Arquivos e Tratamento de Exceções

Prof. André Backes | @progdescomplicada

Arquivos

Arquivos

- Por que usar arquivos?
  - Permitem armazenar grande quantidade de informação;
  - Persistência dos dados (disco);
  - Acesso aos dados poder ser não sequencial;
  - Acesso concorrente aos dados (mais de um programa pode usar os dados ao mesmo tempo).

Tipos de Arquivos

- Basicamente, a linguagem Python trabalha com dois tipos de arquivos
  - arquivo texto
  - arquivo binário

Tipos de Arquivos

Arquivo Texto

- Arquivo texto
  - armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de textos simples como o Bloco de Notas.
  - Os dados são gravados como caracteres de 8 bits.
  - Exemplo: Um número inteiro de 32 bits com 8 dígitos ocupará 64 bits no arquivo (8 bits por dígito).

Arquivo Binário

- Arquivo binário
  - armazena uma sequência de bits que está sujeita as convenções dos programas que o gerou.
    - Ex: arquivos executáveis, arquivos compactados, arquivos de registros, etc.
  - os dados são gravados na forma binária (do mesmo modo que estão na memória).
  - Exemplo: um número inteiro de 32 bits com 8 dígitos ocupará 32 bits no arquivo.

Tipos de Arquivos

- Os dois trechos de arquivo abaixo possuem os mesmo dados

  nome = “Ricardo”;

  i = 30;

  a = 1.74;

Manipulando arquivos

- A linguagem Python possui uma série de funções para manipulação de arquivos
  - Suas funções se limitam a abrir/fechar e ler caracteres/bytes/linhas do arquivo
  - É tarefa do programador criar a função que lerá um arquivo de uma maneira específica
- Para manipularmos um arquivo, vamos precisar apenas de um objeto do tipo **file**

```
>>> f = open("teste.txt","w")
>>> print(f)
<_io.TextIOWrapper name='teste.txt' mode='w' encoding='cp1252'>
```

Abrindo um arquivo

- Para a abertura de um arquivo, usa-se a função **open()**

objeto-file = open(nome-arquivo,modo-abertura)

- O parâmetro **nome-arquivo** determina qual arquivo deverá ser aberto, sendo que o mesmo deve ser válido no sistema operacional que estiver sendo utilizado

Abrindo um arquivo

- No parâmetro nome-arquivo pode-se trabalhar com caminhos absolutos ou relativos
  - Caminho absoluto: descrição de um caminho desde o diretório raiz
    - C:\\Projetos\\dados.txt
  - Caminho relativo: descrição de um caminho desde o diretório corrente
    - arq.txt
    - ..\\dados.txt

Abrindo um arquivo

- O modo de abertura determina que tipo de uso será feito do arquivo
- A tabela a seguir mostra os modo válidos de abertura de um arquivo

Abrindo um arquivo

Modo	*Arquivo*	*Função*
"r"	Texto	Leitura. Arquivo deve existir.
"w"	Texto	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"rb"	Binário	Leitura. Arquivo deve existir.
"wb"	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab"	Binário	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+"	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+"	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+"	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").
"rb+"	Binário	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"wb+"	Binário	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab+"	Binário	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").

Abrindo um arquivo

- Um arquivo texto pode ser aberto para escrita utilizando o seguinte conjunto de comandos
- Neste caso, o arquivo **texte.txt** será aberto.
- Agora já podemos escrever dados ou fechá-lo

```
f = open("teste.txt","w")

f.write("Teste de escrita")

f.close()
```

Fechando um arquivo

- Como visto no slide anterior, sempre que terminamos de usar um arquivo que foi aberto, devemos fechá-lo. Para isso usa-se a função close()

objeto-file.close()

- Mas por que precisamos fechar o arquivo?

