PYTHON

MANIPULAÇÃO DE DICIONÁRIOS DE **DADOS E TUPLAS**

HUMBERTO D. DE SOUSA

4

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

4 MANIPULAÇÃO DE DICIONÁRIOS DE DADOS E TUPLAS

Neste capítulo, apresentamos uma das estruturas mais flexíveis do Python, o dicionário de dados, que, assim como as listas foram exploradas no capítulo anterior, deve ser utilizado com dados voláteis em memória. Vamos ver as diferenças entre um dicionário e uma lista, e quais as situações mais recomendadas para utilização dos dicionários. E, por fim, comentamos sobre as tuplas, a última estrutura de dados que veremos no curso, para dados voláteis.

Vamos em frente!

4.1 Estrutura de um dicionário de dados

Antes de apresentarmos a estrutura de um dicionário de dados, vamos aproveitar para pensar em uma situação em que as listas já não podem ser a melhor das opções.

4.1.1 Listas? Aqui não!

De acordo com o seu projeto para a clínica, vamos pensar na seguinte situação: você, com certeza, já pensou em grupos de usuários e os seus níveis de permissionamento. Imagine que você vai gerar uma fonte, representada por um arquivo que será controlado por sua aplicação, e nele encontrará informações sobre as ações dos usuários, entretanto, realizar essa leitura diretamente no arquivo não é nada prático e tampouco rápido.

Eis que surge a ideia de desenvolver uma ferramenta capaz de realizar, além da inserção dos dados, a leitura dos dados e retorná-los de tal forma que você obtenha as informações necessárias e resumidas e possa tomar as suas decisões de gerenciamento e segurança do ambiente, de maneira rápida, com eficiência e eficácia. Vamos trabalhar, então, com base no perfil do usuário que será avaliado, veja na imagem abaixo:

Figura 4.1 – Perfil do usuário
Fonte: Google Imagens, adaptado pelo autor (2017)

Vamos iniciar considerando apenas os dados encontrados na figura acima, ou seja, todo usuário (colaborador da clínica) terá registrado um *login* (que é único, porque não podemos ter dois usuários com o mesmo *login*), nome, a data do último acesso (futuramente poderemos pensar na hora também, e a estação que ele logou pela última vez). Alguns outros dados parecem ser importantes a fim de complementarmos o registro do usuário como: cargo, departamento, nível de acesso (usuário, administrador, gerente etc.), entre outros, mas, no momento, vamos adotar apenas estes para fins didáticos. Esses dados serão enviados para o arquivo de log (que você irá ter acesso) e serão atualizados todas as vezes em que o usuário logar em uma estação dentro da clínica.

Você precisa, a princípio, pensar em uma estrutura que receba os dados dos usuários para que você possa examiná-los, a fim de identificar alguma “anomalia”, por exemplo, um usuário que aparece acessando uma estação que está em um departamento que ele não atua ou um último acesso localizado em uma data em que o departamento dele não abriu oficialmente. Você pensaria em criar uma estrutura através de lista?

Pois é, essa não seria a situação mais indicada para as listas. Por quê? Vamos ver um dos motivos: se pensarmos em uma lista para cada usuário, quantas listas exatamente precisaremos criar no código? Não conseguiremos definir, pois o número de colaboradores não é fixo e podemos ter mais ou menos do que planejado, por falta, férias, folga, novas contratações, entre outras possibilidades.

ATENÇÃO: nada de pensar em colocar um número máximo de listas achando uma quantidade de colaboradores inatingível pela clínica, ok?
Pense em Escalabilidade, ou seja, seu projeto deve funcionar nesta clínica, como também deverá funcionar em uma megaclínica com milhares de filiais espalhadas pelo mundo inteiro.

Então, você pensa em criar uma lista para colaboradores e, dentro dela, outras listas que irão armazenar os dados efetivamente de cada colaborador. Mas, pense... cada vez que o usuário logar em uma estação, os dados deverão ser atualizados (data do último acesso e a última estação acessada), ou seja, como em listas você não tem como localizar um dado diretamente, você terá que percorrer (montar o *foreach*) toda a lista todas as vezes que todos os colaboradores realizarem um logon.

Percebeu a quantidade de vezes que tivemos que utilizar da palavra “todas”? Isso é redundância, certo? Então, não é redundância só na Língua Portuguesa, não, é redundância na programação também. Podemos perceber facilmente que algo está errado ou, como dizem informalmente por aí, “algo que não está certo, deve estar errado!”. Logo, podemos constatar que as listas (mesmo que sejam listas dentro de listas) não seriam a estrutura mais adequada para uma situação como esta.

4.1.2 *Dictionary... help please*!!!

Como você já pode imaginar, teremos que utilizar, na situação anteriormente proposta, os dicionários de dados, mas, antes, vamos ver como eles são formados e como funcionam.

Um dicionário armazena elementos. De agora em diante, vamos chamar estes elementos de objetos. Cada objeto será dividido em duas partes: chave e

dados. A chave é um dado único e exclusivo do elemento. Todo elemento deverá ter uma chave. Através dela, vamos identificar quem é o objeto e, consequentemente, poderemos visualizar os seus dados.

O objeto “usuário” possui alguma chave? Se observar bem, quando descrevemos a situação proposta, explicamos que o “login” é um dado único, ou seja, ele poderá ser a chave do objeto “usuário”, uma vez que nunca teremos dois usuários com o mesmo login. Podemos também dizer que nunca teremos dois objetos com a mesma chave login em nosso dicionário. Podemos dar outros exemplos para que você entenda o valor de uma chave em um dicionário.

