dados (banco de dados) todos os usuários que possuem acesso ao sistema para a realização de uma auditoria. Cada usuário seria identificado por uma variável, certo? Mas quantos usuários serão pesquisados? Precisaremos de quantas variáveis? Percebe que é impossível definirmos um valor exato? A solução, então, seria criarmos mil variáveis, pensando que o número mil jamais seria alcançado? Não, em vez disso, iremos criar uma única e singela lista que pode armazenar quantos usuários desejarmos.

Se ainda não ficou claro, vamos pensar em mais um exemplo: todas as máquinas da nossa clínica possuirão um gerador de “log”, que nada mais é do que um arquivo do tipo texto com informações capturadas em relação às ações do dia a dia. Podemos inserir no arquivo de log informações sobre qual usuário acessou aquele computador e em qual horário esteve ativo.

A fim de encontrarmos um computador que está recebendo alta rotatividade de acessos em um mesmo dia, iremos verificar o arquivo de log e recuperar os

usuários que utilizaram o computador durante o dia. Para isso, precisaremos armazenar em variáveis os usuários e os respectivos horários em que estiveram on-line para exibi-los na tela. De quantas variáveis iremos precisar para armazenar todos os acessos? Mais uma vez, não temos um valor exato. Qual a solução? Isso...utilizaremos listas!!!

Então, podemos concluir que as listas armazenam uma quantidade indeterminada de dados? Não é bem assim, pois existe o limite físico da memória do seu computador, mas esse limite é tão grande que jamais chegaremos a alcançá-lo dentro de uma situação profissional em que você faça uso das boas práticas e não tenha intenções reais em “estourar a memória” do computador. Com isso, podemos definir que as listas podem armazenar uma quantidade de dados indeterminados, mas de maneira finita.

Poderíamos fazer a seguinte analogia: a variável seria o corpo humano, e em cada corpo só é possível “armazenar” um único indivíduo. Já a lista seria como o planeta Terra, onde podemos armazenar vários indivíduos, um número finito é verdade, mas uma quantidade indeterminada e longe da capacidade máxima de armazenamento. Além disso, perceba que o indivíduo sempre armazenará um indivíduo, já o planeta Terra pode armazenar seres humanos, animais, vegetais, objetos e mais uma enorme diversidade, tudo junto e misturado... assim também são as listas. Vamos para a prática agora...

3.3 Listas na prática

As listas, assim como as variáveis, devem seguir as regras e a padronização em relação aos seus identificadores (nomes), ou seja, nada de usar os seguintes nomes para as listas: while, for, n@me, 1berto, entre outros identificadores inválidos. Monte no seu PyCharm a estrutura abaixo:

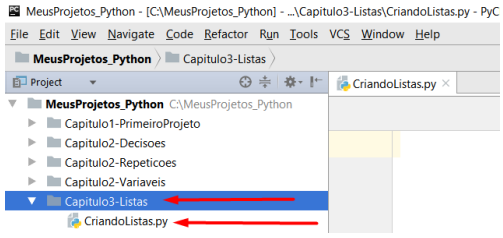

Figura 3.3 – PyCharm – estrutura para listas
Fonte: Elaborado pelo autor (2017)

Todas as listas podem ser inicializadas vazias ou com dados; caso você opte por criar uma lista vazia, deverá igualá-la, conforme é demonstrado no código abaixo.

```
#lista preenchida estaticamente
lista_estatica = ["xpto", True]

#lista preenchida dinamicamente
lista_dinamica = [input("Digite o usuário: "), bool(int(input("Está logado? ")))]

#lista vazia
lista_vazia=[]
```

Código-fonte 3.1 – Criação de listas
Fonte: Elaborado pelo autor (2017)

Observe que, em todos os casos, os valores de uma lista devem estar entre colchetes “[ ]”, mesmo quando a lista for declarada vazia.

Na “lista_dinamica”, o segundo item está passando por duas conversões. Isso ocorreu porque desejamos um dado do tipo “boolean”, ou seja, um dado booleano que pode possuir apenas os valores “True” ou “False”. Esse tipo de dado é utilizado normalmente para perguntas que possam ter como resposta somente os valores “sim” ou “não”, como é o caso do nosso exemplo, a pergunta “Está logado?” somente pode ter as respostas “sim” ou ”não”.

Como o *input* retorna uma *string*, devemos converter o dado para *int* (inteiro), para, então, posteriormente, convertê-lo para *bool* (booleano). Não podemos fazer a conversão diretamente de *string* para *bool*. O valor retornado será “False” somente se o número informado for igual a zero, qualquer outro valor trará o retorno “True”.

Veremos a seguir a aplicação de alguns recursos que serão úteis no seu dia a dia.

3.3.1 Adicionando dados em uma lista de maneira indeterminada

Vimos até o momento que as listas armazenam um número indeterminado de dados, e essa é uma das suas principais características. Veremos a seguir um exemplo prático dessa aplicação. Crie um novo arquivo chamado: “MetodoAppend.py” e digite o código abaixo:

```
inventario=[]
resposta="S"
while resposta=="S":
    inventario.append(input("Equipamento: "))
    inventario.append(float(input("Valor: ")))
    inventario.append(int(input("Número Serial: ")))
    inventario.append(input("Departamento: "))
    resposta=input("Digite \"S\" para continuar: ").upper()
```

Código-fonte 3.2 – Uso do método append()
Fonte: Elaborado pelo autor (2017)

No código acima, utilizamos o nosso conhecido comando “while”, que será responsável por seguir adicionando dados para a nossa lista “inventario” enquanto o usuário digitar “S”.