Fechando um arquivo

- Por que precisamos fechar o arquivo?
  - Ao fechar um arquivo, todo dado que tenha permanecido no "buffer" é gravado
  - O "buffer“ é uma região de memória que armazena temporariamente os dados a serem gravados em disco
  - Apenas quando o "buffer" está cheio é que seu conteúdo é escrito no disco

Fechando um arquivo

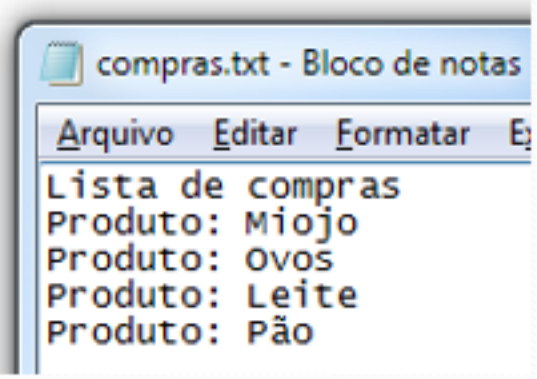
- Mas por que utilizar um "buffer"?? Eficiência!
  - Para ler e escrever arquivos no disco temos que posicionar a cabeça de gravação em um ponto específico do disco
  - Se tivéssemos que fazer isso para cada caractere lido/escrito, a leitura/escrita de um arquivo seria uma operação muita lenta
  - Assim a gravação só é realizada quando há um volume razoável de informações a serem gravadas ou quando o arquivo for fechado

Fechando um arquivo

Com close()

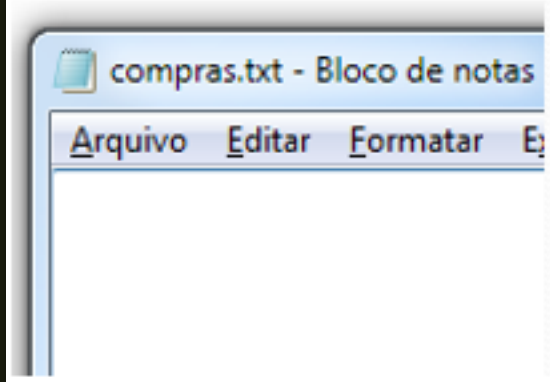
```
f = open("compras.txt","w")
compras = ["Miojo","Ovos","Leite","Pão"]
f.write("Lista de compras\n")
for item in compras:
    f.write("Produto: "+item+"\n")

f.close()
```



Sem close()

```
f = open("compras.txt","w")
compras = ["Miojo","Ovos","Leite","Pão"]
f.write("Lista de compras\n")
for item in compras:
    f.write("Produto: "+item+"\n")
```



Escrita/Leitura em Arquivos

- Uma vez aberto um arquivo, podemos ler ou escrever nele
- Para tanto, a linguagem Python conta com uma série de funções de escrita/leitura que variam de funcionalidade para atender as diversas aplicações
  - Todas as funções que serão vistas trabalham com a escrita/leitura de **texto**
  - Cabe ao programador tratar os dados que escritos/lidos do artigo

Posição do arquivo

- Ao se trabalhar com arquivos, existe uma espécie de posição onde estamos dentro do arquivo
- É nessa posição onde será lido ou escrito o próximo caractere ou cadeia de caracteres
- Quando utilizando o acesso sequencial, raramente é necessário modificar essa posição
  - Após cada leitura, a posição no arquivo é automaticamente atualizada para a próxima posição a ser lida
  - Após cada escrita, a posição no arquivo é automaticamente atualizada para a próxima posição a ser escrita

Escrita em Arquivos | write()

- Para escrever em um arquivo usa-se a função write(). Forma geral:

objeto-file.write(string)

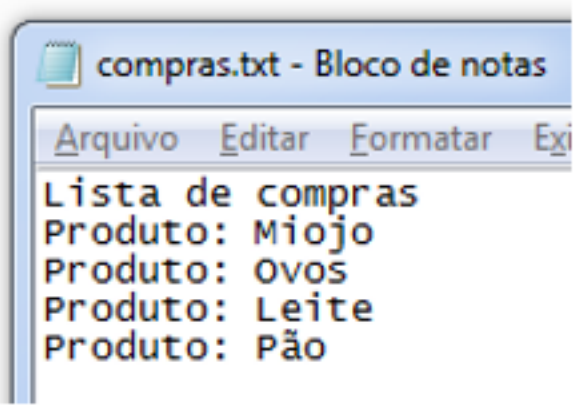
- Esta função recebe como parâmetro uma cadeia de caracteres (string) que será escrita no arquivo aberto especificado por **objeto-file**

Escrita em Arquivos | write()

- A função **write**() permite gravar strings contendo qualquer tipo de dado, inclusive dados formatados e sequências de escape
  - É tarefa do programador colocar os dados a serem gravados dentro da string

```
f = open("compras.txt","w")
compras = ["Miojo","Ovos","Leite","Pão"]
f.write("Lista de compras\n")
for item in compras:
    f.write("Produto: "+item+"\n")

f.close()
```



