

PYTHON

# MANIPULAÇÃO *DE ARQUIVOS* E *JSON*

HUMBERTO D. DE SOUSA



5

**LISTA DE FIGURAS**

Figura 5.1 – Criação do package ManipulaArquivos .....	7
Figura 5.2 – Arquivo teste.txt que foi criado via Python .....	9
Figura 5.3 – Abrindo arquivo HTML em um browser .....	11
Figura 5.4 – Resultado da leitura de um arquivo com readlines() .....	13
Figura 5.5 – Atualização de plugin para arquivo CSV .....	15
Figura 5.6 – Gerando pacote para Funcoes.....	16
Figura 5.7 – Atualização do arquivo ManagerUsers.py .....	17
Figura 5.8 – Download dos indicadores econômicos da cidade de Boston.....	26
Figura 5.9 – JSON.....	30
Figura 5.10 – Estrutura de um arquivo JSON.....	32

## LISTA DE CÓDIGOS-FONTE

Código-fonte 5.1 – Iniciando a prática com arquivos.....	7
Código-fonte 5.2 – Criar arquivo .....	8
Código-fonte 5.3 – Sobrescrita de arquivo .....	9
Código-fonte 5.4 – Gerando um código HTML.....	10
Código-fonte 5.5 – Leitura de um arquivo texto.....	12
Código-fonte 5.6 – Exemplo para gerenciamento de inventário.....	14
Código-fonte 5.7 – Funções para manipulação do inventário .....	17
Código-fonte 5.8 – Estrutura para chamar as funções de manipulação do inventário .....	18
Código-fonte 5.9 – Realizando slice de String.....	20
Código-fonte 5.10 – Utilizando o método find() .....	22
Código-fonte 5.11 – Desmembrando uma String .....	23
Código-fonte 5.12 – Desmembrando uma String com método Split() .....	25
Código-fonte 5.13 – Total de voos internacionais do aeroporto de Logan .....	28
Código-fonte 5.14 – Extração das informações solicitadas .....	29
Código-fonte 5.15 – Gravando e lendo JSON .....	31
Código-fonte 5.16 – Carregando um arquivo JSON ao abrir.....	33
Código-fonte 5.17 – Verificação da existência do arquivo.....	34
Código-fonte 5.18 – Modularização para manipulação de arquivos JSON .....	35
Código-fonte 5.19 – Arquivo Manipular JSON após modularização.....	35

## SUMÁRIO

5 MANIPULAÇÃO DE ARQUIVOS E JSON.....	5
5.1 Manipulação de arquivos.....	5
5.1.1 Criando um arquivo e adicionando conteúdos .....	6
5.1.2 Realizando a leitura do arquivo .....	11
5.2 Gerando uma rotina para inventário.....	13
5.3 Manipulação de Strings na recuperação dos dados de arquivos. ....	19
5.3.1 Slice de String .....	20
5.3.2 Método find() e o poderoso método split() .....	21
5.4 Manipulando arquivos de terceiros.....	26
5.5 Um caminho para a portabilidade.....	29
5.6 Finalizando o capítulo.....	36
REFERÊNCIAS.....	37

## 5 MANIPULAÇÃO DE ARQUIVOS E JSON

Até o momento, nossos dados só existiam enquanto o sistema estivesse aberto e/ou a sua estrutura não fosse limpa ou sobrescrita. A partir de agora, nossos dados passam a ser mais valorizados pelas nossas rotinas e, então, iremos persistir esses dados em disco. Isso significa que eles passarão a existir fisicamente em uma estrutura de arquivo, o que irá permitir que você transporte os dados coletados de uma máquina para outra, ou até mesmo que atualize um sistema por meio das instruções encontradas em arquivos gerados por outros.

Também será possível gerar um arquivo de script e atualizar a configuração de equipamentos como roteadores e switches, avaliar arquivos de log a fim de encontrar informações sobre comportamentos não autorizados de usuários, realizar auditorias, entre muitas outras possibilidades. Serão apresentadas ainda, algumas funções para a manipulação de Strings de suma importância também para a manipulação de um arquivo já gerado. Um mundo se abrirá a partir desse momento. Não podemos esperar mais... Sigamos em frente, desbravadores!!!!

### 5.1 Manipulação de arquivos

Para a manipulação dos arquivos, utilizaremos uma função denominada “open()”, que permite várias ações:

- **open(“<caminhoDoArquivo><nomeDoArquivo>”, “w”)** => indica que você está abrindo um arquivo para o modo de escrita (w => write), ou seja, permite que você escreva nele, caso o arquivo já exista, ele será sobrescrito. É como se você adquirisse um novo “caderno”, completamente em branco.
- **open(“<caminhoDoArquivo><nomeDoArquivo>”, “r”)** => com a letra “r” (read) no segundo parâmetro da função, você abrirá o arquivo somente em modo de leitura, isso permite que outra pessoa, em outro computador, possa abrir esse arquivo para edição, mas você poderá apenas “consumir” os dados que estiverem dentro desse arquivo. Nesse caso, podemos fazer

a analogia com um livro, ou seja, você não irá preenchê-lo, tão pouco alterá-lo, apenas irá fazer a leitura.

- **open("<caminhoDoArquivo><nomeDoArquivo>", "a")** => dessa forma, você poderá ler e escrever no arquivo especificado, como se fosse um diário, no qual você irá acrescentando conteúdos de acordo com algum evento, periodicidade ou qualquer outro advento do dia a dia, mas sempre no final do diário, ou seja, a principal ideia é seguir concatenando os dados, por isso, a letra "a" referindo-se à "*append*" (anexar). Um exemplo bem prático para isso seria o controle de um arquivo de log, tudo o que for adicionado será acrescentado ao final dos dados que já existem.
- **open("<caminhoDoArquivo><nomeDoArquivo>", "x")** => permite criar um novo arquivo em modo exclusivo (*eXclusive*), ou seja, uma vez que você criou/abriu o arquivo, ninguém mais poderá abri-lo. Caso você tente abrir um arquivo que já existe, será retornada uma falha.
- **open("<caminhoDoArquivo><nomeDoArquivo>", "t")** => o arquivo que for aberto com o parâmetro "t" (text) irá retornar para o Python o seu conteúdo como *string*, diferentemente do parâmetro "b", que retornaria os dados em formato binário e exigiria uma conversão para *string*, caso fosse necessário.

Você pode combinar os modos (texto ou binário) com as ações (*exclusive*, *append*, *write* ou *read*, utilizando ou não o operador "+", por exemplo: "w+b" ou "wb".

### 5.1.1 Criando um arquivo e adicionando conteúdos

Para começarmos, vamos criar uma rotina que irá gerar um arquivo, veremos alguns procedimentos simples e, então, ao término do capítulo, nos aprofundaremos um pouco mais. Começaremos montando nossa estrutura, ou seja, dentro do PyCharm, com o nosso projeto ("**MeusProjetos\_Python**") aberto, crie um novo "Python Package" e atribua ao mesmo o nome: "**3\_3\_Manipula\_Arquivos**". Após isso, gere dentro dele um novo arquivo Python denominado: "**GravarArquivo.py**", conforme apresentado na figura abaixo:

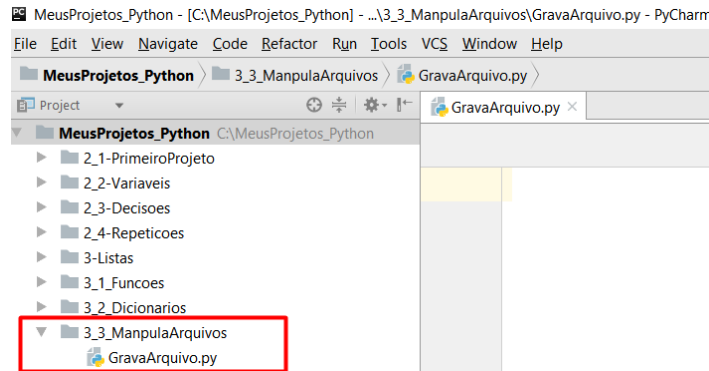


Figura 5.1 – Criação do package ManipulaArquivos  
Fonte: Elaborado pelo autor (2017)

Nesse arquivo, iremos montar o código abaixo. Quando for executá-lo, ele não irá funcionar, explicaremos no parágrafo seguinte:

```
with open("teste.txt", "r") as arquivo:  
    conteudo = arquivo.read()
```

Código-fonte 5.1 – Iniciando a prática com arquivos  
Fonte: Elaborado pelo autor (2017)

Para um primeiro teste, apresentamos a utilização da função **open()** junto ao comando *with*, esse comando permitirá representar, dentro do bloco indentado, o arquivo "teste.txt" por meio do *alias* *arquivo*. Outra grande vantagem que obteremos todas as vezes que formos utilizar a função **open()** combinada ao comando *with* é que o controle do encerramento do arquivo em memória ficará por conta do comando *with*, isso faz com que você não precise utilizar o método **close()**, tampouco ocorrerá o fato do arquivo ficar aberto na memória sem qualquer necessidade. Por isso, recomendo que utilize esse combinado **open() + with**, você não vai se arrepender!

