Fundamentos de programação para SI

Prof. MSc. Anderson Ávila

Abril/19

Manipulando Arquivos

- Em Python, devemos abrir (open) arquivos antes de usá-los e fechar (close) os arquivos depois de que tivermos terminado de utilizá-los.
- Depois de aberto um arquivo passa a ser um objeto Python de maneira semelhante que outros dados.
- A seguir os métodos que podem ser usados para abrir e fechar arquivos.

Manipulando Arquivos

r	Abre o arquivo de texto para leitura. O stream (fluxo de entrada ou saída) é posicionado no início do arquivo.
r+	Abre para leitura e escrita. O fluxo é posicionado no início do arquivo.
w	Trunca o arquivo para zero ou cria um arquivo de texto para escrita. O stream é posicionado no início do arquivo.
w+	Abre para leitura e escrita. O arquivo é criado se ele não existir, caso contrário será sobreescrito. O stream é posicionado no início do arquivo.
a	Abre para escrita. O arquivo é criado caso não exista. O stream é posicionado no final do arquivo. Gravações subsequentes do arquivo sempre vão acabar no fim do arquivo atual.
a+	Abre para leitura e escrita. O arquivo é criado se ele não existir. O stream é posicionado no final do arquivo. Gravações subsequentes no arquivo sempre vão acabar no fim do arquivo atual.

Manipulando Arquivos

- Crie um arquivo chamado data dentro desse diretório crie o arquivo de texto saudacao.txt:
-
-

```
Olá Mundo!
Eu adoro programar em Python.
```

- Usando o método read()
  - Para abrir um arquivo somente para leitura em um programa Python usamos a função open() tendo como argumento o nome e o caminho até arquivo, o caminho pode ser absoluto ou relativo, e como opcional um segundo argumento o caractere “r”.

Manipulando Arquivos

- Exemplo de leitura simples:

```
arq = open('saudacao.txt', 'r')
saudacao = arq.read()
arq.close()
print(saudacao)
```

- Lemos o seu conteúdo com o método read() que coloca todo o conteúdo do arquivo em uma string **única.**
- Depois disso o arquivo é fechado uma vez que já temos os dados dele para trabalhar.

Manipulando Arquivos

- Pode-se utilizar o método split() para strings e criar uma lista de strings:

```
arq = open('saudacao.txt', 'r')
saudacao = arq.read()
arq.close()
print(saudacao)
lista_saudacao = saudacao.split('\n')
print(lista_saudacao)
```

Encontrando um arquivo disco

- A maneira que os arquivos são localizados no disco e através do seu caminho (path).
- No Linux caminho poderia ser /home/anderson/texto.txt
- No sistema operacional Windows o caminho parece um pouco diferente, mas os princípios são os mesmos. Por exemplo no Windows o caminho poderia ser C:\Users\anderson\Meus Documentos\texto.txt.

Encontrando um arquivo disco

- Aqui está uma regra importante para ser lembrada: se o seu arquivo de dados e o seu programa Python estão no mesmo diretório você pode usar simplesmente o nome do arquivo. open('texto.txt','r') .
- Se o seu arquivo de dados e o seu programa Python estão em diretórios diferentes então você deve usar o caminho até o arquivo open('/Users/anderson/texto.txt','r').

Iterando sobre as linha de um Arquivo

- Suponha que temos um texto chamado qbdata.txt que contem os dados a seguir representando estatísticas sobre o quarterbacks da NFL.
- O formato de arquivo de dados é o seguinte:

```
First Name, Last Name, Position, Team, Completions, Attempts, Yards, TDs Ints, Comp%, Rating
```

Iterando sobre as linha de um Arquivo

- Usaremos o arquivo qbdata.txt como entrada de um programa que faz um pouco de processamento de dados.
- No programa nós iremos ler (read) cada linha do arquivo e imprimi-la com algum texto adicional.
- Como arquivos de texto são uma sequência de linhas com texto, nós usaremos um laço for para iterar sobre cada linha do arquivo.