Para a Receita Federal, cada pessoa física é um objeto que possui vários dados: nome, endereço, CPF, RG, estado civil, telefone, entre outros. Qual entre estes dados representa um único objeto, ou seja, uma única pessoa física para a Receita Federal? Isso mesmo, o CPF. Ele é único para cada pessoa física e com isso, poderia ser a chave. Outro exemplo: um ativo de rede, dentro da clínica, deverá fazer parte de um inventário e, por isso, será necessário guardar dados sobre ele, como: descrição, modelo, fabricante, valor de compra, data da aquisição, entre outros.

Uma pergunta: Qual, entre estes apresentados poderia ser a chave do ativo de rede? Acertou novamente, nenhum. Entre os dados apresentados, nenhum pode garantir a unicidade de um objeto dentro do dicionário de dados. Em situações como essa, você deve criar um dado para esse papel. Poderia ser um código, um número para o inventário ou número serial. Estes seriam os mais comuns, mas você poderia definir outro nome qualquer para este dado. O importante é que todo objeto de um dicionário deve possuir um dado como chave, e será através dele que obteremos os dados do objeto.

Agora podemos ver como representaremos os dicionários dentro do Python. Abra o nosso projeto e, consequentemente, crie um Python Package, conforme está representado na figura abaixo:

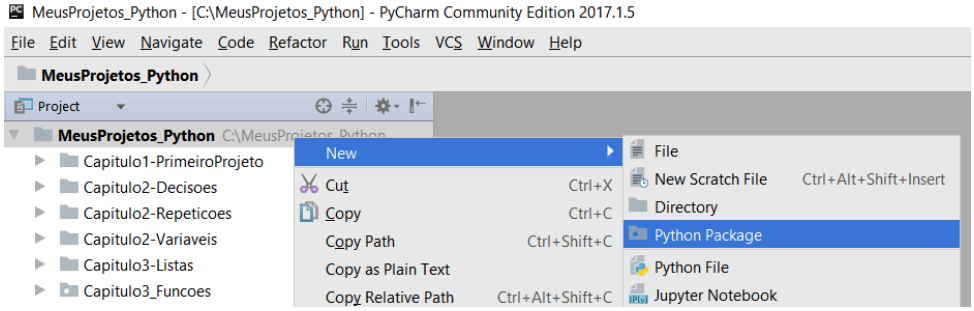

Figura 4.2 – Criando um Python Package
Fonte: Elaborado pelo autor (2017)

Atribua para este pacote o nome: “Capitulo4_Dicionarios”, sua tela deverá estar como a imagem abaixo, e crie um Python File chamado “Dicionarios1.py” dentro do pacote.

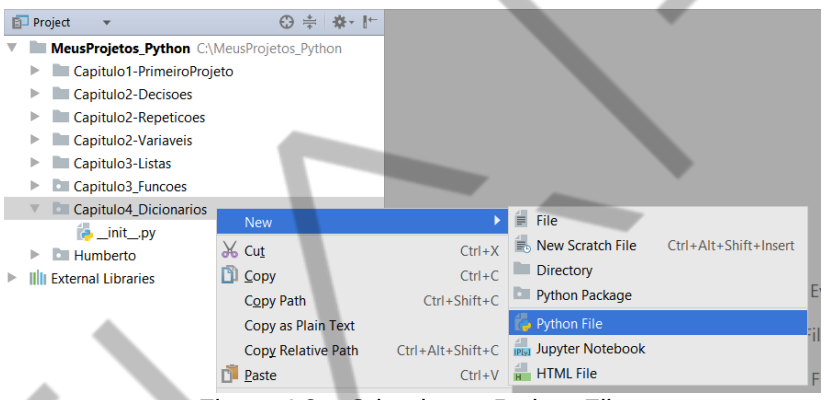

Figura 4.3 – Criando um Python File
Fonte: Elaborado pelo autor (2017)

Dentro do arquivo criado, monte o código abaixo:

```
usuarios={}
usuarios={
    "Chaves":["Chaves Silva","17/06/2017","Recep_01"],
    "Quico":["Enrico Flores","03/06/2017","Raiox_02"]
    }
```

Código-fonte 4.1 – Exemplo de um dicionário de dados
Fonte: Elaborado pelo autor (2017)

Vamos analisar o código. Na primeira linha, estamos criando o dicionário de dados. Repare que, em vez de utilizarmos os colchetes (como fizemos com as listas), usamos as chaves “{}”, essa é a representação de um dicionário de dados.

Na segunda linha, preenchemos fielmente o nosso dicionário de acordo com o nosso exemplo de um dicionário de dados. Acima, perceba que adotamos os logins como chave de cada objeto (no caso, são dois). Depois da chave, utilizamos dois pontos (:) e, então, surgem os dados da chave que foi preenchida. Como no

exemplo, o login “Chaves” possui mais de um dado, optamos em criar uma lista para armazenar estes dados e, dentro desta lista, colocamos o nome, a data de último acesso e o nome da última estação onde esteve logado. Após a vírgula (,) iniciamos o outro objeto, com a chave “Quico”, seguido de dois pontos (:) e, então, a lista com os dados deste objeto. E aí encerramos o dicionário, fechando as chaves. Se, por um engano, você colocar dois objetos com a mesma chave, o segundo objeto irá sobrescrever o primeiro. Também podemos adicionar itens no dicionário da seguinte forma:

```
usuarios["Florinda"]=["Florinda Flores", "26/11/2017", "Recep_01"]
```

Código-fonte 4.2 – Adicionando um objeto no dicionário
Fonte: Elaborado pelo autor (2017)

A linha acima apresentada normalmente é utilizada quando queremos adicionar os objetos de maneira singular, ou seja, objeto por objeto. Repare que, após o nome do dicionário, colocamos entre colchetes do dado que será armazenado na chave e igualamos com o dado que pertence à chave. No nosso caso, uma lista com nome, data do último acesso ao sistema e a última estação que foi utilizada pelo usuário. Da primeira forma que foi apresentada (no Código-fonte: “Exemplo de um dicionário de dados”), se tentarmos dividir em dois momentos a adição dos objetos no dicionário, simplesmente estaremos considerando a última linha onde foi inserido o conteúdo, sobrescrevendo todos os objetos inseridos anteriormente.