Seguindo a leitura do código apresentado anteriormente, perceba que colocamos, no primeiro parâmetro da função **open()**, o arquivo "teste.txt" sem qualquer caminho sendo especificado. Isso indica que o Python irá considerar o caminho do seu arquivo que contém o código-fonte e que, conseqüentemente, estará sendo executado, ou seja, o caminho adotado para o arquivo "teste.txt" será o mesmo do arquivo "GravarArquivo.py". Caso queira, nada impede de definir um caminho para o arquivo, você deverá substituir "teste.txt" por, por exemplo, "c:\meus\_arquivos\teste.txt". Perceba que o destaque negrito representa o caminho, isso significa que terá que ter acesso ao "c:\\" e que, dentro desta unidade,

deverá existir uma pasta chamada “meus\_arquivos”, uma vez que o método **open()** cria arquivos, não diretórios.

No segundo parâmetro, estamos especificando o valor “r”, que significa que abriremos o arquivo somente para leitura e, dentro do *with*, chamamos a função **read()**, que lê o conteúdo do arquivo. Mas como você pode observar, foi gerado um erro ao tentar executar essas duas linhas de código. Isso aconteceu porque o arquivo não existe, como você pode tentar efetuar a leitura de um arquivo sem que ele exista? O Python, então, retornará a mensagem de erro: **FileNotFoundError: [Errno 2] No such file or directory: 'teste.txt'**. Isso foi apenas para que você pudesse já observar o funcionamento do segundo parâmetro, uma vez que a ideia principal, nessa parte, é a de criar um arquivo e não a de ler o arquivo.

Vamos alterar o código para:

```
with open("teste.txt", "w") as arquivo:  
    arquivo.write("Nunca foi tão fácil criar um arquivo.")
```

Código-fonte 5.2 – Criar arquivo  
Fonte: Elaborado pelo autor (2017)

Dois pontos a serem observados:

- Alteramos o segundo parâmetro de “r” para “w”, ou seja, agora o arquivo será aberto para escrita e não leitura.
- Dentro do **with**, não estamos mais utilizando o método **read()** e, sim, o método **write()**, que nos permite passar o conteúdo que queremos para dentro do arquivo.

Você deve estar se perguntando: mas, se o arquivo “teste.txt” não existe, o que irá ocorrer? Execute o arquivo e veja... Agora funcionou, certo? Isso ocorreu devido à função do parâmetro “w”, que cria o arquivo, se você executar novamente, com outra mensagem, verá que o arquivo anterior foi sobrescrito e repare que o arquivo foi criado junto ao seu arquivo de código, conforme apresentado na imagem a seguir (a não ser que você tenha especificado um caminho diferente para o arquivo):



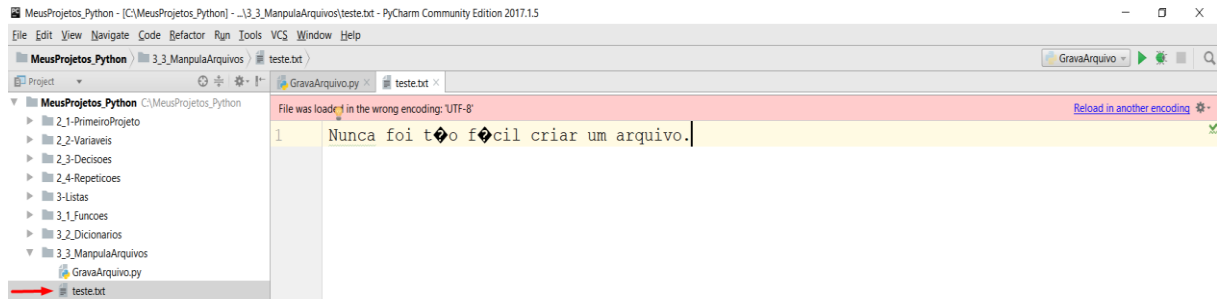


Figura 5.2 – Arquivo teste.txt que foi criado via Python  
Fonte: Elaborado pelo autor (2017)

Repare que os acentos podem estar com caracteres estranhos, isso pode ou não acontecer, depende da configuração da sua IDE (PyCharm) para a leitura de *encodes* para exibição de caracteres do tipo texto. Clique em “*Reload in another encoding*” e opte pela opção **windows-1252** ou **ISO-8859-1**, ou ainda algum outro *encoding* compatível com o seu layout de teclado, idioma e/ou sistema operacional. Pode alterar o *encode*, fechar o arquivo, gerar um novo arquivo (executando seu código) e abri-lo novamente para verificar se a alteração foi realizada com sucesso.

Agora, perceba que, ao colocarmos duas linhas com o método `write()`, o conteúdo das duas linhas será adicionado no arquivo, mas se utilizarmos novamente o método “**with**” com o mesmo arquivo, ele irá sobrescrever o arquivo anterior. Teste o código a seguir e veja como ficará:

```
with open("teste.txt", "w") as arquivo:
    arquivo.write("Nunca foi tão fácil criar um arquivo.")

with open("teste.txt", "w") as arquivo:
    arquivo.write("\nContinuação do texto.")
```

Código-fonte 5.3 – Sobrescrita de arquivo  
Fonte: Elaborado pelo autor (2017)

Esse código irá gerar um novo arquivo e escrever “Nunca foi tão fácil criar um arquivo”. O arquivo será encerrado, em seguida, aberto novamente e, então, a frase “Continuação do texto” será escrita nele, mas, ao abri-lo, você irá encontrar apenas a segunda frase, isso porque o primeiro arquivo foi sobrescrito e não concatenado. Essa é a função da opção “a”, que podemos utilizar no segundo parâmetro, altere somente no segundo *with* o “w” por “a” e veja como irá funcionar melhor agora. Vamos abrir um novo arquivo e gerar um *html* simples, para que você possa ter a noção exata do quão importante pode ser a geração de um arquivo dentro de uma

linguagem de programação. Crie um novo arquivo chamado “ArquivoHTML.py” e monte o seguinte código:

```
with open("pagina.html", "w") as pagina:
    pagina.write("<body> <h1> Esta é um teste para página WEB </h1>")
    pagina.write("<br><h2> Abaixo seguem alguns nomes importantes para o projeto: </h2>")
    pagina.write("<h3>")
    nome=""
    while nome!="SAIR":
        nome = input("Digite um nome ou SAIR: ").upper()
        if nome!="SAIR":
            pagina.write("<br>" + nome)
    pagina.write("</h3></body>")
```

Código-fonte 5.4 – Gerando um código HTML

Fonte: Elaborado pelo autor (2017)

No código, utilizamos a criação do arquivo para gerarmos um arquivo *html*, ou seja, um website simples (afinal, a ideia aqui não é desenvolvê-los “*front end*”) para que veja quantas possibilidades é possível gerar por meio da criação de arquivos. Podem ir desde gerar um simples arquivo texto, passando por uma página html até mesmo criar um código que geraria mais arquivos com códigos. Somente para que entenda de maneira simples, o HTML é desenvolvido entre *tags* (nome atribuído aos seus comandos). Essas *tags* são abertas como o `<h1>` e encerradas sempre com uma barra entre os sinais de menor e maior, como, por exemplo `</h1>`. Vamos às tags que utilizamos no código apresentado:

- `<body>`: indica o início do conteúdo que irá visualizar na página.
- `<h1>`: uma formatação maior para títulos e, à medida que muda o número, a fonte vai diminuindo, por exemplo, `<h2>` tem uma fonte menor que `<h1>`, entretanto, é maior que `<h3>`.
- `<br>`: é uma tag que não precisa ser encerrada, pois sua função é simplesmente dar uma quebra de linha.

Agora que já vimos as *tags* vamos analisar o nosso código: criamos um arquivo chamado “pagina.html”, a extensão “html” é para que o arquivo seja interpretado como uma página pelo seu browser. Logo depois, abrimos o `<body>` e colocamos um título em `<h1>`. Em seguida, abrimos uma nova linha e digitamos um conteúdo com formatação `<h2>`. Abrimos o `<h3>`, criamos uma variável e montamos um laço que irá permitir digitar vários dados nessa variável. Enquanto o seu conteúdo for diferente de “SAIR”, dentro do *while*, o conteúdo da variável será

escrito, na página *html*, somente se não for igual a “SAIR”. Após o laço *while* encerrar, fecharemos a formatação `<h3>` e o `<body>`. Rode sua aplicação, digite alguns nomes conforme for solicitado e, após encerrar a aplicação digitando “SAIR”, você terá um arquivo chamado “pagina.html”. Para abri-lo no browser, clique sobre o arquivo com o botão direito do mouse, escolha a opção “Open in Browser” e o browser que preferir, conforme a figura a seguir apresenta:

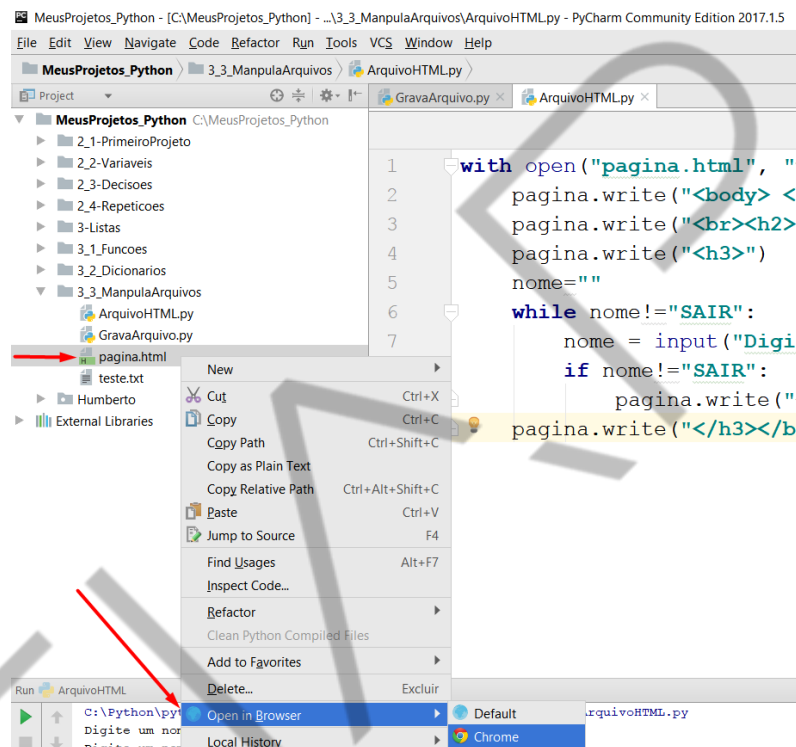


Figura 5.3 – Abrindo arquivo HTML em um browser  
Fonte: Elaborado pelo autor (2017)

Show... Se aprender um pouquinho mais sobre HTML, já poderá gerar relatórios ou dashboards poderosíssimos. Repare que, nesse exemplo, utilizamos apenas variáveis, imagine o que conseguirá fazer acrescentando as listas e os dicionários de dados. Pronto, já pode montar o seu diário de bordo!

### 5.1.2 Realizando a leitura do arquivo

Já vimos como criar os nossos arquivos, então, partiremos, agora, para a leitura do arquivo e isso será fundamental. É muito importante, em qualquer sistema computacional, o armazenamento dos dados, mas isso não traz nada de inovador e/ou agregador ao seu cliente, o diferencial está na leitura dos dados, que deverá

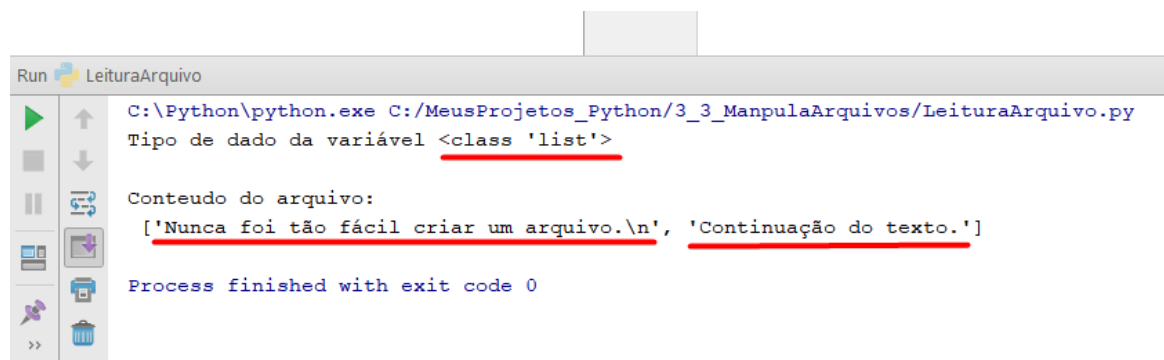
trazer, para o seu cliente, dados na forma de informações, de uma maneira simples e confiável, pois será por meio delas que decisões gerenciais e/ou estratégicas serão tomadas. Por isso, armazenar é importante, mas recuperar de maneira inteligente os dados é muito mais. Vamos para uma prática simples para esquentarmos, crie um novo arquivo de código chamado “LeituraArquivo.py” e digite o seguinte código:

```
with open("teste.txt", "r") as arquivo:
    conteudo=arquivo.read()
print("Tipo de dado da variável", type(conteudo))
print("\nConteúdo do arquivo:\n", conteudo)
```

Código-fonte 5.5 – Leitura de um arquivo texto  
Fonte: Elaborado pelo autor (2017)

Nesse código, abrimos o arquivo, teste.txt, para leitura (“r”) e atribuímos, para a variável “conteudo”, todo o conteúdo do arquivo (quando utilizamos o método **read()**), depois, imprimimos o tipo da variável e, na última linha, imprimimos o conteúdo do arquivo que está dentro da variável. O tipo de dado da variável “conteudo” que será exibido é “str”, ou seja, *string*, isso porque, quando não definimos o modo de saída, o valor-padrão é o “t”(text), por isso “r” ou “rt” ou “r+t” como segundo parâmetro retornarão o mesmo tipo. Podemos mudar para que a saída seja em bytes, para isso, substitua “r” por “rb” ou “r+b” e execute novamente esse bloco de código; verá que não só o tipo mudará para “byte” como também o conteúdo do arquivo a ser exibido terá uma formatação diferente, por *byte*. O mais comum mesmo é fazê-lo utilizando a saída no formato texto, não é à toa que ele é a forma-padrão.

Outra alteração simples refere-se à forma como podemos retornar o conteúdo do arquivo. No código anterior, utilizamos a função **read()**, que retornará todo o conteúdo como se fosse uma única *string*. Façamos o seguinte teste: substitua **read()** por **readlines()** e execute novamente o seu código. Percebeu a diferença? A saída da variável “conteudo” não é mais uma *string* e, sim, uma lista formada por duas posições, uma para cada linha, como podemos observar no resultado, que deve ser semelhante ao da seguinte figura:



```
Run LeituraArquivo
C:\Python\python.exe C:/MeusProjetos_Python/3_3_ManpulaArquivos/LeituraArquivo.py
Tipo de dado da variável <class 'list'>
Conteúdo do arquivo:
['Nunca foi tão fácil criar um arquivo.\n', 'Continuação do texto.']
Process finished with exit code 0
```

Figura 5.4 – Resultado da leitura de um arquivo com `readlines()`

Fonte: Elaborada pelo autor (2017)

Será muito útil a função **`readlines()`** quando optarmos por quebrar um grande arquivo de texto em partes, para que possamos retirar somente os dados que nos interessarem. Por falar nisso, vamos para um exemplo prático, no qual poderemos explorar mais ainda os conceitos, funções e métodos apresentados até o momento.

## 5.2 Gerando uma rotina para inventário

Uma das suas atribuições dentro do projeto será a de gerir todo o inventário de ativos que fazem parte da rede do seu cliente. É fundamental saber quantos são, quais são e onde estão distribuídos os ativos que fizeram parte da rede. Além disso, muitas outras informações podem ser devidamente documentadas, como licenças de softwares, controle sobre o que deve trafegar na rede ou em partes dela, local definido para armazenamento de dados específicos, controle de atualizações, enfim, o gerenciamento e a segurança de uma rede, sem dúvida, começam com um inventário gerenciado, atualizado e funcional.

Montaremos uma rotina pequena e simples, a princípio, para que possamos praticar as técnicas para a manipulação de arquivos vistas até o momento, com uma aplicação prática que, com certeza, fará parte das suas atribuições como um profissional de defesa cibernética. Vamos começar criando um novo arquivo chamado: “Inventario.py” e iremos armazenar apenas os seguintes dados: o número patrimonial do ativo, a descrição do ativo, data da última atualização e o nome do departamento em que está localizado. Primeiro iremos definir uma estrutura de dados para armazená-lo; eles serão recebidos pelo colaborador responsável por catalogar os ativos, e, então, persistiremos os dados para um arquivo, para que

possam ser recuperados, “backupeados”, alterados, excluídos e estejam disponíveis para qualquer outra consulta que possa ser necessária posteriormente.

Utilizaremos uma estrutura de um dicionário de dados para armazená-los enquanto estiverem na condição de dados voláteis, em que a chave será o número patrimonial do ativo (que não pode ser repetido). Sobre essa chave, teremos uma lista com o departamento, data da última alteração e descrição. Após a gerência dos dados no dicionário, iremos persistir em um arquivo do tipo “csv”, que representa um padrão de arquivo no qual os dados poderão ser abertos dentro do Excel e de outros programas que aceitam esse tipo de arquivo. Isso irá garantir maior flexibilidade e portabilidade sobre os dados persistidos. Agora, vamos montar o seguinte código:

```
inventario={}
opcao=int(input("Digite: "
                "\n<1> para registrar ativo"
                "\n<2> para persistir em arquivo"
                "\n<3> para exibir ativos armazenados: "))
while opcao>0 and opcao<4:
    if opcao==1:
        resp="S"
        while resp=="S":
            inventario[input("Digite o número patrimonial: ")] =[
                input("Digite a data da última atualização: "),
                input("Digite a descrição: "),
                input("Digite o departamento: ")]
            resp=input("Digite <S> para continuar.").upper()
        elif opcao==2:
            with open("inventario.csv", "a") as inv:
                for chave, valor in inventario.items():
                    inv.write(chave + ";" + valor[0] + ";" +
                               valor[1] + ";" + valor[2]+"\n")
                print("Persistido com sucesso!")
        elif opcao==3:
            with open("inventario.csv", "r") as inv:
                print(inv.readlines())
    opcao=int(input("Digite: "
                    "\n<1> para registrar ativo"
                    "\n<2> para persistir em arquivo"
                    "\n<3> para exibir ativos armazenados: "))
```

Código-fonte 5.6 – Exemplo para gerenciamento de inventário  
Fonte: Elaborado pelo autor (2017)

Vejamos alguns pontos importantes para serem abordados:

- Criamos um dicionário de dados, chamado “inventario”, e montamos um menu de escolha, com as opções 1, 2 ou 3. Enquanto o usuário digitar qualquer um dos três valores, o programa continuará sendo executado, entretanto, para qualquer outro valor, o programa será encerrado.