Iterando sobre as linha de um Arquivo

- A medida com o laço for itera sobre cada linha do arquivo a variável de controle do laço conterá uma referência para um string com o conteúdo da linha corrente do arquivo.
- O padrão geral para processar cada linha de um arquivo texto é o seguinte:
-

```
for linha in ref_arquivo:
    comando1
    comando2
    ...
```

Iterando sobre as linha de um Arquivo

- Para processar todos os dados sobre os quarterbacks, usamos o laço for para iterar sobre as linhas do arquivo.
- Usando o método split podemos quebrar cada linha em uma lista contendo todos os campos de interesse sobre o quarterback.
-

```
ref_arquivo = open("Aula 2/qbdata.txt","r")

for linha in ref_arquivo:
    valores = linha.split()
    print('QB ', valores[0], valores[1],
    'obteve a avaliacao ', valores[10] )

ref_arquivo.close()
```

Métodos alternativos para ler arquivos

- Para processar todos os dados sobre os quarterbacks, usamos o laço for para iterar sobre as linhas do arquivo.
- Usando o método split podemos quebrar cada linha em uma lista contendo todos os campos de interesse sobre o quarterback.

Métodos alternativos para ler arquivos

Além do laço for Python fornece três métodos para lermos dados de um arquivo.
O método **readline()** lê uma linha de um arquivo e retorna essa linha como um string. O string retornado por readline conterá o caractere de nova linha (‘\n’) no final. Esse método retorna vazio quando chega ao final do arquivo.
O método **readlines()** retorna o todo o conteúdo do arquivo em uma lista de strings, cada item da lista representa uma linha do arquivo.
Também é possível ler todo o conteúdo de um arquivo em um único string como o método **read().**

Exemplos

```
1   ref_arquivo = open("Aula 2/qbdata.txt","r")
2   linha = ref_arquivo.readline()
3   print(linha)
4
5   ref_arquivo = open("Aula 2/qbdata.txt","r")
6   lista_de_linhas = ref_arquivo.readlines()
7   print(len(lista_de_linhas))
8
9   print(lista_de_linhas[0:4])
10
11
12  ref_arquivo = open("Aula 2/qbdata.txt","r")
13  string_arquivo = ref_arquivo.read()
14  print(len(string_arquivo))
```

Leitura com while

```
1  ref_arquivo = open("Aula 2/qbdata.txt","r")
2  linha = ref_arquivo.readline()
3  while linha:
4      valores = linha.split()
5      print('QB ', valores[0], valores[1], 'obteve a avaliacao ', valores[10] )
6      linha = ref_arquivo.readline()
7
8  ref_arquivo.close()
```

Escrevendo em um arquivo

Caso você queira adicionar (escrever) outra frase ou parágrafo no arquivo que já lemos.
Digamos que queria adicionar a seguinte frase “Fim da lista” no arquivo. Isso pode ser feito em Python usando o método write().

Exemplo

```
1  with open('Aula 2/Text.txt', 'r+') as text_file:
2      print ('The file content BEFORE writing content:')
3      print (text_file.read())
4      text_file.write(' and I\'m looking for more')
5      print ('The file content AFTER writing content:')
6      text_file.seek(0)
7      print (text_file.read())
```

- O with ... as ...nos permite abrir o arquivo, processá-lo e certificar-se de que está fechado.
- O seek() por outro lado, nos permite mover o ponteiro (ou seja, o cursor) para alguma outra parte do arquivo.

Exemplo

```
1   # Abra o arquivo (leitura)
2   arquivo = open('Aula 2/Text.txt', 'r')
3   conteudo = arquivo.readlines()
4
5   # insira seu conteúdo
6   # obs: o método append() é proveniente de uma lista
7   conteudo.append('Nova linha')
8
9   # Abre novamente o arquivo (escrita)
10  # e escreva o conteúdo criado anteriormente nele.
11  arquivo = open('Aula 2/Text.txt', 'w')
12  arquivo.writelines(conteudo)
13  arquivo.close()
14
```

Exercícios

- Usando o arquivo texto **notas_estudantes.dat** escreva um programa que imprime o nome dos alunos que têm mais de seis notas.
- Usando o arquivo texto **notas_estudantes.dat** escreva um programa que calcula a nota mínima e máxima de cada estudante e imprima o nome de cada aluno junto com a suas notas máxima e mínima.
- Usando o arquivo de texto **notas_estudantes.dat** escreva um programa que calcula a nota mínima e máxima de cada estudante e imprima o nome de cada aluno junto com a suam notá máxima e mínima.