Perceba no código acima que utilizamos o método append() para adicionar novos itens em nossa lista. Podemos afirmar que a cada passagem dentro do “while”, quatro novos dados serão adicionados na lista (nome do equipamento, valor do equipamento, número serial do equipamento e o nome do departamento onde se encontra o equipamento).

Agora, vamos utilizar outro comando que aprendemos no capítulo anterior, o “for”. Ele é o mais indicado para percorrermos a lista e exibirmos tudo o que está armazenado nela, veja no código abaixo como é simples:

```
inventario=[]
resposta="S"
while resposta=="S":
    inventario.append(input("Equipamento: "))
    inventario.append(float(input("Valor: ")))
    inventario.append(int(input("Número Serial: ")))
    inventario.append(input("Departamento: "))
    resposta=input("Digite \"S\" para continuar: ").upper()

for elemento in inventario:
    print(elemento)
```

Código-fonte 3.3 – Exibindo dados da lista
Fonte: Elaborado pelo autor (2017)

A estrutura foreach nos permite definir um nome para cada elemento que ele encontrar na lista. No nosso caso, chamamos de “elemento” e, então, ele irá percorrer todas as posições da lista e exibir todos os valores. Poderíamos também, em vez de criar uma única lista, criar uma lista para cada dado, por exemplo, uma lista para equipamento, outra para valor, número serial e departamento, respectivamente. Vamos tentar? Crie um novo arquivo chamado: “MultiplasListas.py” e tente montar o código sozinho, antes de continuar a leitura.

Agora, veja como o seu código deve ficar:

```
equipamentos = []
valores = []
seriais = []
departamentos = []
resposta = "S"
while resposta == "S":
    equipamentos.append(input("Equipamento: "))
    valores.append(float(input("Valor: ")))
    seriais.append(int(input("Número Serial: ")))
    departamentos.append(input("Departamento: "))
    resposta = input("Digite \"S\" para continuar: ").upper()

for equipamento in equipamentos:
    print("Equipamento: ", equipamento)
```

Código-fonte 3.4 – Utilizando múltiplas listas
Fonte: Elaborado pelo autor (2017)

Repare agora que estamos utilizando quatro listas, mas que o nosso “for” somente mostra os nomes dos equipamentos, que é o conteúdo da lista “equipamentos”. E para saber os outros dados? Precisaríamos criar mais três laços com o “for”? A resposta é...não. Podemos utilizar os *índices*, como veremos no próximo tópico.

3.3.2 Um amigo chamado “índice”

O índice é o número que define onde está armazenado um elemento dentro de uma lista. Quando um primeiro append() é executado na lista, ele abre a posição 0 (zero) para armazenar o dado, quando ele for executado novamente, abrirá a posição 1 (um) para o próximo dado, e assim sucessivamente. Se armazenarmos três elementos em uma lista, teremos índices de 0 a 2. Podemos dizer que teremos na memória estruturas similares aos quadros abaixo:

Lista: equipamentos

0	Roteador
1	Impressora XPTO
2	Servidor Bell

Quadro 3.1 – Tabela representando a lista “equipamentos”
Fonte: Elaborado pelo autor (2017)

Lista: valores

0	4500.00
1	880.00
2	15000.00

Quadro 3.2 – Tabela representando a lista “valores”
Fonte: Elaborado pelo autor (2017)

Lista: seriais

0	123456
1	122377
2	000012

Quadro 3.3 – Tabela representando a lista “seriais”
Fonte: Elaborado pelo autor (2017)

Lista: departamentos

0	Rh
1	Rh
2	Cpd

Quadro 3.4 – Tabela representando a lista “departamentos”
Fonte: Elaborado pelo autor (2017)

Com isso, podemos dizer que, na posição 0 (zero), teremos os dados referentes ao equipamento “Roteador”, que:

- custa “4500.00” (valor que está na posição zero);
- possui o número serial: “123456” (serial que está na posição zero); e
- se encontra no departamento “rh” (departamento que está na posição zero), e assim sucessivamente para os outros equipamentos.

Com isso, podemos alterar o código do nosso “for”, da seguinte forma:

```
equipamentos = []
valores = []
seriais = []
departamentos = []
resposta = "S"
while resposta == "S":
    equipamentos.append(input("Equipamento: "))
    valores.append(float(input("Valor: ")))
    seriais.append(int(input("Número Serial: ")))
```

```
    departamentos.append(input("Departamento: "))
    resposta = input("Digite \"S\" para continuar: ").upper()

for indice in range(0,len(equipamentos)):
    print("\nEquipamento..: ", (indice+1))
    print("Nome.........: ", equipamentos[indice])
    print("Valor........: ", valores[indice])
    print("Serial.......: ", seriais[indice])
    print("Departamento.: ", departamentos[indice])
```

Código-fonte 3.5 – Utilizando múltiplas listas
Fonte: Elaborado pelo autor (2017)

A estrutura do nosso “for” mudou, agora não estamos trabalhando com base nos elementos diretamente, mas, sim, de acordo com o índice. Para a variável “índice” que criamos no “for”, será atribuído o valor de 0 até a quantidade de elementos que existirem dentro da nossa lista “equipamentos” (função “len()”), que obviamente será a mesma quantidade de elementos que existirão nas listas: valores, seriais e departamentos, conforme apresentado nas tabelas do tópico anterior, deste mesmo capítulo. Com o índice em mãos, poderemos recuperar os dados de todas as nossas listas e a saída dos dados ficará muito mais apresentável.