- Se o valor digitado for “1”, iremos colocar o usuário em outro laço “while” e, enquanto ele digitar “S”, será permitido a ele adicionar itens para o nosso dicionário “inventario”.
- Se o valor digitado for “2”, abriremos o arquivo “inventario.csv” em modo de concatenação e, então, para cada objeto encontrado no nosso dicionário, iremos adicionar um linha no arquivo. Na linha, escreveremos a chave e os valores desta chave separados por “;” (para o Excel, isso indicará que cada valor deverá ficar em uma coluna). Ao término da linha, adicionamos uma quebra de linha “\n”, isso também indicará ao Excel uma quebra de linha, indicando que temos, na sequência, outro objeto.
- Se finalmente o valor digitado for “3”, então, abriremos o arquivo em modo de leitura e utilizaremos o método “readlines()” para exibirmos todo o conteúdo do arquivo.

Após a execução do seu projeto e a verificação de que esteja tudo ok, abra dentro do próprio PyCharm o seu arquivo “csv”, assim como fizemos com o arquivo “txt”, e observe que talvez, dependendo da sua instalação, será necessária uma atualização no seu PyCharm, conforme apresentado na figura a seguir:

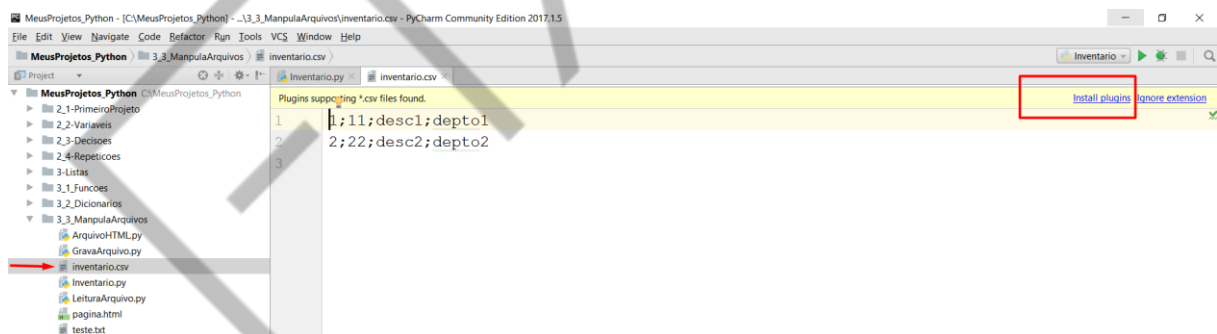


Figura 5.5 – Atualização de plugin para arquivo CSV  
Fonte: Elaborado pelo autor (2017)

Faça a instalação e reinicie o seu PyCharm para que o plugin possa estar devidamente configurado. Instalar o plugin é importante para que você possa explorar todos os recursos adicionais que a IDE poderá lhe oferecer para a manipulação de arquivos “CSV”, nesse caso. Essa atualização pode ser necessária para outros conteúdos que veremos no decorrer do curso. Sempre que puder, realize-as para que tenha uma IDE com o máximo de recursos possíveis.



Procure abrir o seu arquivo no Excel, caso o tenha instalado em seu computador, você verá que os itens estarão devidamente separados e que você poderá manipulá-los, ordená-los e realizar qualquer outra operação necessária com os dados que foram persistidos no arquivo CSV.

Perceba que o nosso código ficou um pouco extenso, e oferecemos poucas funções para o usuário. Isso significa que seria interessante, para esses exemplos, aplicarmos o conceito das funções e da modularização. Por isso, vamos criar as funções que simplificariam esse código.

Seguiremos aprimorando nossa estrutura de projeto e, em vez de criarmos simplesmente um arquivo com as funções, vamos criar um “Python Package” chamado “Funcoes” e, dentro desse pacote, iremos criar todos os nossos arquivos que armazenarão funções. Começaremos criando o arquivo “Funcoes\_Arquivos.py”, aproveite também para arrastar o arquivo “Funcoes.py” que estava dentro do pacote “3\_2\_Dicionarios” para o nosso pacote “Funcoes”. Sua tela deverá ficar de acordo com a imagem a seguir:



Figura 5.6 – Gerando pacote para Funcoes  
Fonte: Elaborado pelo autor (2017)

Renomeie o arquivo “Funcoes.py” para “Funcoes\_Dicionarios.py”, somente a fim de identificar sua origem. Para renomear, clique com o botão direito sobre o



arquivo, escolha a opção “Refactor” e, em seguida, “Rename”. Agora, atualize o arquivo “ManagerUsers.py”, que está no pacote “3\_2\_Dicionarios”, conforme a imagem a seguir:

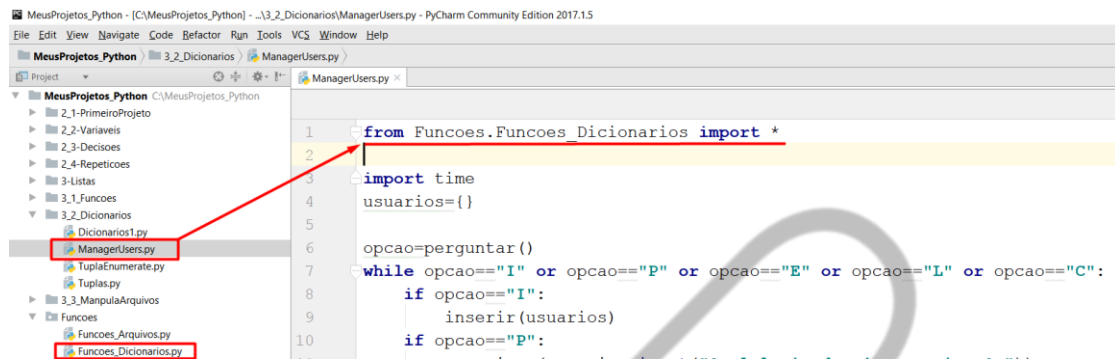


Figura 5.7 – Atualização do arquivo ManagerUsers.py  
Fonte: Elaborado pelo autor (2017)

Pronto, dessa forma, todos os nossos arquivos que armazenem funções estarão localizados em um mesmo pacote, o que nos facilitará reaproveitar funções entre projetos diferentes e também padronizar as importações.

Agora, dentro do arquivo “Funcoes\_Arquivos.py”, monte o código seguinte:

```
def chamarMenu():
    escolha = int(input("Digite: "
                        "\n<1> para registrar ativo"
                        "\n<2> para persistir em arquivo"
                        "\n<3> para exibir ativos armazenados: "))
    return escolha

def registrar(dicionario):
    resp="S"
    while resp=="S":
        dicionario[input("Digite o número patrimonial: ")] =[
            input("Digite a data da última atualização: "),
            input("Digite a descrição: "),
            input("Digite o departamento: ")]
        resp=input("Digite <S> para continuar.").upper()

def persistir(dicionario):
    with open("inventario.csv", "a") as inv:
        for chave, valor in dicionario.items():
            inv.write(chave + ";" + valor[0] + ";" +
                      valor[1] + ";" + valor[2] + "\n")
    return "Persistido com sucesso"

def exibir():
    with open("inventario.csv", "r") as inv:
        linhas=inv.readlines()
    return linhas
```

Código-fonte 5.7 – Funções para manipulação do inventário  
Fonte: Elaborado pelo autor (2017)

Foram criadas, no código apresentado, quatro funções:

- A função chamarMenu() apenas exibe para o usuário as opções que ele possui para a manipulação do inventário e retorna o valor selecionado por meio da variável “escolha”;
- A função registrar() irá receber o dicionário responsável por armazenar os ativos e, então, irá preenchê-lo, enquanto o usuário digitar “S”;
- A função persistir() ficou encarregada de descarregar o dicionário que foi passado como parâmetro para dentro do arquivo “inventario.csv”; e
- A função exibir() irá, finalmente, retornar o conteúdo das linhas do arquivo “inventario.csv” por meio da variável “linhas”.

Agora, vamos atualizar o nosso arquivo “Inventario.py”, com o seguinte código:

```
from Funcoes.Funcoes_Arquivos import *
inventario={}
opcao=chamarMenu()
while opcao>0 and opcao<4:
    if opcao==1:
        registrar(inventario)
    elif opcao==2:
        persistir(inventario)
    elif opcao==3:
        print(exibir())
opcao = chamarMenu()
```

Código-fonte 5.8 – Estrutura para chamar as funções de manipulação do inventário  
Fonte: Elaborado pelo autor (2017)

O código do nosso arquivo “Inventario.py” tornou muito mais simples e muito mais fácil aplicar alterações e/ou manutenções. Uma atenção para a linha na qual chamamos o método exibir(), pois o chamamos dentro de um print() que será responsável por exibir tudo o que método exibir() retornar.