Caso repita a mesma linha do Código-fonte: “Adicionando um objeto no dicionário”, isto é, se tentar usar a **mesma chave**, o código não irá gerar erros ou exceções, mas o objeto que estiver na segunda linha irá sobrescrever o objeto que estava na primeira linha. Segue abaixo um código para exemplificar:

```
usuarios={}
usuarios={
    "Chaves":["Chaves Silva","17/06/1975","Recep_01"],
    "Quico":["Enrico Flores","03/06/1976","Raiox_02"],
    "Quico":["Enrico Flores","03/06/1976","Raiox_03"]
    }
usuarios["Florinda"]=["Florinda Flores", "26/11/2017", "Recep_01"]
usuarios["Florinda"]=["Florinda Flores", "26/11/2016", "Recep_01"]
```

Código-fonte 4.3 – Adicionando dados com chaves duplicadas
Fonte: Elaborado pelo autor (2017)

Com base no código acima, ao executar o Python irá armazenar dentro do dicionário “usuarios” apenas três objetos, um com a chave “Chaves”, outro com a

chave “Quico” (que teve como último acesso a estação “Raiox_03”) e o último com a chave “Florinda” (com a data “26/11/2016” atribuída ao último acesso).

Veremos agora como podemos retornar os dados de um objeto da lista, acrescente no seu código as linhas abaixo:

```
print(usuarios)
print("###############========#########")
print("Dados: ",usuarios.get("Chaves"))
```

Código-fonte 4.4 – Exibindo dados de um dicionário de dados
Fonte: Elaborado pelo autor (2017)

No primeiro print(), exibimos tudo o que existe dentro do dicionário. Já no último print(), somente os dados do objeto que tiver a chave “Chaves”. Isso foi possível porque invocamos o nosso dicionário “usuarios” e utilizamos o seu método get(), que recebe um dado e vai pesquisá-lo entre as chaves que existem dentro do dicionário, caso ele encontre, retornará os dados relativos à chave encontrada.

Se você alterar, dentro do get(), o valor “Chaves” para “Chapolim”, verá que será retornado o valor “None”, palavra reservada que representa um valor não encontrado dentro do Python. Percebeu que não precisamos criar um *foreach* para localizar uma informação dentro do dicionário? Por isso, em determinadas situações, é muito mais prático e eficiente utilizar dicionários, assim como você pode ter outras situações em que as listas serão o caminho mais viável, ou ainda, conforme demonstramos neste exemplo, você pode ter uma lista dentro de um dicionário. Cabe a você julgar a melhor estrutura que atenderá à sua necessidade.

4.1.3 Começando nossa ferramenta

Podemos agora resolver a questão da nossa ferramenta que vai armazenar os dados dos usuários dentro dos dicionários e realizar as devidas alterações e possíveis exclusões. Vamos começar criando um arquivo, chamado: “ManagerUsers.py” e iniciar perguntando ao usuário o que ele deseja realizar: inserir, pesquisar, excluir, listar ou sair. Mão na massa. Agora, tente seguir desenvolvendo e olhe o código a seguir, quando tiver dúvidas, mas é fundamental que tente fazer o código, pelo menos parcialmente. Abaixo, segue o código para os nossos comentários subsequentes:

```
usuarios={}
opcao=input("O que deseja realizar?\n" +
            "<I> - Para Inserir um usuário\n"+
            "<P> - Para Pesquisar um usuário\n"+
            "<E> - Para Excluir um usuário\n"+
            "<L> - Para Listar um usuário: ").upper()
while opcao=="I" or opcao=="P" or opcao=="E" or opcao=="L":
    if opcao=="I":
        chave=input("Digite o login: ").upper()
        nome=input("Digite o nome: ").upper()
        data=input("Digite a última data de acesso: ")
        estacao=input("Qual a última estação acessada: ").upper()
        usuarios[chave]=[nome, data, estacao]
    opcao = input("O que deseja realizar?\n" +
                  "<I> - Para Inserir um usuário\n" +
                  "<P> - Para Pesquisar um usuário\n" +
                  "<E> - Para Excluir um usuário\n" +
                  "<L> - Para Listar um usuário: ").upper()
```

Código-fonte 4.5 – Nossa ferramenta – Inserir
Fonte: Elaborado pelo autor (2017)

O código acima apresenta alguns pontos a serem melhorados. Olhe-o com criticidade e identifique algumas falhas. Antes de apontá-las, vamos descrever o que está ocorrendo: criamos um dicionário chamado “usuarios” e uma variável chamada “opcao”, que vai armazenar, em caixa alta, o que o usuário deseja fazer. Para as letras “I”, “P”, “E” ou “L”, o código vai realizar uma ação, para qualquer outro valor, o código será encerrado.

Por isso, o comando “while” na sequência, ou seja, enquanto houver uma opção válida o código continuará executando. Se a variável “opcao” armazenar o valor “I”, vamos inserir um objeto no dicionário, para isso, criamos quatro variáveis, uma para chave, nome, data de último acesso e última estação acessada, respectivamente. Após o preenchimento das variáveis, vamos utilizá-las no dicionário de dados e, assim, sucessivamente, enquanto o conteúdo da variável “opcao” for igual a “I”. Conseguiu perceber alguns pontos fracos deste código? Primeiro, em relação à criação das variáveis, poderíamos substituir as cinco linhas a seguir:

```
chave=input("Digite o login: ").upper()
nome=input("Digite o nome: ").upper()
data=input("Digite a última data de acesso: ")
estacao=input("Qual a última estação acessada: ").upper()
usuarios[chave]=[nome, data, estacao]
```