3.3.3 Um pouco mais sobre as listas

Ainda com listas, podemos pesquisar um determinado dado, digite o código abaixo, no final do nosso arquivo “MultiplasListas.py”:

```
busca=input("\nDigite o nome do equipamento que deseja buscar: ")
for indice in range(0,len(equipamentos)):
    if busca==equipamentos[indice]:
        print("Valor..: ", valores[indice])
        print("Serial.:", seriais[indice])
```

Código-fonte 3.6 – Pesquisando dados em lista
Fonte: Elaborado pelo autor (2017)

Observamos acima que, dentro do “for”, montamos uma tomada de decisão simples, cuja função será comparar o conteúdo da variável “busca” com todos os elementos que estiverem armazenados dentro da lista “equipamentos”. Quando ele encontrar um valor igual, irá exibir, então, o valor e o número serial desse equipamento.

Vamos imaginar algumas outras situações:

- **Situação 1:** todos os equipamentos “impressora” receberão uma depreciação (desvalorização após certo período) de 10%. Monte o código que seria responsável por alterar o valor de todos os equipamentos “impressora”.
- **Situação 2:** um equipamento com um determinado número serial foi danificado e será descartado. Precisamos eliminar esse equipamento. **Dica:** para eliminar um item de uma lista, você utilizará o comando “del”. Exemplo: del lista[<indice>]

Tente elaborar o código das duas situações propostas. Veja a seguir os códigos que resolveriam essas tarefas.

```
depreciacao=input("\nDigite o nome do equipamento que será depreciado: ")
for indice in range(0,len(equipamentos)):
    if depreciacao==equipamentos[indice]:
        print("Valor antigo: ", valores[indice])
        valores[indice] = valores[indice] * 0.9
        print("Novo valor: ", valores[indice])
```

Código-fonte 3.7 – Solução da Situação 1
Fonte: Elaborado pelo autor (2017)

Como podemos observar no código acima, quando igualamos uma lista (acompanhada de um índice), sobrescrevemos o conteúdo do dado na posição especificada pelo índice, ou seja:

departamentos[0]=”Teste” => essa linha seria responsável por substituir pela string “Teste” o dado que está na posição zero.

Para a solução da segunda situação, o seu código deve ser semelhante a:

```
serial=int(input("\nDigite o serial do equipamento que será excluido: "))
for indice in range(0, len(departamentos)):
    if seriais[indice]==serial:
        del departamentos[indice]
        del equipamentos[indice]
        del seriais[indice]
        del valores[indice]
        break

for indice in range(0,len(equipamentos)):
    print("\nEquipamento..: ", (indice+1))
    print("Nome.........: ", equipamentos[indice])
    print("Valor........: ", valores[indice])
    print("Serial.......: ", seriais[indice])
    print("Departamento.: ", departamentos[indice])
```

Código-fonte 3.8 – Solução da Situação 2
Fonte: Elaborado pelo autor (2017)

No código acima proposto, repare que utilizamos o comando “break” dentro do “if”. O que significa que quando ele encontrar o valor desejado irá excluir o elemento da posição onde foi encontrado e depois irá sair do laço “for” (função do break). Isso porque, a partir do momento que excluímos um item, o índice poderá se perder, pois um elemento foi excluído e, consequentemente, o contador (laço) foi quebrado. O comando “break” serve também para forçar o fim dos laços com o comando “while”. Após o laço de exclusão, repetimos o laço para exibição, assim você poderá verificar que não existe mais o equipamento referente ao número serial que foi excluído.

Com esses exemplos, você aprendeu a pesquisar um dado dentro de uma lista e também a excluir um elemento específico dentro de uma lista.

Agora observe o seguinte detalhe: se o equipamento do nosso inventário possuir dezenas de características, teremos que criar também dezenas de listas, e assim o código irá aumentar assustadoramente. O nosso “for”, onde utilizamos o comando “del”, foi formado por quatro linhas para excluir o elemento. Se tivéssemos 20 listas, teríamos, então, não mais quatro linhas, mas, sim, 20 linhas, esse é apenas um exemplo. O que poderíamos fazer para resolver isso? Simples, poderíamos criar listas dentro de listas. Veremos isso no tópico a seguir.

3.3.4 Listas dentro de listas

Conforme dito no último tópico, podemos explorar as listas também, inserindo uma lista dentro da outra. Para isso, iremos manipular dois índices, um para se referir à lista externa (responsável por armazenar outras listas) e outro para se referir à lista interna (que está dentro de uma lista). Conforme representamos na imagem abaixo:

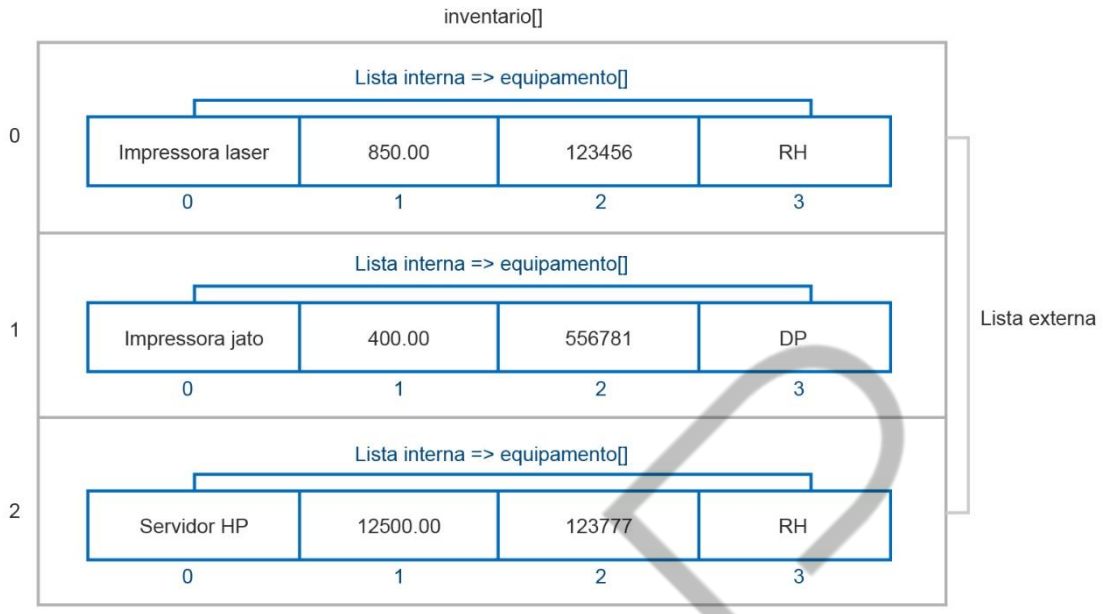

Figura 3.4 – Representação de listas dentro de lista
Fonte: Elaborada pelo autor (2017)

De acordo com a Figura 3.4, iremos definir que a lista “inventario[]” é a lista externa e armazenará nas suas posições uma lista chamada “equipamento[]”, que, por sua vez, fará o papel de lista interna. De acordo com a imagem, se utilizarmos os prints abaixo:

- print (inventario[0][2]) => será exibido o dado “123456”, porque ele retornará o elemento 0 da lista exterior e, dentro desse elemento, 0 irá buscar o dado que estiver na posição 2 da lista interior, ou seja, o serial (elemento 2) da impressora a laser. Outro exemplo:
- print (inventario[1][3]) => irá retornar “DP”, que é o conteúdo da posição 3 do segundo elemento da lista “inventario”.

Vamos agora criar um novo arquivo no PyCharm, chamado “ListasDentroDeLista.py”, no qual iremos montar o código com o mesmo objetivo do arquivo “MultiplasListas.py”, mas, dessa vez, iremos utilizar apenas duas listas (interna e externa – equipamento e inventário, respectivamente):

```
inventario=[]
resposta = "S"
while resposta == "S":
```

```
    equipamento=[input("Equipamento: "),
                 float(input("Valor: ")),
                 int(input("Número Serial: ")),
                 input("Departamento: ")]
    inventario.append(equipamento)
    resposta = input("Digite \"S\" para continuar: ").upper()

for elemento in inventario:
    print("Nome.........: ", elemento[0])
    print("Valor........: ", elemento[1])
    print("Serial.......: ", elemento[2])
    print("Departamento.: ", elemento[3])

busca=input("\nDigite o nome do equipamento que deseja buscar: ")
for elemento in inventario:
    if busca==elemento[0]:
        print("Valor..: ", elemento[1])
        print("Serial.:", elemento[2])

depreciacao=input("\nDigite o nome do equipamento que será depreciado: ")
for elemento in inventario:
    if depreciacao==elemento[0]:
        print("Valor antigo: ", elemento[1])
        elemento[1] = elemento[1] * 0.9
        print("Novo valor: ", elemento[1])

serial=int(input("\nDigite o serial do equipamento que será excluído: "))
for elemento in inventario:
    if elemento[2]==serial:
        inventario.remove(elemento)

for elemento in inventario:
    print("Nome.........: ", elemento[0])
    print("Valor........: ", elemento[1])
    print("Serial.......: ", elemento[2])
    print("Departamento.: ", elemento[3])
```

Código-fonte 3.9 – Exemplo de listas dentro de lista
Fonte: Elaborado pelo autor (2017)

Observe o código entre os dois arquivos, “ListasDentroDeListas.py” e “MultiplasListas.py”, e procure perceber qual está mais legível e mais simples de compreender. Caso você não tenha uma tendência em preferir códigos mais complexos, irá perceber que o código do arquivo “ListasDentroDeListas.py” se tornou mais legível que o anterior. A única regra é saber que, na posição 0, teremos o nome do equipamento; na posição 1, o valor; na posição 2, o serial; e na posição 3, o departamento.

Com o código do arquivo “ListasDentroDeListas.py”, não corremos o risco de exibir um dado de um equipamento com o dado de outro equipamento de maneira equivocada. Isso porque, antes, utilizávamos uma lista para cada dado, de maneira

independente, tornando o nosso código suscetível a enganos no retorno dos elementos das listas.

E, finalmente, se não colocarmos o “break” no nosso primeiro código, a aplicação tenderá a se encerrar abruptamente, ou “cair”, como é costumeiramente denominado, pois os índices podem se perder, e para que pudéssemos excluir dois elementos com um mesmo “serial”, teríamos um sério problema, tente fazer isso. Execute o arquivo “MultiplasListas.py”, acrescente dois equipamentos com o mesmo serial e tente excluí-los. Você verá que apenas um elemento será excluído e, se tirar o “break”, o sistema cairá e será interrompido.