Repare, também, algo interessante. O arquivo “inventario.csv” é utilizado dentro das funções que estão no arquivo Funcoes\_Arquivo.py, que, por sua vez, está no pacote “Funcoes”, já o arquivo “inventario.csv” está no pacote “3\_3\_ManipulaArquivos”, ou seja, estão em pacotes diferentes. Mesmo assim, as funções reconhecem o arquivo “inventario.csv”, pois o caminho que irão utilizar é o

caminho do qual forem “chamadas”, ou seja, do arquivo “Inventario.py”, que é o mesmo do arquivo “inventario.csv”.

Agora, repare onde será fundamental o programador Python... A saída dos dados do “inventario.csv”, pelo Python, está clara? Se tivermos 200 ativos cadastrados, imagine como ficaria a saída! Uma bagunça, não é mesmo? E se quiséssemos saber, por exemplo, quais as descrições dos ativos que pertencerem a um departamento, em específico? Ou, ainda, quais os ativos que foram atualizados hoje? Agora, caberá a você recuperar esses dados e trabalhá-los da melhor forma possível, trazendo inteligência para o seu gerenciamento do ambiente. Será fundamental, neste momento, a manipulação de *Strings*, e é com esse assunto que iremos trabalhar no próximo tópico.

### 5.3 Manipulação de Strings na recuperação dos dados de arquivos

No tópico anterior, paramos com a missão de melhorarmos a recuperação dos dados de um arquivo. Atualmente, o nosso arquivo “Inventario.py” retorna todas as informações do arquivo “inventario.csv” de uma forma nada agradável, por exemplo, para que se possa apresentar em uma reunião. Começaremos pelo seguinte: o número patrimonial do ativo, normalmente, não é um dado importante para ser exibido, salvo em consultas específicas.

É como se eu fosse até o supermercado e comprasse um refrigerante, mas, antes, quisesse saber o código que representa o refrigerante no estoque do supermercado... Insano, não é mesmo? Imagine-se, agora, fazendo as compras do mês e querendo saber todos os códigos de todos os produtos, ou seja, temos um dado (código do produto), para essa situação específica, que não agrega absolutamente nada, tornando-se um dado descartável para a situação.

O mesmo podemos pensar sobre o número do patrimônio, que, para um relatório simples, não se faz necessário. A nossa função `exibir()` retorna todos os elementos dos ativos em forma de lista, e cada elemento da lista é um ativo que possui: número patrimonial, data da última alteração, descrição e departamento, todos separados por “;”. Teremos que tratar cada linha (ativo) do nosso arquivo.

### 5.3.1 Slice de String

Vejamos como poderíamos retirar esse código durante a exibição simples dos ativos. Para o nosso caso, vamos alterar o nosso arquivo “Inventario.py”, conforme demonstrado a seguir:

```
from Funcoes.Funcoes_Arquivos import *
inventario={}
opcao=chamarMenu()
while opcao>0 and opcao<4:
    if opcao==1:
        registrar(inventario)
    elif opcao==2:
        persistir(inventario)
    elif opcao==3:
        resultado = exibir()
        for linha in resultado:
            print(linha[2:-1])
opcao = chamarMenu()
```

Código-fonte 5.9 – Realizando slice de String  
Fonte: Elaborado pelo autor (2017)

As linhas em vermelho são as linhas que foram alteradas e/ou inseridas, em relação ao código que já existia. Vamos aos detalhes:

- Primeiro, atribuímos para a variável “resultado” os dados que forem retornados pela função `exibir()`, ou seja, o conteúdo do arquivo “inventario.csv” em forma de lista.
- Montamos um “for” para percorrer toda a lista, isso porque não queremos que saia o número do patrimônio em todos os ativos.
- De cada linha, iremos exibir somente do terceiro caractere [2] até o último [-1]. Repare que fizemos esse recorte no elemento da lista por meio do recurso “[2:-1]”, que se chama “*slice*” e representa um recorte em uma String. Cada elemento da lista é uma *string*, e, por isso, podemos fatiá-la e exibir somente o que desejamos. Toda *string* tem, para o seu primeiro caractere, a posição 0, logo, estamos descartando a posição 0 (número do patrimônio) e a posição 1 (o “;”), exibindo do caractere 2 (primeiro dígito da data da última atualização) até o final, que é representado pelo -1. Logo, a primeira linha do meu arquivo está com:

```
1;11/11/2017;Impressora XPTO Laser;Recepção
```

**Mas será exibido apenas:**

```
11/11/2017;Impressora XPTO Laser;Recepção
```

O recurso “slice” facilita muito a manipulação de *Strings*, mas devemos tomar muito cuidado, pois, às vezes, a solução não é tão simples. Vejamos o que ocorreria se a minha primeira linha do arquivo estivesse com o seguinte conteúdo:

```
10;11/11/2017;Impressora XPTO Laser;Recepção
```

O que seria “fatiado” dessa linha???? Apenas o “1” e o “0”, isso significa que o “;” seria exibido. E se, em vez do “10”, tivéssemos o “100” ou o “1000”? Percebe que ficou engessado, quando disse querer um slice de [2;-1]? Na verdade, não é essa a lógica correta, o necessário é que ele retorne do primeiro caractere, após o primeiro “;” encontrado, até o final da *string*. Para isso, precisaremos localizar o “;”, ou seja, a sua posição dentro da *string*, e é aí que entra o método **find()**.

### 5.3.2 Método find() e o poderoso método split()

O método find(), também utilizado por *Strings*, permite que você encontre um caractere e/ou uma parte da *string* (conjunto de caracteres), retornando a posição da primeira incidência encontrada. Por exemplo: caso tenhamos a *string* “AmoPython” e executarmos um comando como: “AmoPython”.find(“o”), ele irá retornar o valor 2, pois a primeira vogal “o” se encontra na posição 2. Lembre-se de que a *String* começa do elemento zero. E caso ele não encontre o caractere desejado, irá retornar o valor “-1”. Agora, o nosso código do arquivo “Inventario.py” deverá ficar da seguinte forma:

```
from Funcoes.Funcoes_Arquivos import *
inventario={}
opcao=chamarMenu()
while opcao>0 and opcao<4:
    if opcao==1:
        registrar(inventario)
    elif opcao==2:
        persistir(inventario)
    elif opcao==3:
        resultado = exibir()
        for linha in resultado:
            print(linha[linha.find(";")+1:-1])
opcao = chamarMenu()
```

Código-fonte 5.10 – Utilizando o método find()

Fonte: Elaborado pelo autor (2017)

O que está na cor vermelha, no código apresentado, representa a única alteração que fizemos. Substituímos o número “2” pela implementação do método find(), ele irá retornar a posição do primeiro “;” encontrado e, então, acrescentará +1 para que não exiba o próprio “;” na saída. Agora, se tivermos, em nosso arquivo “inventario.csv”, linhas como:

```
10;11/11/2017;Impressora XPTO Laser;Recepção
200;22/11/2017;Estação ABC 4GB RAM HD 1TB Processador
X;Recepção
3;26/11/2017;Estação ABC 4GB RAM HD 1TB Processador X;Compras
```

A saída, mesmo com números de patrimônios com quantidade de dígitos distintos, será:

```
11/11/2017;Impressora XPTO Laser;Recepção
22/11/2017;Estação ABC 4GB RAM HD 1TB Processador X;Recepção
26/11/2017;Estação ABC 4GB RAM HD 1TB Processador X;Compras
```

Fácil, não é mesmo? Mas nossa saída ainda não está tão clara assim, poderíamos separar os dados: data da última atualização da descrição e do departamento. E, lógico, você deve ter pensado em uma solução como:

```
for linha in resultado:
    separacao=linha[linha.find(";")+1:-1]
    data=separacao[0:separacao.find(";")]
    separacao = separacao[separacao.find(";")+1:-1]
    descricao=separacao[0:separacao.find(";")]
    departamento=linha[linha.rfind(";")+1:-1]
    print("Data.....: ", data)
    print("Descrição.....: ", descricao)
    print("Departamento..: ", departamento)
opcao = chamarMenu()
```

Código-fonte 5.11 – Desmembrando uma String

Fonte: Elaborado pelo autor (2017)

As linhas na cor vermelha são aquelas inseridas e/ou alteradas para que possamos pegar exatamente cada informação dentro de cada linha que é recuperada do arquivo “inventario.csv”. Primeiro, utilizamos uma variável “separacao” e colocamos, dentro dessa variável, o valor da variável “linha” de onde encontrar o “;” até o final, ou seja, estaremos descartando o número patrimonial. Com o conteúdo da minha primeira linha do arquivo, os valores das variáveis estariam assim, até o momento:

```
linha = 10;11/11/2017;Impressora XPTO Laser;Recepção
separacao = 11/11/2017;Impressora XPTO Laser;Recepção
```

Para extrair somente a data, aplicaremos o find() novamente, mas, desta vez, dentro da variável separação. Repare que mudaremos a posição da qual utilizaremos o find(), porque não precisamos mais dele definindo o início do valor que desejamos, mas, sim, para delimitar o final do que desejamos. Observe, no valor da variável “separacao” apresentada, a data que desejamos. Ela começará na posição zero, invariavelmente; já no término, precisaremos demarcar com o primeiro “;” que for encontrado, por isso, utilizamos para a variável data, o valor:

```
data = separacao[0:separacao.find(";")]
```