Exercícios

Faça um programa que leia um arquivo texto contendo uma lista de endereços IP e gere um outro arquivo, contendo um relatório dos endereços IP válidos e inválidos.O arquivo de entrada possui o seguinte formato:

```
200.135.80.9
192.168.1.1
8.35.67.74
257.32.4.5
85.345.1.2
1.2.3.4
9.8.234.5
192.168.0.256
```

O arquivo de saída possui o seguinte formato:

```
[Endereços válidos:]
200.135.80.9
192.168.1.1
8.35.67.74
1.2.3.4

[Endereços inválidos:]
257.32.4.5
85.345.1.2
9.8.234.5
192.168.0.256
```

Exercícios FAB II

A FAB II Inc., uma empresa de 500 funcionários, está tendo problemas de espaço em disco no seu servidor de arquivos. Para tentar resolver este problema, o Administrador de Rede precisa saber qual o espaço ocupado pelos usuários, e identificar os usuários com maior espaço ocupado. Através de um programa, baixado da Internet, ele conseguiu gerar o seguinte arquivo, chamado "usuarios.txt":

```
alexandre       456123789
anderson        1245698456
antonio         123456456
carlos          91257581
cesar           987458
rosemary        789456125
```

Exercício FAB II

Neste arquivo, o nome do usuário possui 15 caracteres. A partir deste arquivo, você deve criar um programa que gere um relatório, chamado "relatório.txt", no seguinte formato:

```
ACME Inc.               Uso do espaço em disco pelos usuários
------------------------------------------------------------------------
Nr.  Usuário        Espaço utilizado     % do uso

1    alexandre       434,99 MB             16,85%
2    anderson       1187,99 MB             46,02%
3    antonio         117,73 MB              4,56%
4    carlos           87,03 MB              3,37%
5    cesar             0,94 MB              0,04%
6    rosemary        752,88 MB             29,16%
```

O arquivo de entrada deve ser lido uma única vez, e os dados armazenados em memória, caso sejam necessários, de forma a agilizar a execução do programa. A conversão da espaço ocupado em disco, de bytes para megabytes deverá ser feita através de uma função separada, que será chamada pelo programa principal. O cálculo do percentual de uso também deverá ser feito através de uma função, que será chamada pelo programa principal.

Exercício Log Squid

- Analisador de logs do Squid: sites bloqueados. Desenvolva um analisador de log do Squid que mostre quais os sites mais bloqueados em uma organização.
- Explicação sobre o log:
- http://aguiadeti.blogspot.com.br/2014/06/entendendo-log-do-squid.html?m=1

Passando parâmetros via shell/cmd

- Para isso vamos usar o módulo “**sys**” do Python.
- Este módulo nos permite usar algumas informações do sistema.
- O que buscamos fazer é o seguinte: por exemplo, chamar um programa dessa forma, no terminal:

```
python3 programa.py arquivo_entrada.txt arquivo_saida.txt
```

- Neste exemplo estaríamos passando dois parâmetros para o programa, um arquivo de entrada e um arquivo de saída, ambos arquivos de texto.

Passando parâmetros via shell/cmd

- Para fazer nossos programas entenderem esses parâmetros vamos utilizar a lista argv, do módulo sys, assim:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import sys
for param in sys.argv :
    print(param)
```

Tratamento de exceções

- É possível escrever programas que tratam exceções específicas.
- Observe o exemplo seguinte, que pede dados ao usuário até que um inteiro válido seja fornecido

```
while True:
    try:
        x = int(input("Por favor, informe um número: "))
        break
    except ValueError:
        print("Oops!  Não foi um número válido.  Tente novamente...")

```

Tratamento de exceções

```
1   import sys
2
3   try:
4       f = open('meuarquivo.txt')
5       s = f.readline()
6       i = int(s.strip())
7   except IOError as (errno, strerror):
8       print("I/O error({0}): {1}".format(errno, strerror))
9   except ValueError:
10      print("Não foi possível converter o dado para inteiro.")
11  except:
12      print("Erro inesperado:", sys.exc_info()[0])
13      raise
```

Tratamento de exceções

- O try...except possui uma *cláusula else* opcional, que quando presente, deve ser colocada depois de todas as outras cláusulas. É útil para um código que precisa ser executado se nenhuma exceção foi levantada.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('não foi possível abrir', arg)
    else:
        print (arg, 'tem', len(f.readlines()), 'linhas')
        f.close()
```

Definindo ações de limpeza

- A instrução **try** possui outra cláusula opcional, cuja finalidade é permitir a implementação de ações de limpeza, que sempre devem ser executadas independentemente da ocorrência de exceções.

```
def divide(x, y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print("divisão por zero!")
    else:
        print("resultado é", resultado)
    finally:
        print("executando a cláusula finally")
```

Orientação a Objetos

- O desenvolvimento de aplicações de software estão cada vez mais complexas;
- Cresceram as demandas por metodologias que pudessem abstrair e modularizar as estruturas básicas de programas; e
- A maioria das linguagens de programação suportam orientação a objetos: Haskell, Java, C++, Python, PHP, Ruby, Pascal, entre outras.