Código-fonte 4.6 – Criação das variáveis
Fonte: Elaborado pelo autor (2017)

Pelas duas linhas a seguir:

```
chave=input("Digite o login: ").upper()
usuarios[chave]=[input("Digite o nome: ").upper(),
                 input("Digite a última data de acesso: "),
                 input("Qual a última estação acessada: ").upper()]
```

Código-fonte 4.7 – Reduzindo o código
Fonte: Elaborado pelo autor (2017)

Economizamos, assim, três variáveis em memória, e talvez você esteja se perguntando, mas por que não fazer o mesmo com a chave? Assim, não precisaríamos de nenhuma variável, não é mesmo? Sim, você está correto, mas optei em deixar a variável-chave, pois caso eu coloque o *input* dentro dos colchetes da chave, ele será preenchido por último, pois o Python resolverá primeiro o que está depois do sinal de igual, ou seja, ele pediria o nome, a data, a estação e, por último, o login.

Acredito que a ordem ficaria um tanto quanto desconexa, ou seja, uma simples questão de estética para o usuário final. Caso isso não seja um problema para você, pode economizar a variável-chave e colocar tudo em uma única linha, conforme está demonstrado no seguinte código:

```
usuarios[input("Digite o login: ").upper()]=[input("Digite o nome: ").upper(),
                 input("Digite a última data de acesso: "),
                 input("Qual a última estação acessada: ").upper()]
```

Código-fonte 4.8 – Sem o uso de variável
Fonte: Elaborado pelo autor (2017)

Conforme o código: sem o uso de variável, acima, estaríamos substituindo, então, cinco linhas, por apenas, uma linha. Melhor, não é mesmo? Mas caso nesse momento ainda se sinta mais confortável criando as variáveis, não tem problema. Neste ponto, o importante é conseguir desenvolver. Com o passar do tempo e com uma intimidade maior com a linguagem, você começará a desenvolver códigos mais enxutos.

Mas, ainda temos algo “feio” dentro do nosso código, algo bem visível. Encontrou? Vamos lá, temos uma linha enorme sendo utilizada duas vezes, a linha onde você preenche o conteúdo da variável “opção”, assim seu código não está elegante, alguém por aí lembra o que devemos fazer quando repetimos muito um mesmo conjunto de código? Isso. Devemos utilizar “F-U-N-Ç-Õ-E-S”, de cara olhando para o nosso código, podemos dividi-lo em duas ações: preencher a variável “opcao” e preencher o dicionário de dados. Mãos à obra. Crie mais um arquivo Python chamado “Funcoes.py” e, dentro dele, digite o código abaixo:

```
def perguntar():
    resposta = input("O que deseja realizar?\n" +
                     "<I> - Para Inserir um usuário\n" +
                     "<P> - Para Pesquisar um usuário\n" +
                     "<E> - Para Excluir um usuário\n" +
                     "<L> - Para Listar um usuário: ").upper()
    return resposta

def inserir(dicionario):
    dicionario[input("Digite o login: ").upper()] = [input("Digite o nome: ").upper(),
                                                     input("Digite a última data de acesso: "),
                                                     input("Qual a última estação acessada: ").upper()]
```

Código-fonte 4.9 – Criação das Funções – perguntar() e inserir()
Fonte: Elaborado pelo autor (2017)

O código acima representa duas funções. A primeira que perguntará ao usuário e, então, retornará a resposta do usuário; e a segunda que receberá um dicionário (parâmetro que está entre parênteses “()”) e, então, inserirá um objeto dentro deste dicionário. Agora, tente implementar as duas funções no seu arquivo “ManagerUsers.py”. Veja se ficará semelhante ao código abaixo:

```
from Capitulo4_Dicionarios.Funcoes import *
usuarios={}
opcao=perguntar()
while opcao=="I" or opcao=="P" or opcao=="E" or opcao=="L":
    if opcao=="I":
        inserir(usuarios)
    opcao = perguntar()
```

Código-fonte 4.10 – Utilização das Funções – perguntar() e inserir()
Fonte: Elaborado pelo autor (2017)

Veja como o código ficou muito mais limpo, neste que será o nosso módulo principal. Não se esqueça de fazer a importação, que está na primeira linha do código. Agora, nos resta montar as outras funções e implementá-las dentro deste módulo principal. Topa o desafio? Abra o seu arquivo “funcoes.py” e crie as funções: pesquisar() – dica: você precisará receber como parâmetro a chave e o dicionário, excluir() – dica: você precisará utilizar o comando “del”, e listar(). Depois volte para o material e confira o seu código, com a nossa proposta logo abaixo:

```
def pesquisar(dicionario, chave):
    lista=dicionario.get(chave)
    if lista!=None:
        print("Nome...........: " + lista[0])
        print("Último acesso..: " + lista[1])
        print("Última estação.: " + lista[2])

def excluir(dicionario, chave):
    if dicionario.get(chave)!=None:
        del dicionario[chave]
    print("Objeto Eliminado")

def listar(dicionario):
    for chave, valor in dicionario.items():
```

```
        print("Objeto......")
        print("Login: ", chave)
        print("Dados: ", valor)
```

Código-fonte 4.11 – Criação das Funções – pesquisar(), excluir() e listar()
Fonte: Elaborado pelo autor (2017)

O código acima é responsável por criar as outras três funções. Observe alguns detalhes:

- Na função pesquisar(), precisamos receber o dicionário (onde se pretende pesquisar) e a chave (o dado que será pesquisado). Logo após, vamos preencher uma lista com o resultado da pesquisa proveniente do uso da função get(). Verificamos se a lista não está vazia (!= - representa diferente), caso esta condição seja verdadeira, vamos exibir os três dados que compõem a lista. Teremos, na primeira posição (zero), o nome do usuário; na segunda posição (um), a última data de acesso; e na terceira posição (dois), a última estação acessada. Caso não encontre a chave, não será retornada nenhuma mensagem;
- Na função excluir(), também recebemos o dicionário de onde o objeto será excluído e a chave do objeto que se deseja excluir. Efetivamente, antes da exclusão, devemos verificar se a chave existe. Por isso, verificamos com a função get() se será retornado algo diferente de vazio e, se isso for verdade, invocamos o comando “del”, que eliminará o objeto de acordo com a chave que foi recebida.
- Na função listar(), precisaremos apenas do dicionário que contém os dados que desejamos exibir, com isso, montamos um foreach, mas perceba que, desta vez, utilizamos dois valores (chave e valor) para que possamos dar uma saída um pouco mais “clean” para o nosso usuário final. Poderíamos fazer de outras formas, inclusive, utilizando a nossa outra função pesquisar().