Dessa forma, os conteúdos das variáveis estarão assim:

```
linha = 10;11/11/2017;Impressora XPTO Laser;Recepção
separacao = 11/11/2017;Impressora XPTO Laser;Recepção
data = 11/11/2017
```

Agora, precisaremos da descrição, para isso, iremos descartar a “data” da variável “separacao”, assim como descartamos, no início, o número patrimonial. Vamos recortar a nossa variável “separacao” por meio da linha:

```
separacao = separacao[separacao.find(";")+1:-1]
```

A partir dessa linha, as variáveis passarão a conter os seguintes valores:

```
linha = 10;11/11/2017;Impressora XPTO Laser;Recepção  
separacao = Impressora XPTO Laser;Recepção  
data = 11/11/2017
```

Assim, fica mais fácil recuperar a descrição que começará, invariavelmente, na posição zero (0) e terminará quando for encontrado o primeiro “;”. Isso justifica a linha:

```
descricao=separacao[0:separacao.find(";")]
```

Nessa linha, preenchemos a variável “descricao” com o conteúdo da variável “separacao”, utilizando os caracteres da posição zero até o “;”. Agora, as variáveis estão com os seguintes valores:

```
linha = 10;11/11/2017;Impressora XPTO Laser;Recepção  
separacao = Impressora XPTO Laser;Recepção  
data = 11/11/2017  
descricao = Impressora XPTO Laser
```

Falta apenas o departamento o qual podemos perceber, olhando da direita para a esquerda, representado pelos caracteres até que se encontre o primeiro “;” (**considerando a leitura da direita para a esquerda**) na variável “linha” e, para considerarmos a leitura da direita para esquerda, utilizaremos o método rfind() (r – right - direita). Os métodos find() e rfind() possuem o mesmo objetivo, entretanto, o primeiro faz a leitura da esquerda para a direita e o segundo da direita para a esquerda.

Assim, ficou fácil recuperar o departamento no qual queremos os caracteres a partir do primeiro “;” (sentido da direita para esquerda) até o final, por isso, a linha:

```
departamento=linha[linha.rfind(";")+1:-1]
```

Agora, as variáveis estarão com os seguintes valores:



```
linha = 10;11/11/2017;Impressora XPTO Laser;Recepção
separacao = Impressora XPTO Laser;Recepção
data = 11/11/2017
descricao = Impressora XPTO Laser
departamento = Recepção
```

Finalmente, conseguimos fazer a devida separação e, então, exibirmos por meio dos prints, conforme o código apresenta. A saída ficou bem melhor agora, não é mesmo?

MAS... foi prático fazer tudo isso? Imagine se tivéssemos 40 dados em vez de três (data, descrição e departamento)! Seria uma loucura, não é mesmo? Por isso, no Python, nós temos o método `split()`, um verdadeiro herói! Ele quebra uma *String* de acordo com um conjunto de caracteres que você pode definir. Se olharmos o conteúdo da nossa variável “linha”, perceberemos que os nossos dados são separados por um “;” (daí a importância em não separar os dados com um símbolo comum qualquer, procure sempre utilizar símbolos que dificilmente existirão dentro de uma *String*, como: #, !, ##, entre outros). A função `split()` irá gerar uma lista, e, em cada posição, teremos uma parte da *String*, de acordo com a quebra que foi proposta. Por exemplo:

`lista="AmoPython".split("o")` => teremos uma lista com três posições: na primeira posição, teremos “Am”; na segunda, teremos “Pyth”; e, na terceira, “n”, uma vez que quebramos a *String* por meio da vogal “o”. Essa função facilitará muito o nosso trabalho, observe como o código ficará agora:

```
for linha in resultado:
    lista=linha.split(";")
    print("Data.....: ", lista[1])
    print("Descrição.....: ", lista[2])
    print("Departamento.: ", lista[3])
opcao = chamarMenu()
```

Código-fonte 5.12 – Desmembrando uma *String* com método `Split()`

Fonte: Elaborado pelo autor (2017)

Compare com a solução anterior e veja o quão simples foi essa solução proposta. Apesar de as duas funcionarem e retornarem as informações desejadas, a segunda proposta, com o método `split()`, é muito mais prática e rápida de ser implementada. Sempre que manipular *Strings*, terá várias formas de chegar ao

mesmo resultado, mas, quando achar que o seu código está muito extenso e repetitivo, procure saber se não existe algum método capaz de simplificar o que você deseja realizar.

## 5.4 Manipulando arquivos de terceiros

Até o momento, manipulamos dados e arquivos que nós mesmos organizamos e criamos e, sem dúvida, esse é o mundo perfeito. Entretanto, em muitas ocasiões, você deverá realizar a leitura e recuperar dados de arquivos de terceiros, como, por exemplo, logs gerados pelo sistema operacional, arquivos de configuração de roteadores e outros equipamentos ou ainda arquivos gerados por ferramentas de outros programadores.

Por isso, nesse tópico, realizaremos o tratamento dos dados de um arquivo de terceiro, a fim de praticarmos e fixarmos os métodos, sobre *Strings* e arquivos, abordados até o momento e também para que possamos praticar um pouco mais a lógica para o trabalho com arquivos. Vamos lá.

Comece acessando o link: <https://data.boston.gov/dataset/economic-indicators-legacy-portal>, que irá direcioná-lo para uma página que permitirá realizar o download de um arquivo “csv”, que contém indicadores oficiais relacionados à economia da cidade de Boston-EUA, conforme apresentado na figura a seguir:

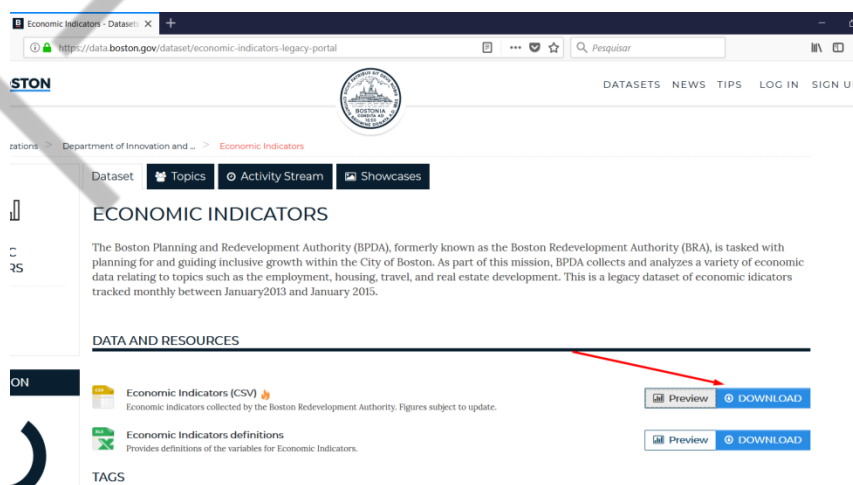


Figura 5.8 – Download dos indicadores econômicos da cidade de Boston  
Fonte: Elaborado pelo autor (2017)

Após baixar o arquivo, arraste-o para dentro do pacote 3\_3\_ManipulaArquivos e crie o arquivo Boston.py. Abra o arquivo “economic-indicators.csv” e você irá encontrar, na primeira linha, as siglas dos indicadores e, nas linhas subsequentes, os dados separados por vírgula, seguindo a ordem dos indicadores. Para facilitar nossa análise, seguem, a seguir, os significados das siglas, encontradas na primeira linha do arquivo, que utilizaremos para o nosso exemplo:

- Year = ano do dado
- Month = mês do dado
- logan\_passengers = quantidade de passageiros, internacionais e domésticos, que passaram pelo aeroporto de Logan.
- logan\_intl\_flights = total de voos internacionais que partiram do aeroporto de Logan.
- hotel\_occup\_rate = porcentagem de vagas ocupadas nos hotéis de Boston.
- hotel\_avg\_daily\_rate = média da diária, em dólar, de um hotel em Boston.
- total\_jobs,unemp\_rate = porcentagem de desempregados em Boston.

Deveremos buscar as respostas às seguintes questões:

- Qual o total de voos internacionais que partiram do aeroporto de Logan no ano de 2014?
- Quando (mês/ano) ocorreu o maior trânsito de passageiros no aeroporto de Logan?
- Qual o total de passageiros que passaram pelo aeroporto de Logan, no ano que for especificado pelo usuário?
- Qual o mês que possui a maior média da diária de um hotel, com base no ano especificado pelo usuário?

Para responder a essas perguntas, deveremos montar nossa ferramenta. Por isso, no arquivo “Boston.py”, digite o código seguinte para resolver a primeira questão:

```
with open("economic-indicators.csv", "r") as boston:
    total=0
    for linha in boston.readlines()[1:-1]:
        total=total+float(linha.split(",")[3])
    print("O total de voos é: ", total)
```

Código-fonte 5.13 – Total de voos internacionais do aeroporto de Logan

Fonte: Elaborado pelo autor (2017)

Detalhando um pouco o código apresentado:

- Na primeira linha, abrimos o arquivo em modo somente de leitura;
- Criamos uma variável para armazenar o total de voos;
- Montamos um laço “foreach” para percorrermos somente da linha um “1” até a última linha válida, descartando a linha zero “0” (em que temos apenas os títulos);
- Atribuímos, na variável total, o valor do terceiro elemento da lista que foi criada por meio do split(), separando os elementos por vírgula(“,”);
- Imprimimos o conteúdo da variável, quando o laço for encerrado e a variável “total” tiver acumulado todos os valores do arquivo.