Agora execute o arquivo “ListasDentroDeListas.py” e faça o mesmo, cadastre dois elementos com o mesmo serial e tente excluí-los. Repare que ele excluirá todos os encontrados coincidentes com o serial informado e não encerrará a aplicação de maneira abrupta.

Logo, trabalhar com listas dentro de listas pode parecer desafiador, mas acredite: em determinadas situações, será a forma mais clara e objetiva para conseguir resultados profissionais para a sua aplicação.

3.3.5 Funções para listas numéricas

Caso você possua uma lista que armazene somente valores numéricos, poderá, então, fazer uso de algumas funções bem úteis. Adicione ao final do código do arquivo ListasDentroDeListas.py, o código abaixo:

```
valores=[]
for elemento in inventario:
    valores.append(elemento[1])
if len(valores)>0:
    print("O equipamento mais caro custa: ", max(valores))
    print("O equipamento mais barato custa: ", min(valores))
    print("O total de equipamentos é de: ", sum(valores))
```

Código-fonte 3.10 – Funções para listas numéricas
Fonte: Elaborado pelo autor (2017)

No código acima, criamos uma lista para armazenar somente os valores dos equipamentos, preenchemos a mesma dentro do “for” e, depois de ser preenchida, verificamos se a lista possui ao menos um valor, e, se a condição for verdadeira, as funções abaixo serão executadas:

- “max()”: que retorna o maior valor numérico dentre os elementos da lista;
- “min()”: que retorna o menor valor numérico dentre os elementos da lista; e
- “sum()”: que retorna o total entre os valores que estão na lista.

Encerramos o nosso assunto de listas por aqui, espero que tenham gostado. Existem mais funções que podem ser utilizadas, mas, para o propósito do curso, este conteúdo sobre listas já é o suficiente. E para o próximo capítulo, observe o seu último arquivo criado… Ficou extenso, não é mesmo? Muitas linhas de comando, o que dificulta uma eventual manutenção e até mesmo a leitura do código… Pois bem, no próximo tópico, iremos aprender a criar e gerenciar as nossas funções e como elas poderão nos ajudar a reduzir o nosso código. print(“Go go… 3.2”)

3.4 Funções

Até agora trabalhamos com diversas funções: max(), min(), sum(), int(), str(), float(), input(), print()… Todas elas já fazem parte da linguagem Python e estão incorporadas na nossa PVM (Python Virtual Machine). Agora vamos imaginar se, por acaso, não existissem as funções. Para facilitar, vamos pensar em uma delas, a input(). Se ela não existisse, a todo momento teríamos que, dentro do nosso código, montar uma sequência de outros códigos para que o usuário pudesse digitar uma informação, isso poderia levar, por exemplo, 15 linhas. Ou seja, em vez de utilizarmos um simples input(), deveríamos utilizar 15 linhas. Agora conte quantos *inputs* utilizou em nosso último arquivo de código gerado e multiplique por 15… Assustador, não é mesmo?!

Com este exemplo, conseguimos perceber quanto as funções permitem reduzir o nosso código, mas o poder das funções vai além… Seguindo ainda a possibilidade de a função input() não existir, você acha que as 15 linhas que deveriam ser digitadas seriam iguais entre mim, você e todos os outros programadores de Python do mundo? Com certeza não, dentro da programação cada um possui sua lógica e defende a sua forma de programar; desde que utilize boas práticas, o que vale é o funcionamento do código, não é mesmo?!

Ou seja, o meu código, necessariamente, não seria igual ao seu. Com isso, perderíamos um conceito muito importante entre os programadores atualmente, o

reaproveitamento. Veja bem, quando você me enviar o seu código para a correção de uma atividade, por exemplo, tenho certeza de que o input() que você utilizou no seu código funcionará perfeitamente no meu Python. Isso porque a PVM está padronizada e, com isso, podemos reaproveitar as funções da PVM em qualquer outro bloco de códigos desenvolvido por qualquer programador Python do mundo.

Estamos percebendo todo o poder das funções, mas eu não quero programar pensando no mundo inteiro, ainda assim as funções seriam úteis? Sim… Vamos pensar na seguinte situação: você foi incubido de desenvolver vários trechos de código, entre eles: gerador de log, leitor de log, automatizador de backup, varredor de portas, atualizador de *switches*, comunicador de sensor de temperatura, entre outros. A pergunta-chave é: O que irá funcionar mais rápido? Você vai fazer tudo sozinho ou vai dividir a tarefa de criar essas funções com outros programadores da empresa? Sem dúvida, a segunda opção, e se você optou por ela, parabéns!!! Com isso, você e o outro programador geraram 500 linhas de código.

Agora temos um problema… Outra filial também precisará das tarefas que você e o outro programador realizaram, **mas** ela precisará apenas de três tarefas, ou seja, parte das 500 linhas programadas. Você terá que separar somente as linhas que a filial irá utilizar, ainda bem que é apenas uma filial… Ufa. **Mas** a sua empresa se fundiu com um concorrente e, com isso, vieram mais 100 filiais, cada uma com suas necessidades específicas, e agora??? Percebeu o trabalho imenso que terá? Como solucionaríamos esse problema? Através das funções!

Crie uma função para cada tarefa, em vez de ter 500 linhas programadas com todas as tarefas, você terá 10 tarefas, cada uma com 50 linhas (todos esses valores são estimados e apenas para exemplificar). E quando as filiais solicitarem para você as rotinas que desejam, é só encaminhar as funções para que elas utilizem o que precisarem exatamente. Prático, não é mesmo?! Denominamos essa distribuição de "**modularização**".