História

- Em 1967, Kristen Nygaard e Ole-Johan Dahl, do Centro Norueguês de Computação em Oslo, desenvolveram a linguagem Simula 67 que introduzia os primeiros conceitos de orientação a objetos;
- Em 1970, Alan Kay, Dan Ingalls e Adele Goldberg, do Centro de Pesquisa da Xerox, desenvolveram a linguagem totalmente orientada a objetos;
- Em 1979–1983, Bjarne Stroustrup, no laboratório da AT & T, desenvolveu a linguagem de programação C++, uma evolução da linguagem C; e
- Maior divulgação a partir de 1986 no primeiro workshop “Object-Oriented Programming Languages, Systems and Applications”

Principais Vantagens

- Aumento de produtividade;
- Reuso de código;
- Redução das linhas de código programadas;
- Separação de responsabilidades;
- Componentização;
- Maior flexibilidade do sistema; e
- Facilidade na manutenção.

Objetos

- É a metáfora para se compreender a tecnologia orientada a objetos;
- Estamos rodeados por objetos: mesa, carro, livro, pessoa, etc; e
- Os objetos do mundo real têm duas características em comum:

1. Estado – representa as propriedades (nome, peso, altura, cor, etc.); e
2. Comportamento – representa ações (andar, falar, calcular, etc.).

Definição

- É um paradigma para o desenvolvimento de software que basea-se na utilização de componentes individuais (objetos) que colaboram para construir sistemas mais complexos.
- A colaboração entre os objetos é feita através do envio de mensagens;
- Descreve uma série de técnicas para estruturar soluções para problemas computacionais; e
- É um paradigma de programação no qual um programa é estruturado em objetos

Os Quatros Pilares

1. Abstração;
2. Encapsulamento
3. Herança; e
4. Polimorfismo

Os Quatros Pilares

1. Abstração;
2. Encapsulamento
3. Herança; e
4. Polimorfismo

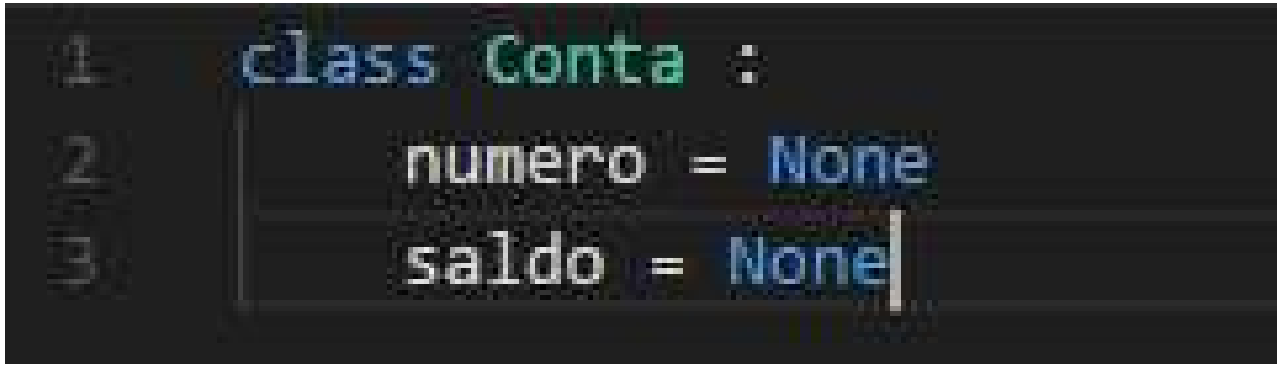
Abstração

1. A estrutura fundamental para definir novos objetos é a classe; e
2. Uma classe é definida em código-fonte.

Classe em Python

```
class nome_da_classe:
    atributos
    construtor
    métodos
```

Demonstração de Classe



Instância

- Uma instância é um objeto criado com base em uma classe definida;
- Classe é apenas uma estrutura, que especifica objetos, mas que não pode ser utilizada diretamente;
- Instância representa o objeto concretizado a partir de uma classe;
- Uma instância possui um ciclo de vida:
  - Criada;
  - Manipulada; e
  - Destruída.