Agora, complemente o arquivo resolvendo as outras questões, procure resolver uma a uma e siga abaixo para ver a solução proposta para a versão final do nosso arquivo “Boston.py”.

```
with open("economic-indicators.csv", "r") as boston:
    total_voos=0
    maior=0
    total_passageiros=0
    maior_media_diaria=0
    ano_usuario = input("Qual ano deseja pesquisar? ")
    for linha in boston.readlines()[1:-1]:
        lista=linha.split(",")
        total_voos=total_voos+float(lista[3])
        if float(lista[2])>float(maior):
            maior=lista[2]
            ano=lista[0]
            mes=lista[1]
        if ano_usuario==lista[0]:
            total_passageiros=total_passageiros+float(lista[2])
            if float(lista[5])>float(maior_media_diaria):
                maior_media_diaria=lista[5]
                mes_maior_diaria=lista[1]
    print("O total de voos é: ", total_voos)
    print("O mês/ano de maior movimento no aeroporto foi: ",
          str(mes),"/",str(ano))
    print("O total de passageiros do ano ", ano_usuario,
          "foi de ", total_passageiros)
    print("O mês do ano ", ano_usuario,
```

```
"com maior média para diária de hotel foi ", mes_maior_diaria)
```

Código-fonte 5.14 – Extração das informações solicitadas

Fonte: Elaborado pelo autor (2017)

Para a resolução das perguntas que foram realizadas anteriormente, você pode observar, no código acima, a criação das quatro variáveis que servirão como base para a resposta, mais uma variável que irá armazenar o ano específico que o usuário deseja consultar. Dentro do “foreach”, foi realizada a identificação dos dados e as respectivas atribuições, até o seu final, em que, então, foram apresentados os valores das variáveis nas últimas quatro linhas do código.

Agora, você mesmo pode tentar localizar algumas informações, baixar outros arquivos “csv” e fazer o filtro que desejar, como, por exemplo, tentar localizar o mês do ano que possui a maior taxa de ocupação das vagas nos hotéis em Boston ou ainda em que mês/ano ocorre a menor taxa de desemprego na cidade de Boston. Enfim, são inúmeras possibilidades, e o céu é o limite.

## 5.5 Um caminho para a portabilidade

Já trabalhamos com arquivos que nós mesmos geramos e com arquivos de terceiros, mas você também deve pensar em outros profissionais que podem precisar realizar leitura sobre os seus arquivos, e pensando um pouco maior ainda, os seus arquivos gerados pelo Python podem ser exportados para outras plataformas, até mesmo por uma aplicação mobile. O que desejamos dizer é que não basta o Python ser portátil, os dados que você gera por meio dele também devem ter a qualidade da portabilidade.

Imagine que você irá desenvolver alguma ferramenta que deverá se comunicar com outra aplicação, como, por exemplo, se conectar aos Correios, à Receita Federal, ao Detran ou, simplesmente, à aplicação de um programador que fala outro idioma em outro país... Como será possível estabelecer uma comunicação entre a sua aplicação e qualquer outra aplicação do mundo que queira trocar dados? Simples: utilizando um padrão de comunicação.

Um padrão muito comum utilizado pelos programadores é o XML, que permite a troca de dados entre plataformas distintas. Uma prova disso é a nota fiscal eletrônica, que roda em todo o País e não importa se você usa Windows 32 bits, 64

bits, Linux, iOS, Solaris ou qualquer outro sistema operacional, tampouco importa a linguagem que foi utilizada para desenvolver a ferramenta e não importa também se o programa que irá gerar a nota fiscal for realizado em .Net, Java, PHP, Python ou qualquer outra linguagem. O único fator que importa é que os dados sejam criados dentro do padrão XML. Mas não abordaremos o XML, que é mais comumente utilizado para sistemas comerciais, como é o caso da nota fiscal eletrônica.

Abordaremos outro padrão, também muito utilizado, para que você garanta que a saída dos dados da sua aplicação tenha portabilidade, esse é o tal JSON!



Figura 5.9 – JSON  
Fonte: NPM (2017)

JSON é uma sigla, **JavaScript Object Notation**, que representa um padrão para armazenamento de dados, cuja principal função é permitir a troca de dados entre aplicações e plataformas distintas. O Python possui um módulo para facilitar o trabalho com esse tipo de padronização. Vamos a um exemplo prático, crie um arquivo chamado “Manipular\_JSON.py” e digite o código a seguir:

```
import json
inventario={}
opcao=int(input("Digite: "
                "\n<1> para registrar ativo"
                "\n<2> para exibir ativos armazenados: "))
while opcao>0 and opcao<3:
    if opcao==1:
```

```
resp = "S"
while resp == "S":
    inventario[input("Digite o número patrimonial: ")] = [
        input("Digite a data da última atualização: "),
        input("Digite a descrição: "),
        input("Digite o departamento: ")]
    resp = input("Digite <S> para continuar.").upper()
    with open("inventario_json.json", "w") as arq_json:
        json.dump(inventario, arq_json)
    print("JSON gerado!!!!")
elif opcao==2:
    with open("inventario_json.json", "r") as arq_json:
        resultado = json.load(arq_json)
        for chave, dado in resultado.items():
            print("Data.....: ", dado[0])
            print("Descrição....: ", dado[1])
            print("Departamento.: ", dado[2])
opcao = int(input("Digite: "
    "\n<1> para registrar ativo"
    "\n<2> para exibir ativos armazenados: "))
```

Código-fonte 5.15 – Gravando e lendo JSON

Fonte: Elaborado pelo autor (2017)

Como você já percebeu, o exemplo adotado foi o mesmo do tópico anterior, para que você possa ter um parâmetro entre o código que manipula um arquivo “txt”, “csv” ou “html” e o código que utiliza o padrão JSON.

O primeiro detalhe a ser observado está na primeira linha do código, o “import”. Como já afirmamos anteriormente, o Python possui um módulo para a manipulação de JSON e, para que possamos utilizar os seus objetos e métodos, devemos importá-lo.

Perceba que, nesse exemplo, retiramos a antiga opção “2”, que era responsável por persistir no arquivo, pois agora realizamos a persistência logo após o usuário digitar todos os dados no dicionário “inventario”, assim que o loop for encerrado, já realizamos a persistência no formato JSON.

Para gravarmos o dicionário “inventario” no arquivo, utilizamos o método dump(), que pertence ao objeto “json”. Esse método é formado, basicamente, por dois parâmetros: o que será gravado, no nosso caso, o dicionário “inventario” e onde será gravado, quando definimos por meio do alias “arq\_json”, que aponta para o arquivo “inventario\_json.json”, o qual estará aberto em modo escrito, ou seja, caso ele não exista será criado automaticamente e, caso já exista, será sobrescrito.

## IMPORTANTE

Por que o arquivo não foi aberto para concatenação, assim como fizemos no tópico anterior? Porque, quando o método `dump()` fosse chamado, após a primeira vez, ele não iria adicionar as informações do dicionário “inventario” ao arquivo, mas criar uma nova estrutura dentro do arquivo.

Execute e lance dois ativos no inventário, depois disso, encerre sua aplicação e veja que surgiu o arquivo “inventario\_json.json”. E, quando você der o duplo clique nele, aparecerá o seu conteúdo, que deverá estar semelhante ao da imagem abaixo:

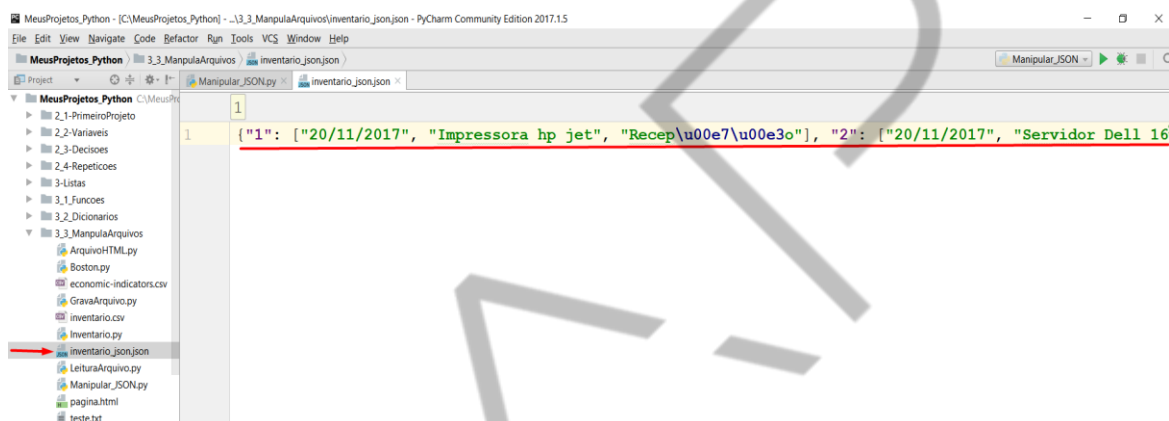


Figura 5.10 – Estrutura de um arquivo JSON  
Fonte: Elaborado pelo autor (2017)

E, então, reconheceu algo na estrutura do arquivo? Parece com alguma estrutura de dados que vimos dentro do Python? Sim, isso mesmo! É o dicionário de dados, isso significa que a estrutura do JSON não o assusta, pois já manipulamos bastante a estrutura de dicionário de dados por meio da chave e dos dados.