Modularização é o conceito que irá incentivá-lo a identificar, dentro do seu sistema, partes independentes que podem ser construídas de maneira separada (em funções) e, então, reaproveitadas para outras situações, com outros programadores.

Neste momento, você já percebeu que as funções são fundamentais para o gerenciamento do seu código e você irá precisar delas independentemente da vontade de programar algo para todos os programadores Python do mundo ou não.

3.4.1 Identificando as primeiras funções

Abra o nosso arquivo “ListasDentroDeListas.py” e tente identificar visualmente os módulos que esse código poderia possuir. Confira a nossa proposta abaixo, separamos as funções por comentário:

```
inventario=[]
resposta = "S"

#adicionar item no inventario
while resposta == "S":
    equipamento=[input("Equipamento: "),
                 float(input("Valor: ")),
                 int(input("Número Serial: ")),
                 input("Departamento: ")]
    inventario.append(equipamento)
    resposta = input("Digite \"S\" para continuar: ").upper()

#exibir dados do inventário
for elemento in inventario:
    print("Nome.........: ", elemento[0])
    print("Valor........: ", elemento[1])
    print("Serial.......: ", elemento[2])
    print("Departamento.: ", elemento[3])

#localizar um item no inventario
busca=input("\nDigite o nome do equipamento que deseja buscar: ")
for elemento in inventario:
    if busca==elemento[0]:
        print("Valor..: ", elemento[1])
        print("Serial.:", elemento[2])

#depreciar itens no inventario
depreciacao=input("\nDigite o nome do equipamento que será depreciado: ")
for elemento in inventario:
    if depreciacao==elemento[0]:
        print("Valor antigo: ", elemento[1])
        elemento[1] = elemento[1] * 0.9
        print("Novo valor: ", elemento[1])

#excluir um item do inventario
serial=int(input("\nDigite o serial do equipamento que será excluído: "))
for elemento in inventario:
    if elemento[2]==serial:
        inventario.remove(elemento)

#exibir dados do inventário
```

```
for elemento in inventario:
    print("Nome.........: ", elemento[0])
    print("Valor........: ", elemento[1])
    print("Serial.......: ", elemento[2])
    print("Departamento.: ", elemento[3])

#resumo de valores do inventário
valores=[]
for elemento in inventario:
    valores.append(elemento[1])
if len(valores)>0:
    print("O equipamento mais caro custa: ", max(valores))
    print("O equipamento mais barato custa: ", min(valores))
    print("O total de equipamentos é de: ", sum(valores))
```

Código-fonte 3.11 – Identificando funções
Fonte: Elaborado pelo autor (2017)

Perceba que poderemos dividir significativamente o nosso código. Também vale ressaltar que temos dois blocos de códigos idênticos, o bloco “exibir dados do inventário”, ocupando mais linhas que o necessário. Primeiramente, iremos montar aos poucos um novo arquivo chamado: “IdentificacaoDeFuncoes.py”, conforme as funções que separamos.

Antes de iniciarmos, vale ressaltar alguns aspectos:

- Para criarmos funções, utilizaremos o comando “def”, e a estrutura básica de uma função é a seguinte:

  def <identificador da funcao> (<parametro(s)>):

  <código que será executado>

  return <Dado que será retornado, caso seja necessário>

Onde:

- O identificador da função deve seguir as mesmas regras e padronizações dos identificadores das variáveis e listas. As funções representam “ações” como: inserir, exibir, consultar, apagar, calcular, então, é de bom-tom utilizar um verbo no identificador da função.
- O parâmetro é um dado que será fornecido para que a função possa executar o seu bloco de códigos. É como se fossem os ingredientes de uma receita, por exemplo, para que uma função possa calcular uma média aritmética entre duas notas, você deverá fornecer as duas notas ou ainda

para que uma função calcule o salário líquido de um colaborador, precisará informar para a função, no mínimo, o salário bruto.

- O código a ser executado representa o conjunto de códigos que possuem uma mesma finalidade dentro da aplicação.
- A última linha de “return” é opcional e deve ser usada somente quando você desejar que a função retorne um valor para o módulo principal.

Cientes dessas informações, vamos colocar em prática. Criaremos a primeira função, que terá a finalidade de preencher o inventário. Por isso, iremos chamá-la de preencherInventario(). Dentro do nosso novo arquivo “IdentificacaoDeFuncoes.py”, digite o seguinte código:

```
def preencherInventario(lista):
    resp="S"
    while resp == "S":
        equipamento=[input("Equipamento: "),
                     float(input("Valor: ")),
                     int(input("Número Serial: ")),
                     input("Departamento: ")]
        lista.append(equipamento)
        resp = input("Digite \"S\" para continuar: ").upper()
```

Código-fonte 3.12 – Função preencherInventario()
Fonte: Elaborado pelo autor (2017)

Para a nossa função preencherInventario(), recebemos um parâmetro que é a lista na qual o módulo principal irá armazenar os itens do inventário. O nome fornecido ao parâmetro (no nosso caso, “lista”) não tem qualquer relação com a lista que será criada no módulo principal.

O “módulo principal” a que estamos nos referindo é o local onde as nossas funções serão chamadas, pois a função sozinha não executa absolutamente nada. Tente executar o seu arquivo “IdentificacaoDeFuncoes.py” e verá que não ocorre absolutamente nada até o momento. Somente quando chamarmos nossa função no “módulo principal” é que a mesma entrará em ação.