Agora nos resta implementar as nossas funções dentro do módulo principal, ou seja, dentro do arquivo “ManagerUsers.py”, conforme apresentado abaixo (vale tentar implementar antes de verificar o código abaixo):

```
from Capitulo4_Dicionarios.Funcoes import *
usuarios={}

opcao=perguntar()
```

```
while opcao=="I" or opcao=="P" or opcao=="E" or opcao=="L":
    if opcao=="I":
        inserir(usuarios)
    if opcao=="P":
        pesquisar(usuarios,input("Qual login deseja pesquisar? "))
    if opcao == "E":
        excluir(usuarios,input("Qual login deseja excluir? "))
    if opcao == "L":
        listar(usuarios)
    opcao = perguntar()
```

Código-fonte 4.12 – Utilização das Funções – pesquisar(), excluir() e listar()
Fonte: Elaborado pelo autor (2017)

E está aí tudo pronto e funcional, fácil, não é mesmo? Você deve estar se perguntando, por que não realizamos uma função para alterar os dados de um objeto já armazenado na lista? Simples, caso o usuário final digite ao incluir uma mesma chave, todos os dados que ele inserir vão sobrescrever os dados do objeto que já estavam armazenados, logo, não se faz necessário nesse momento uma função para realizar alterações.

Concluímos um exemplo diferente do que foi abordado no capítulo das Listas, mas perceba que as ações sempre estarão em torno de inclusões, consultas, alterações e exclusões. O que vai diferenciar são os recursos que você será capaz de aplicar, o meu conselho é: primeiro procure resolver o problema, depois melhore e evolua o seu código para que ele fique mais reaproveitável, escalável e o mais flexível possível.

A seguir, alguns métodos adicionais que podem ser úteis na manipulação dos dicionários.

4.1.4 Métodos adicionais para dicionários

Os dicionários possuem alguns métodos que podem ser úteis em outras situações do dia a dia, vamos explorar alguns deles:

- **items()**: responsável por retornar em forma de lista os elementos do dicionário. Para isso, ele fará uso do recurso de tuplas, que será apresentado no tópico seguinte. Basicamente, cada tupla terá uma chave e um dado, conforme podemos perceber no exemplo abaixo. Considerando nosso exemplo, poderíamos ter este método retornando os dados da seguinte forma:

[(“ologin”,[“nome”,”data”,”estacao”]),(“login2”,[“nome2”, “data2”,”est2”])]

Perceba que temos a lista e, dentro da lista, dois elementos entre parênteses (cada elemento é uma tupla) e, dentro de cada elemento, dois elementos, a chave (login) e o dado (lista com nome, data e estação). Este método é muito utilizado em *foreachs*, como podemos perceber no método listar() que criamos anteriormente.

- **values()**: podemos retornar também somente os dados, descartando as chaves, ou seja, esse método retorna uma lista formada apenas pelos dados. Se pegarmos como exemplo o nosso caso do histórico dos usuários, teremos uma lista dentro de outra lista, e o exemplo ficaria como o que está apresentado abaixo:

  [[“nome”,”data”,”estacao”],[“nome2”, “data2”,”est2”]]

- **keys()**: claro que se retornarmos somente os dados, podemos retornar também somente as chaves. E é isto o que este método faz, retornando todas as chaves do dicionário em forma de lista, conforme apresentado na linha abaixo:

  [“ologin”, “login2”]

- **has_key()**: este método permitirá que você tenha a resposta se a chave existe ou não dentro do dicionário, ele irá retornar True (1) ou False (0).

- **clear()**: esvazia completamente o dicionário.

- **popitem()**: este é um método próprio para quem deseja montar algum dicionário que contenha elementos que serão executados, de maneira aleatória, individualmente e, na sequência, deverão ser eliminados do dicionário. Poderíamos pensar em um dicionário com dicas, e à medida que cada dica fosse exibida, automaticamente ela deveria ser retirada do dicionário.

Enfim, estes são alguns métodos que, em eventuais e particulares circunstâncias, podem ser necessários e, então, utilizados. Estes métodos também podem ser apontados como diferenciais quando comparamos listas com dicionários.

Para praticar um pouco, procure aperfeiçoar o nosso exemplo ManagerUsers.py, inserindo mais dados que podem ser importantes para o seu

controle. Você acredita que o “nome de login” realmente seria um dado único para a situação de controle com base nos logs que seriam gerados, nas estações, a cada login?

Imagine, por exemplo, a seguinte situação: você detectou uma operação suspeita em um “login”, chamou o colaborador responsável pelo login e o questionou sobre as atividades suspeitas. O colaborador disse que passou o dia de trabalho logado em outra estação. Da forma como está o nosso código, o que o colaborador disse pode ser verdade? Sim. E por que você não conseguiu visualizar os dois logins em estações diferentes? Você não conseguiu porque o login está como chave, portanto, quando o login, da estação suspeita, foi realizado, ele sobrescreveu o login da estação onde o colaborador disse que estava trabalhando. Dessa forma, sabemos que existe uma falha, mas ficou difícil apontar para o verdadeiro culpado ou ainda de levantar mais suspeitas para que pudéssemos verificar com maior segurança.