Escrita em Arquivos | writelines()

- O método **writelines**() é usado para escrever múltiplas strings de uma única vez em um arquivo. Forma geral:

objeto-file.writelines(lista_de_strings)

- Esta função recebe como parâmetro uma lista contendo vários objetos **string** que será escrita no arquivo aberto especificado por **objeto-file**

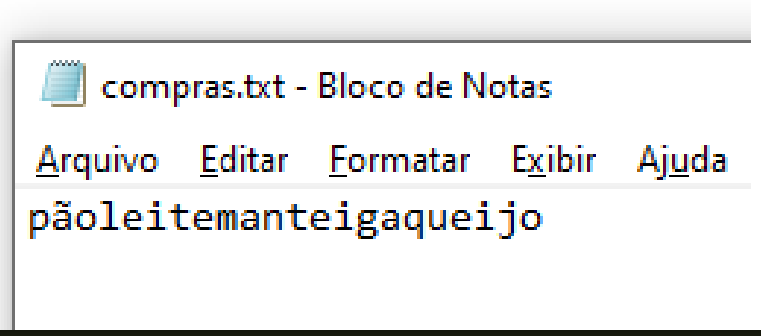
Escrita em Arquivos | writelines()

- A função **writelines**() permite gravar um conjunto de strings definidas por uma lista de maneira fácil.
- Porém, note que nenhum tipo de separador é colocado entre as strings
  - É tarefa do programador formatar os dados a serem gravados dentro da lista de strings

```
f = open('compras.txt','w')
compras = ['pão','leite','manteiga','queijo']

f.writelines(compras)

f.close()
```



Leitura em Arquivos | read()

- A função **read**() lê todo o conteúdo de um arquivo aberto e armazena em uma **string**
- Os dados serão lidos exatamente como estão salvos no arquivo, inclusive com as quebras de linhas.

```
f = open("compras.txt","r")
s = f.read()
print(s)
f.close()


>>>
Lista de compras
Produto: Miojo
Produto: Ovos
Produto: Leite
Produto: Pão

>>>
```

Leitura em Arquivos | read(N)

- A função **read(N)** lê apenas os **N** próximos caracteres de um arquivo aberto e armazena em uma **string**
- Como na função **read()**, os dados serão lidos exatamente como estão salvos no arquivo, inclusive com as quebras de linhas.

```
f = open("compras.txt","r")
s = f.read(8)
print(s)
f.close()


>>>
Lista de
>>>
```

Leitura em Arquivos | readline()

- A função **readline**() lê apenas uma linha de texto de um arquivo aberto e armazena em uma **string**
- Essa função lê da posição atual no arquivo até encontrar um quebra de linha
  - **Atenção**: O caractere de quebra de linha (**\n**) será armazenado dentro da string

```
f = open("compras.txt","r")
s = f.readline()
print(s)
f.close()


>>>
Lista de compras

>>>
```

Leitura em Arquivos | readlines()

- A função **readlines**() lê todo o conteúdo de um arquivo aberto e armazena em uma **lista de strings**
- **Atenção**: O caractere de quebra de linha (**\n**) separa uma string da outra e será armazenado dentro da string

```
f = open("compras.txt","r")
s = f.readlines()
print(s)
f.close()

>>>
['Lista de compras\n', 'Produto: Miojo\n', 'Produto: Ovos\n', 'Produto: Leite\n'
, 'Produto: Pão\n']
>>>
```

Lendo um arquivo até o final

- Diferente de outras linguagens, Python não possui um teste de **fim de arquivo** (**EOF**, *end of file*)
  - Em Python, a indicação de final de arquivo se dá pela ausência de bytes retornada por uma função de leitura
- Assim, a melhor maneira de se verificar se chegamos ao final do arquivo é testar se algo foi retornado pela função de leitura

Lendo um arquivo até o final

- O teste de final de arquivo pode ser facilmente implementado com uma construção do tipo

while True / break

- Basicamente, criamos um laço infinito (**while True**) e sempre que a leitura retorna nada interrompemos o laço (**break**)

```
f = open("compras.txt","r")
while True:
    linha = f.readline()
    if linha == "":
        break
    print(linha)

f.close()


>>>
Lista de compras

Produto: Miojo

Produto: Ovos

Produto: Leite

Produto: Pão
```