Continuando a explicação do código, na opção “2”, abrimos o nosso arquivo em modo de leitura apenas e, então, chamamos o método `load()`, que também pertence ao objeto “json”. Ele descarrega o arquivo informado no parâmetro para uma variável, no nosso exemplo, a variável “resultado”. Em seguida, montamos um laço “foreach”, que irá trazer a chave (número patrimonial) e os dados (uma lista com data, descrição e departamento); finalizamos com a exibição e o menu de opções novamente.

Agora perceba que, como não utilizamos, na opção “1”, o modo de concatenação, se você executar novamente o seu código, verá que o seu arquivo anterior será sobrescrito, isto é, se você lançar um ativo, o seu arquivo ficará apenas



com um ativo e não mais com três (considerando os outros dois que você preencheu anteriormente). Para solucionar isso, faremos uma alteração na estrutura do nosso código, ao invés de criarmos o dicionário “inventario” vazio, já o criaremos preenchido pelo arquivo, assim, a nossa opção “1” irá adicionar mais um objeto ao dicionário, que já estará preenchido com o conteúdo do arquivo e, finalmente, quando for realizar a gravação, irá gravar todos os objetos que estiverem no dicionário. Veja como ficará o código:

```
import json
with open("inventario_json.json", "r") as arq_json:
    inventario = json.load(arq_json)
opcao=int(input("Digite: \n<1> para registrar ativo"
               "\n<2> para exibir ativos armazenados: "))
while opcao>0 and opcao<3:
    if opcao==1:
        resp = "S"
        while resp == "S":
            inventario[input("Digite o número patrimonial: ")] = [
                input("Digite a data da última atualização: "),
                input("Digite a descrição: "),
                input("Digite o departamento: ")]
            resp = input("Digite <S> para continuar.").upper()
            with open("inventario_json.json", "w") as arq_json:
                json.dump(inventario, arq_json)
            print("JSON gerado!!!!")
        elif opcao==2:
            for chave, dado in inventario.items():
                print("Data.....: ", dado[0])
                print("Descrição....: ", dado[1])
                print("Departamento.: ", dado[2])
    opcao = int(input("Digite: "
                    "\n<1> para registrar ativo"
                    "\n<2> para exibir ativos armazenados: "))
```

Código-fonte 5.16 – Carregando um arquivo JSON ao abrir  
Fonte: Elaborada pelo autor (2017)

As alterações estão destacadas pela cor vermelha. Criamos o nosso dicionário “inventario”, já o preenchendo com o conteúdo do arquivo “inventario\_json.json”. Na opção “2”, de exibição, basta utilizarmos o dicionário “inventario” no “foreach”, assim percorreremos todos os dados de maneira atualizada.

Mas, agora, vamos para um pequeno problema. Elimine o arquivo “inventario\_json.json”, para isso, clique com o botão direito sobre o arquivo, escolha

a opção “Delete”, clique no botão “OK” e, no painel abaixo do PyCharm, normalmente irá aparecer um botão com o texto “Do Factor”, clique sobre ele para confirmar a exclusão. Rode novamente a aplicação e, então, verá um erro que ocorrerá porque o Python não localizou o arquivo para preencher o dicionário “inventario”.

Uma forma simples para resolver esse problema é identificar se o arquivo já existe. Para isso, precisaremos importar o pacote “os” (Operation System), que permite acessar objetos e funções referentes ao sistema operacional. Uma dessas funções é a “path.exists()”, que retorna “True”, caso encontre o arquivo especificado como parâmetro, ou “False”, se o arquivo não for encontrado. Por isso, nossa alteração será realizada somente no início do código, conforme podemos observar na sequência a seguir:

```
import json
import os
if os.path.exists("inventario_json.json"):
    with open("inventario_json.json", "r") as arq_json:
        inventario = json.load(arq_json)
else:
    inventario={}
opcao=int(input("Digite: \n<1> para registrar ativo"
               "\n<2> para exibir ativos armazenados: "))
```

Código-fonte 5.17 – Verificação da existência do arquivo  
Fonte: Elaborado pelo autor (2017)

Observe, nas linhas que estão na cor vermelha, que colocamos uma condição para a criação ou preenchimento do dicionário “inventario”, que é: se o arquivo existir, então, preencha o dicionário com o conteúdo do arquivo, caso contrário (se o arquivo não existir), crie o dicionário “inventario” vazio. Pronto!

Pronto? Podemos melhorar um último detalhe, que é o fato de modularizar, isso significa aplicar as funções. Vamos lá! Crie, no pacote “Funcoes”, o arquivo “Funcoes\_JSON.py” e digite o código seguinte:

```
import json
import os
def chamarMenu():
    escolha = int(input("Digite: "
                       "\n<1> para registrar ativo"
                       "\n<2> para exibir ativos armazenados: "))
    return escolha

def ler_arquivo(arquivo):
    if os.path.exists(arquivo):
        with open(arquivo, "r") as arq_json:
```

```

        dicionario=json.load(arq_json)
    else:
        dicionario = {}
    return dicionario

def gravar_arquivo(dicionario,arquivo):
    with open(arquivo, "w") as arq_json:
        json.dump(dicionario, arq_json)

def registrar(dicionario, arquivo):
    resp = "S"
    while resp == "S":
        dicionario[input("Digite o número patrimonial: ")] = [
            input("Digite a data da última atualização: "),
            input("Digite a descrição: "), input("Digite o departamento:
") ]
        resp = input("Digite <S> para continuar.").upper()
        gravar_arquivo(dicionario,arquivo)
    return "JSON gerado!!!!"

def exibir(arquivo):
    dicionario = ler_arquivo(arquivo)
    for chave, dado in dicionario.items():
        print("Data.....: ", dado[0])
        print("Descrição....: ", dado[1])
        print("Departamento.: ", dado[2])

```

Código-fonte 5.18 – Modularização para manipulação de arquivos JSON

Fonte: Elaborado pelo autor (2017)

Nesse arquivo, criamos algumas rotinas tão abstratas que poderiam funcionar com qualquer aplicação que tivesse a função de manipular arquivos JSON, é o caso das funções: ler\_arquivo() e gravar\_arquivo(). Para qualquer situação envolvendo arquivos JSON, elas podem ser reaproveitadas, tanto que as próprias funções registrar() e exibir() utilizam essas funções.

Agora sim, podemos pensar no nosso arquivo “Manipular\_JSON.py”, que ficará muito mais “clean”, conforme podemos perceber no código a seguir:

```

from Funcoes.Funcoes_JSON import *

inventario = ler_arquivo("inventario_json.json")
opcao=chamarMenu()
while opcao>0 and opcao<3:
    if opcao==1:
        print(registrar(inventario, "inventario_json.json"))
    elif opcao==2:
        exibir("inventario_json.json")
    opcao = chamarMenu()

```

Código-fonte 5.19 – Arquivo Manipular JSON após modularização

Fonte: Elaborado pelo autor (2017)

Nesse código, podemos perceber que até mesmo os imports foram reduzidos e que poderíamos passar qualquer nome de arquivo que as nossas funções se adaptariam perfeitamente. Agora sim... FINISH!

Muitos recursos apresentados, não é mesmo? Mas é fundamental que incorpore tudo o que foi visto aqui. Se for necessário, refaça os exemplos, pois só existe uma forma de se acostumar com a lógica, os padrões e os recursos para a manipulação de arquivos e *Strings*, que é: PRATICANDO! Por isso, é muito importante que você procure desenvolver mais exemplos e praticar sempre que possível. Isso fará toda a diferença quando tiver que converter dados em informações.

Lembre-se sempre: manipular os ARQUIVOS e as *STRINGS* lhe dará poderes extras para tarefas do dia a dia!

## 5.6 Finalizando o capítulo

Para finalizar este capítulo, você poderá aplicar todos os conceitos vistos, desenvolvendo uma ferramenta mais completa para catalogar o inventário do seu cliente. Armazene tudo em arquivo e crie, pelo menos, duas formas de saídas, por modelos, atualizações e valores, por exemplo.

Agora que você consegue armazenar os seus dados de maneira física, será direcionado, no próximo capítulo, para os ataques. Sim, isso mesmo! Você terá que realizar ataques sobre o próprio ambiente que projetou para que consiga identificar possíveis vulnerabilidades no seu ambiente. E, para isso, será fundamental conhecer os serviços mais comuns que deverão ser disponibilizados e suas respectivas fragilidades, também o fará com os ativos de rede.

Na sequência, veremos como o software e o hardware podem ser afetados por ataques via TCP/IP. Mais responsabilidades virão no próximo capítulo. Mãos na massa e atacaataaaaaarrrrrr...

## REFERÊNCIAS

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

JET BRAINS. **Download PyCharm**. 2017. Disponível em: <<https://www.jetbrains.com/pycharm/download/#section=windows>>. Acesso em: 10 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.

RAMEZ, Elmasri; SHAMKANT, B. Navathe. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson Addison Wesley, 2011.

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.