Para resolver isso, o correto seria que cada login fosse armazenado no log, ou seja, se o colaborador realizar dez logins por dia, esses dez logins deverão constar no arquivo de log. Para isso, não podemos deixar que o login seja considerado a chave, também será imprescindível armazenar a hora do login. Assim, teremos mais dados que poderão nos completar as informações necessárias para uma tomada de decisão. Pode ser que houve falha no treinamento do colaborador, falha nas políticas de segurança do sistema operacional, falha de segurança na aplicação da empresa ou ainda um colaborador mal-intencionado. Percebe como são importantes os dados?

Deste modo, fica aqui o desafio: aprimore o nosso “ManagerUsers.py” e o ‘Funcoes.py”, acrescente dados novos, como hora do login, código do lançamento e nível do usuário, por exemplo. Remova o nome do login da chave e repasse-o para os dados. Para a chave, use o código do lançamento ou um outro dado, caso queira. Mas lembre-se de que a chave deve ser um dado único, certo?

Espero que esteja curtindo até o momento e procurando observar como o Python pode ser útil para o desenvolvimento de ferramentas pequenas, simples e muito úteis que podem auxiliar no processo de uma tomada de decisão. Tenho certeza que irá se surpreender ainda mais. Na sequência, falaremos um pouco

sobre as tuplas, a última estrutura para dados voláteis que apresentaremos neste capítulo.

Pronto para mais esta missão? Seguimos em frente... e que venham as tuplas!!!

4.2 Tuplas

As tuplas, como já dissemos, são estruturas também para dados voláteis, assim como as variáveis, listas e dicionários. Porém, você percebeu que cada uma possui as suas devidas aplicações, e com as tuplas não poderia deixar de ser diferente.

Uma característica única das tuplas é o fato de elas não aceitarem alteração sobre os dados que já estiverem nelas inseridos. Além disso, as tuplas sempre são representadas com seus dados entre parênteses. Então, podemos sintetizar que: listas envolvem os dados entre colchetes; dicionários entre chaves; e tuplas entre parênteses.

Normalmente, aplicamos mais o conceito de tuplas para realizar a leitura de uma resposta do Python, e não de alguém que inseriu dados. É o exemplo do método “items()” que vimos dos dicionários, ele retorna os dados do dicionário, identificando cada elemento como uma tupla, naturalmente aplicado para exibição, e foi retornado pelo Python, uma vez que o dado foi preenchido através do dicionário.

Por isso, podemos afirmar que as tuplas não estão entre as estruturas mais selecionadas pelos programadores Python, mas devemos saber como trabalham, pois, como explicado, o Python responde muito através dessa estrutura. Ela também tem uma possibilidade que as listas não possuem, que é o fato de poderem ser utilizadas como chave de um dicionário.

No nosso exemplo anterior, utilizamos os dados do usuário na posição de dados do dicionário, e como chave utilizamos uma string, no caso, o nome do login. Se, por exemplo, quiséssemos atribuir como chave o e-mail de uma pessoa, e este e-mail estivesse dividido entre dois dados (usuário e servidor), não poderíamos colocá-lo na chave de um dicionário como lista, mas, sim, como tupla.

Outro exemplo: você poderia pensar em definir um endereço IP como chave de um dicionário, mas seria importante manter os dados divididos em octetos, a fim de identificar grupos que pertencem a uma mesma rede (normalmente os dois primeiros octetos) ou usuários que logaram em uma mesma máquina (normalmente os dois últimos octetos identificam a estação).

Para isso, você será obrigado a utilizar a estrutura de tuplas, que poderá armazenar os octetos separadamente e também ser implementada pelo dicionário na posição de chave. Existe uma explicação técnica para que isso seja possível com tuplas e não com listas, mas não vamos abordar este assunto neste curso. Esse tipo de informação seria útil para um desenvolvedor de sistemas, mas se sentiu curiosidade, não deixe de pesquisar.

Veremos a seguir a manipulação básica dos dados que podem ser inseridos e recuperados em uma tupla. Vamos iniciar criando um arquivo chamado: “Tupla.py” (no próprio Python Package do Capítulo 4), e, então, digitaremos o código abaixo:

```
ips={}
resp="S"
while resp=="S":
    ips[(input("Digite os dois primeiros octetos: "),
       input("Digite os dois últimos octetos: "))]=input("Nome da máquina: ")
    resp=input("Digite <S> para continuar: ").upper()
```

Código-fonte 4.13 – Utilização das tuplas
Fonte: Elaborado pelo autor (2017)

Como podemos observar no código, criamos um dicionário chamado “ips”, uma variável chamada “resp” para controlar o nosso laço e, dentro do laço de repetição, vamos preencher o nosso dicionário, mas perceba que, na chave do dicionário, inserimos dois valores. A primeira parte do ip e a segunda parte do ip, dentro de uma tupla, e o dado do elemento que está sendo adicionado no dicionário é o nome da estação.

Vejamos agora como aproveitar os dados da chave no formato de tupla. Adicione o código abaixo:

```
print("Exibindo ip´s: ")
for ip in ips.keys():
    print(ip[0]+"."+ip[1])
```

Código-fonte 4.14 – Exibição dos dados da tupla
Fonte: Elaborado pelo autor (2017)

A saída dos dados, após a execução do código, apresentou os ips concatenados com um ponto ".", conforme a função print() que está dentro do laço "for". Veja como unimos as duas partes da chave do dicionário na mesma linha da função print(), utilizando a posição dos dados, ou seja, recuperando a posição zero (ip[0]) e a posição um (ip[1]). Veja também que o nosso laço for está recuperando apenas as chaves do nosso dicionário, que é onde se encontra a nossa tupla.