Durante a montagem das funções, muito cuidado com as tabulações. Tente você, agora, montar o código da nossa segunda função: exibirInventario(). Monte o código dessa função imediatamente abaixo da função anteriormente criada (preencherInventario()). Seu código deverá ficar semelhante ao código abaixo:

```
def preencherInventario(lista):
    resp="S"
```

```
    while resp == "S":
        equipamento=[input("Equipamento: "),
                     float(input("Valor: ")),
                     int(input("Número Serial: ")),
                     input("Departamento: ")]
        lista.append(equipamento)
        resp = input("Digite \"S\" para continuar: ").upper()

def exibirInventario(lista):
    for elemento in lista:
        print("Nome.........: ", elemento[0])
        print("Valor........: ", elemento[1])
        print("Serial.......: ", elemento[2])
        print("Departamento.: ", elemento[3])
```

Código-fonte 3.13 – Função preencherInventario() e exibirInventario()
Fonte: Elaborado pelo autor (2017)

A nossa função “exibirInventario()”irá receber a lista, por parâmetro, e, então, executará o laço “for” para exibir os dados da lista recebida. Nessa função, não precisamos de *return*, uma vez que as informações já estão sendo “printadas”. Agora, cabe a você tentar montar as outras funções conforme as marcações que fizemos no início deste tópico.

Tente criar as funções: localizarPorNome(), depreciarPorNome(), excluirPorSerial() e resumirValores(). Abaixo, o código da função localizarPorNome():

```
def localizarPorNome(lista):
    busca=input("\nDigite o nome do equipamento que deseja buscar: ")
    for elemento in lista:
        if busca==elemento[0]:
            print("Valor..: ", elemento[1])
            print("Serial.:", elemento[2])
```

Código-fonte 3.14 – Função localizarPorNome()
Fonte: Elaborado pelo autor (2017)

A função acima não apresentou nenhuma novidade: recebe a lista, solicita o nome a ser pesquisado, localiza o nome do produto na lista e, se for encontrado, irá exibir os dados do elemento localizado.

```
def depreciarPorNome(lista, porc):
    depreciacao=input("\nDigite o nome do equipamento que será depreciado: ")
    for elemento in lista:
        if depreciacao==elemento[0]:
            print("Valor antigo: ", elemento[1])
            elemento[1] = elemento[1] * (1-porc/100)
            print("Novo valor: ", elemento[1])
```

Código-fonte 3.15 – Função depreciarPorNome()
Fonte: Elaborado pelo autor (2017)

A função depreciarPorNome() apresenta dois parâmetros: um deles é a lista na qual estão os equipamentos que sofrerão a depreciação; e o outro é a porcentagem que se deseja depreciar. Repare que a fórmula matemática foi alterada quando comparada com o arquivo anterior.

```
def excluirPorSerial(lista):
    serial=int(input("\nDigite o serial do equipamento que será excluido: "))
    for elemento in lista:
        if elemento[2]==serial:
            lista.remove(elemento)
    return "Itens excluídos."
```

Código-fonte 3.16 – Função excluirPorSerial()
Fonte: Elaborado pelo autor (2017)

A função excluirPorSerial() retorna uma *string*, ou seja, quando formos chamar essa função, devemos fazê-la dentro de um comando print(), para que possamos ver a mensagem.

```
def resumirValores(lista):
    valores=[]
    for elemento in lista:
        valores.append(elemento[1])
    if len(valores)>0:
        print("O equipamento mais caro custa: ", max(valores))
        print("O equipamento mais barato custa: ", min(valores))
        print("O total de equipamentos é de: ", sum(valores))
```

Código-fonte 3.17 – Função resumirValores()
Fonte: Elaborado pelo autor (2017)

E para finalizar, a função resumirValores() poderia ser facilmente dividida em três funções: somar(), exibirMaiorValor() e exibirMenorValor(), tudo depende muito do contexto e da especificação que se deseja obter. Para o nosso exemplo, manteremos apenas como uma função que realiza as três tarefas.

Pronto??? Ainda não, agora que já temos as nossas funções, podemos montar o nosso módulo principal a fim de chamá-las e testá-las para checarmos se está tudo correto. Para isso, vamos seguir as seguintes etapas:

- Crie um “Python Package”, cuja função é apontar para o PyCharm um pacote em que teremos arquivos que poderão ser importados por outros projetos. Para criá-lo, clique com o botão direito sobre MeusProjetos_Python, selecione a opção New/Python Package e atribua o nome Capitulo3_Funcoes. Conforme é demonstrado na imagem abaixo:

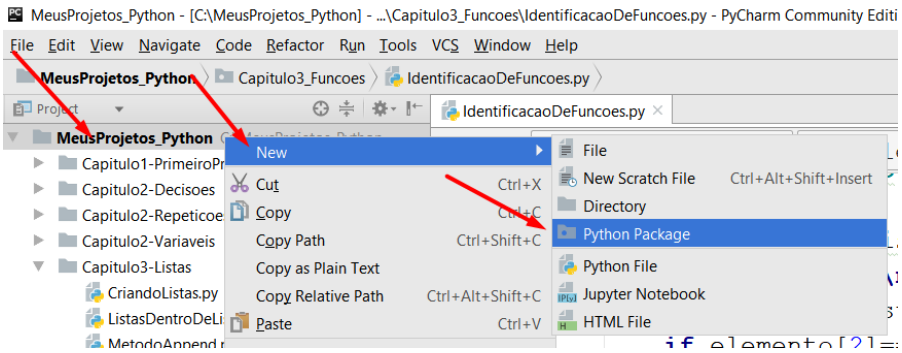

Figura 3.5 – PyCharm – adicionando um Python Package
Fonte: Elaborado pelo autor (2017)