Tratamento de Exceções

Tratamento de erros e exceções

- Nenhum sistema é perfeito. Todo sistema está sujeito a ações que causam anomalias
  - Divisão por zero, raiz quadrada de um número negativo, abrir um arquivo que não existe
- Em algum momento qualquer sistema terá que tratar uma exceção, e é melhor ele estar preparado para isso
  - Ou o usuário terá seu sistema encerrado de forma inesperada
  - Ou receberá uma mensagem incompreensível

Instruções try-except

- Uma forma de apanhar os erros e exceções que podem ocorrer no nosso programa Python é usar o comando **try-except**
- Com ele, podemos
  - Tratar qualquer exceção
  - Tratar uma determinada exceção

```
try:
    instrução 1
    instrução 2
    ...
    instrução n
except:
    instrução 1
    instrução 2
    ...
    instrução n
```

```
try:
    instrução 1
    instrução 2
    ...
    instrução n
except Exceção:
    instrução 1
    instrução 2
    ...
    instrução n
```

Instruções try-except

- Um bloco **try** delimita um segmento do programa onde alguma coisa errada pode acontecer
- Já o bloco **except** contém o código que será executado se a exceção ocorrer

```
import math

try:
    x = math.sqrt(-1)
    print("Raiz = ",x)
except:
    print("Erro no cálculo")
```

Instruções try-except

- Um bloco **try** pode possuir mais de um bloco **except**
- Nesse caso, cada bloco **except** pode tratar um tipo de exceção específica

```
try:
    x = int(input("Digite x: "))
    y = 10 / x
    print("y = ",y)
except ValueError:
    print("Valor digitado não é inteiro")
except ZeroDivisionError:
    print("Valor não pode ser zero")
```

Instruções try-except

- Algumas exceções já pré-definidas
  - **IOError**: Erros de leitura/escrita de arquivos
  - **ValueError**: Parâmetros fora do domínio (exemplo, sqrt(-1))
  - **IndexError**: Índice fora de limites
  - **TypeError**: Erro de tipos
  - **KeyError**: Chave não encontrada no dicionário
  - **NameError**: Nome de variável não encontrado
  - **RecursionError**: Foi alcançada a profundidade máxima da recursão
  - **IndentationError**: indentação incorreta detectada

Instruções try-finally

- A instrução **try-finally** é similar a **try-except**
  - A instrução **finally** sempre é executada
  - O código que estiver no bloco **finally** sempre será executado, independentemente se ocorre ou não uma exceção

```
try:
    instrução 1
    instrução 2
    ...
    instrução n
finally:
    instrução 1
    instrução 2
    ...
    instrução n
```

Instruções try-finally

- A instrução **finally** é muito útil quando queremos liberar algum recurso utilizado, como fechar arquivo
- Assim, mesmo que dê erro na leitura do arquivo, o mesmo será fechado

```
f = None
try:
    f = open("compras.txt","r")
    total = 0
    while True:
        linha = f.readline()
        if linha == "":
            break
        itens = linha.split()
        total = total + int(itens[1]) * float(itens[2])

    print("Total da compra: ",total)
finally:
    print("Fechando o arquivo")
    f.close() #arquivo deve ter sido aberto com sucesso
```

Instruções try-finally

- Pode-se ainda usar as instruções **try**, **except** e **finally** em conjunto

```
f = None
try:
    f = open("compras.txt","r")
    total = 0
    while True:
        linha = f.readline()
        if linha == "":
            break
        itens = linha.split()
        total = total + int(itens[1]) * float(itens[2])

    print("Total da compra: ",total)
except:
    print("Problema na leitura do arquivo")
finally:
    print("Fechando o arquivo")
    f.close() #arquivo deve ter sido aberto com sucesso
```

Material Complementar

- Vídeo Aulas
  - Aula 34 - Arquivos: definição
  - https://youtu.be/1ZFe-OqMB28
  - Aula 35 - Arquivos: abrindo e fechando
  - https://youtu.be/vtc5P6V_8t8
  - Aula 36 - Arquivos: leitura e escrita
  - https://youtu.be/VscBZSm4K10
  - Aula 37 - Tratamento de erros e exceções
  - https://youtu.be/FxCEHvk3Sjl