No código abaixo, vamos pedir ao usuário a segunda parte do ip (dois últimos octetos), isto é, o endereço da máquina, para, então, exibirmos o nome das máquinas que estão sob o mesmo endereço, mas em redes diferentes (baseado nos dois primeiros octetos). Perceba que está se tornará uma tarefa simples, pois a nossa chave já está desmembrada, facilitando muito a recuperação do dado. Adicione agora o código abaixo:

```
print("Exibindo máquinas com o mesmo endereço: ")
pesquisa=input("Digite os dois últimos octetos: ")
for ip,nome in ips.items():
    print("Máquinas no mesmo endereço (redes diferentes)")
    if(ip[1]==pesquisa):
        print(nome)
```

Código-fonte 4.15 – Pesquisa na chave de um dicionário em forma de tupla
Fonte: Elaborado pelo autor (2017)

Avaliando o código acima, vemos a solicitação do dado para o usuário e o armazenamento deste dado na variável pesquisa. Em seguida, na linha onde definimos o nosso laço "for", vamos recuperar dois dados ip e o nome da máquina, que serão retornados pelo método items(); Depois, exibimos uma mensagem para o usuário e fazemos a comparação da parte dois do ip (dois últimos octetos) com o que o usuário digitou na variável pesquisa. Caso eles sejam iguais, exibimos o nome da máquina. Agora tente fazer o seguinte: gostaríamos de saber quais estações compõem uma rede. Para isso, iremos nos basear nos dois primeiros octetos do ip. Procure fazer este código e depois compare com o código abaixo:

```
print("Exibindo as máquinas que compõem uma mesma rede: ")
rede=input("Digite os dois primeiros octetos: ")
for ip,nome in ips.items():
    if(ip[0]==rede):
        print(nome)
```

Código-fonte 4.16 – Pesquisa na chave de um dicionário em forma de tupla – 2
Fonte: Elaborado pelo autor (2017)

Desta vez, de acordo com o código acima, pedimos os dois primeiros octetos para o usuário e montamos o laço "for" recebendo os dois valores do dicionário

(chave e dado). Como a chave é uma tupla formada por dois valores, utilizamos o primeiro valor para comparar com o conteúdo da variável “rede”. Caso sejam iguais, vamos exibir os nomes das estações que compõem ou que estão “penduradas” na rede que foi especificada pelo usuário.

O importante, neste momento, é perceber que as duas operações foram simples e utilizamos pouco código devido ao dado “ip” estar desmembrado dentro da tupla. Se o dado do dicionário fosse uma lista (como no exemplo dos usuários), teríamos todas as estruturas em um só exemplo; variáveis, listas, tuplas e dicionários, é assim que elas devem ser utilizadas, dentro das suas características e utilizando o que cada uma tem de melhor. Você não pode, dentro da programação, querer utilizar apenas uma ou outra estrutura, pois cada uma tem um papel específico que cabe melhor, de acordo com o contexto em que será implementada.

Uma outra situação em que podemos utilizar as tuplas é quando usarmos a função “enumerate()” em uma lista. Mas quando precisaremos utilizar a função “enumerate()” em uma lista? Quando quisermos numerar cada componente da lista a fim de garantir que cada elemento dela poderá ser utilizado como dado chave de um dicionário. Vejamos um novo exemplo: crie um novo arquivo chamado: “TuplaEnumerate.py” e monte o código abaixo:

```
usuarios={}
resp="S"
emails=[]
while resp=="S":
    emails.append(input("Digite um e-mail: ").lower())
    resp=input("Digite <S> para continuar: ").upper()
```

Código-fonte 4.17 – Convertendo uma lista em tupla – Parte 1
Fonte: Elaborado pelo autor (2017)

Pensaremos no seguinte cenário: criamos um dicionário para usuários e desejamos que o e-mail do usuário esteja como chave do dicionário e os outros dados, como nome e nível, fiquem como os dados do dicionário no formato de lista. O problema é: os e-mails podem ser repetidos, por isso precisamos enumerá-los. Vamos acrescentar os e-mails em uma lista e, então, gerar tuplas, formadas por um número sequencial e o e-mail propriamente dito, para, então, depois adicionarmos a tupla como chave do dicionário. Vejamos como irá ficar:

```
tupla = list(enumerate(emails))
for chave in range(0,len(tupla)):
    print("Email: ", tupla[chave][1])
    usuarios[tupla[chave]]=[input("Digite o nome"), input("Digite o nível")]
```

Código-fonte 4.18 – Convertendo uma lista em tupla – Parte 2
Fonte: Elaborado pelo autor (2017)

Na primeira linha do código acima, estamos enumerando (função enumerate()) cada item encontrado na lista “e-mails” e gerando uma tupla com cada elemento (função: list()), formados pelo número e pelo e-mail. Em seguida, utilizamos um laço “for” atrelado ao tamanho da nossa tupla, ou seja, à quantidade de e-mails que foram armazenados.

Se foram armazenados três e-mails, teremos que solicitar três nomes e três níveis e assim sucessivamente. Por isso, utilizamos a função range(), que controlará o laço “for” de zero até a quantidade de elementos encontrados na “tupla”. Dentro do laço “for”, vamos, primeiramente, exibir o e-mail que receberá o nome e o nível: tupla[chave][1] => da tupla e recuperar o elemento de acordo com o valor da variável-chave, ou seja, da primeira vez, ela estará valendo zero, por isso pegará o primeiro elemento da tupla.

Cada elemento da tupla é formado por um número (que está na posição zero dentro do elemento da tupla) e o e-mail (que está na posição um dentro do elemento da tupla). O que queremos exibir deste elemento é o e-mail que está na posição um, por isso, o número um entre colchetes.