- Agora você deverá arrastar o arquivo “IdentificacaoDeFuncoes.py” para dentro do Python Package criado. Sua tela do PyCharm ficará assim:

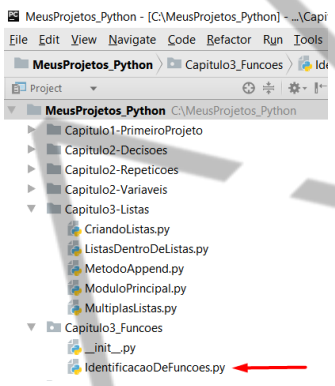

Figura 3.6 – PyCharm – definindo o módulo de funções
Fonte: Elaborado pelo autor (2017)

- Repare que, dentro do pacote gerado, foi criado um arquivo chamado “__init__.py”. Não discutiremos sobre esse arquivo agora e ele também não nos será útil. Por isso, você pode excluí-lo. Agora, vamos criar um novo arquivo, dentro do diretório Capitulo3-Listas, chamado “ModuloPrincipal.py”. Dentro dele, devemos utilizar dois novos comandos, que normalmente são utilizados em conjunto, identificados como “from” e “import”. O “from” deverá receber o local físico no qual se encontram as funções que se deseja importar. Já no “import”, você deverá definir qual ou quais funções deseja importar. Se utilizar um asterisco “*”, ele importará todas as funções contidas no “from”, conforme podemos perceber na primeira linha do código do nosso módulo principal abaixo:

```
from Capitulo3_Funcoes.IdentificacaoDeFuncoes import *

minhaLista=[]
print("Preenchendo")
preencherInventario(minhaLista)
print("Exibindo")
exibirInventario(minhaLista)

print("Pesquisando")
localizarPorNome(minhaLista)
print("Alterando")
depreciarPorNome(minhaLista, 20)

print("Excluindo")
print(excluirPorSerial(minhaLista))
exibirInventario(minhaLista)

print("Resumindo")
resumirValores(minhaLista)
```

Código-fonte 3.18 – Módulo principal
Fonte: Elaborado pelo autor (2017)

Ficou muito mais *clean* a leitura do módulo principal, não é mesmo? Repare que somente a função excluirSerial() foi chamada dentro do print(), isso porque somente ela possui retorno. Veja também a economia de linhas quando apenas chamamos duas vezes a função exibirInventario(), em vez de repetir todas as linhas que estão dentro dessa função. E vamos pensar na hipótese das centenas de filiais que podem aparecer, cada uma com sua necessidade. Não será problema algum agora, pois passaremos o nosso módulo e a filial irá reaproveitar as funções que desejar, assim, não teremos que ficar separando linhas.

Poderíamos ainda sugerir mudanças ou inovar nas nossas funções, como, por exemplo, pedir ao módulo principal que já informe o nome do equipamento e que esse dado não seja solicitado pela função localizarPorNome(). Mas essas são decisões que dependem muito do contexto em que estiverem inseridas, o que você precisa, neste momento, é, toda vez que estiver diante de um código muito extenso, pensar se há a possibilidade de o separar em funções e assim criar módulos que irão facilitar a manutenção do código e, principalmente, o seu reaproveitamento.

Parabéns por chegar até aqui, o seu código começou a ficar com um aspecto altamente profissional.

Agora você está pronto para encarar os dicionários… São estruturas para dados voláteis, assim como as listas e as variáveis, mas com muito mais recursos. Apenas para aguçar a sua curiosidade, vamos para uma característica dos

dicionários. Percebeu que, para qualquer busca em lista, devemos utilizar um “for” ou “while”, ou seja, devemos percorrer toda a lista? Nos dicionários, podemos encontrar os dados de maneira direta, agilizando, assim, o processamento e economizando mais linhas de código. Então, é isso. Prontos para o próximo capítulo? print (“Common JSON”).

REFERÊNCIAS

ELMASRI, Ramez; NAVATHE, Shamkant, B. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson Addison-Wesley, 2011.

FORBELLONE, Andre Luiz Villar. **Lógica de programação**. 3. ed. São Paulo: Prentice Hall, 2005.

JET BRAINS. **PyCharm Community, version 2017.2.4:** IDE para Programação. Disponível em: <https://www.jetbrains.com/pycharm/download/#section=windows>. Acesso em: nov. 2017.

KUROSE, James F. **Redes de computadores e a Internet:** uma abordagem top-down. 6. ed. São Paulo: Pearson Education do Brasil, 2013.

MAXIMIANO, Antonio Cesar Amaru. **Empreendedorismo**. São Paulo: Pearson Prentice Hall Brasil, 2012.

PIVA, Dilermando Jr. **Algoritmos e programação de computadores**. São Paulo: Elsevier, 2012.

PUGA, Sandra. **Lógica de programação e estruturas de dados**. São Paulo: Prentice Hall, 2003.

RHODES, Brandon. **Programação de redes com Python**. São Paulo: Novatec, 2015.

STALLINGS, W. **Arquitetura e organização de computadores**. 8. ed. São Paulo: Prentice Hall Brasil, 2010.