Após a exibição do e-mail, vamos preencher a chave do dicionário “usuários” com o elemento da tupla já enumerado que foi exibido anteriormente e, então, pediremos o nome e o nível para preencher o dado do objeto do dicionário, lembrando que o nome e o nível estão em formato de lista. Poderíamos exemplificar o nosso dicionário com dois objetos, da seguinte forma:

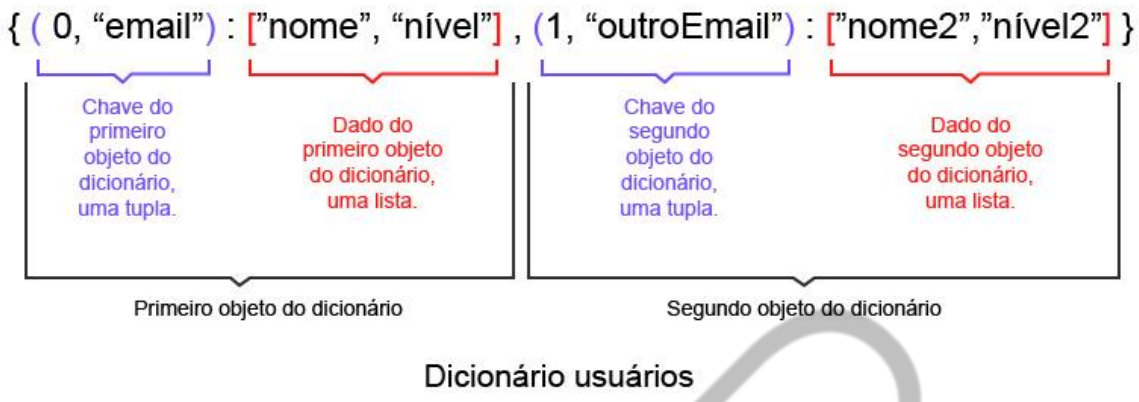

Figura 4.4 – Estrutura de um dicionário com tupla e lista
Fonte: Elaborada pelo autor (2017)

Procure compreender bem o parágrafo anterior, use a imagem como base e perceba que utilizamos todos os recursos vistos até o momento em um único exemplo: variável, dicionário, lista e tupla. Cada um sendo explorado com a sua melhor característica: a simplicidade e a economia de recursos da variável (para controlar o laço); a estrutura perfeita do dicionário; a segurança da tupla pelo fato de não permitir alteração da chave do dicionário; e a flexibilidade da lista no dado do dicionário.

As possibilidades são enormes quando conseguimos entender e aplicar os recursos de acordo com as situações que surgirão. Para completar o nosso código, vamos exibir os dados do dicionário, mostrando, mais uma vez, a transformação dos dados em informação para o nosso usuário final. Adicione o código abaixo:

```
for chave,dado in usuarios.items():
    print("Usuario.:", chave[0])
    print("Email...: ",chave[1])
    print("Nome....: ", dado[0])
    print("Nível...: ", dado[1])
```

Código-fonte 4.19 – Exibindo dados do dicionário com tupla e lista
Fonte: Elaborada pelo autor (2017)

Observe os dados que estamos exibindo, aproveitando suas posições. Primeiro, mostramos os dois dados da chave do dicionário (numeração e e-mail) e, depois, os dois dados do dicionário (nome e nível). Prático, não é mesmo?

Para concluir este capítulo, vale explicar que a String é uma lista com característica de tupla. Uma verdadeira simbiose! Isso quer dizer o seguinte: quando

você armazena uma String como “Defesa”, internamente, foi criada uma lista na qual cada elemento terá um caractere.

Por exemplo, na posição três, você tem a letra “e” e, na posição zero, a letra “D”, entretanto, você não pode alterar o conteúdo de uma posição. Por isso, disse que a String é uma lista (em que cada caractere é um elemento), mas não pode sofrer alteração em uma determinada posição (assim como as tuplas) – ou você altera toda a String ou não altera nada. Vamos abordar esse assunto sobre as Strings com maior profundidade nos capítulos seguintes.

O importante agora é praticar, este momento é crucial para isso. Agora que você já leu o capítulo, procure refazer todos os códigos para que possa fixá-los bem. Depois disso, imagine mais situações em seu projeto para que possa aplicar o que foi visto até aqui. Você pode aproveitar o inventário dos ativos de rede, que fizemos com listas no capítulo 3, e aperfeiçoá-lo com a utilização de dicionários e listas (caso sejam necessários). Utilize o seu projeto como base e coloque a mão na massa, o *hands on* é fundamental para o seu processo de aprendizagem, nunca se esqueça disso.

Neste ponto, talvez vocês já estejam se perguntando, mas qual a necessidade prática dentro do mercado profissional em aprendermos a manipular dados temporários? Pois bem, chegamos em um divisor de águas. Até o momento, a nossa maior preocupação foi passar para vocês as estruturas e fazer com que dominassem todas elas, para, então, agora, entrarmos na persistência dos dados, ou seja, no armazenamento físico.

Uma vez que manipulamos as estruturas, poderemos ler arquivos de textos longos e sem padrão para o usuário final, capturar seus dados e, então, exibir na forma de informação para o nosso usuário. Assim como poderemos preparar estruturas para receber os dados e armazená-las fisicamente em um arquivo, ou seja, o sistema pode ser fechado e, quando for posteriormente executado, as informações poderão ser recuperadas do arquivo.

Não vamos manipular banco de dados, mas os arquivos já abrirão muitas portas e muitas possibilidades para que possamos dar mais vida e mais realidade aos nossos códigos.

O desafio é grande, mas estamos juntos nessa. Vamos para o próximo capítulo, muitas novidades nos aguardam... print(“#DNAFIAP”)!

REFERÊNCIAS

ELMASTI, 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. 2017. Disponível em: <https://www.jetbrains.com/pycharm/download/#section =windows>. Acesso em: 30 nov. 2017.

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