Estrutura
variável = **Classe()**

Demonstração de Instância

```
conta = Conta ()
conta.numero = 1
conta.saldo = 10
print (conta.numero)
print (conta.saldo)
```

Construtor

- Determina que ações devem ser executadas quando da criação de um objeto; e
- Pode possuir ou não parâmetros.

Estrutura

```
def __init__(self,parâmetros):
```

Construtor

- Determina que ações devem ser executadas quando da criação de um objeto; e
- Pode possuir ou não parâmetros.

```
1   class Conta :
2       def __init__ (self , numero ):
3           self.numero = numero
4           self.saldo = 0.0
5
6   conta = Conta(11)
7   print(conta.numero)
8   print(conta.saldo)
```

Métodos

- Representam os comportamentos de uma classe;
- Permitem que acessemos os atributos, tanto para recuperar os valores, como para alterá-los caso necessário;
- Podem retornam ou não algum valor; e
- Podem possuir ou não parâmetros.

Estrutura

```
def nome_do_método(self,parâmetros):
```

Importante

O parâmetro **self** é obrigatório.

Métodos

```python
class Conta :
    def __init__ (self , numero ):
        self.numero = numero
        self.saldo = 0.0
    def consultar_saldo ( self ):
        return self.saldo

    def creditar (self , valor ):
        self.saldo += valor

    def debitar (self , valor ):
    self.saldo -= valor

    def transferir (self , conta , valor ):
        self.saldo -= valor
        conta.saldo += valor

conta1 = Conta (1)
conta1.creditar (10)
conta2 = Conta (2)
conta2.creditar (5)
print(conta1.consultar_saldo())
print( conta2 . consultar_saldo())
conta1.transferir ( conta2 ,5)
print(conta1.consultar_saldo())
print(conta2.consultar_saldo())
```

Encapsulamento

```python
class Conta :
    def __init__ (self , numero ):
        self.__numero = numero
        self.__saldo = 0.0
    def consultar_saldo ( self ):
        return self.__saldo

    def creditar (self , valor ):
        self.__saldo += valor

    def debitar (self , valor ):
        self.__saldo -= valor

    def transferir (self , conta , valor ):
        self.__saldo -= valor
        conta.creditar(valor)

conta1 = Conta (1)
conta1.creditar (10)
conta2 = Conta (2)
conta2.creditar (5)
print(conta1.consultar_saldo())
print( conta2.consultar_saldo())
conta1.transferir( conta2 ,5)
print(conta1.consultar_saldo())
print(conta2.consultar_saldo())
```

Herança

- É uma forma de abstração utilizada na orientação a objetos;
- Pode ser vista como um nível de abstração acima da encontrada entre classes e objetos;
- Na herança, classes semelhantes são agrupadas em hierarquias;
- Cada nível de uma hierarquia pode ser visto como um nível de abstração;

Herança

- Cada classe em um nível da hierarquia herda as características das classes nos níveis acima;
- É uma forma simples de promover reuso através de uma generalização;
- Facilita o compartilhamento de comportamento comum entre um conjunto de classes semelhantes; e
- As diferenças ou variações de uma classe em particular podem ser organizadas de forma mais clara

Herança

Estrutura

```
class nome_da_classe(classe_pai_1, classe_pai_2, classe_pai_n):
    atributos
    métodos
```

Herança exemplo

```python
from conta import Conta

class Poupanca ( Conta ):
    def __init__ (self , numero ):
        super().__init__ ( numero )
        self.__rendimento = 0.0
    def consultar_rendimento ( self ):
        return self.__rendimento
    def gerar_rendimento (self , taxa ):
        self.__rendimento += super().consultar_saldo () * taxa / 100


poupanca = Poupanca(12)
poupanca.creditar(200)
print(poupanca.consultar_rendimento())
```

Polimorfismo

- É originário do grego e significa “muitas formas” (poli = muitas, morphos = formas);
- Indica a capacidade de abstrair várias implementações diferentes em uma única interface;
- É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos; e
- Quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução.

Polimorfismo

```
class Poupanca(Conta):
    def __init__(self,numero):
        super().__init__(numero)
        self.__rendimento = 0.0

    def consultar_rendimento(self):
        return self.__rendimento

    def gerar_rendimento(self,taxa):
        self.__rendimento += super().consultar_saldo() * taxa / 100

    def consultar_saldo(self):
        return super().consultar_saldo() + self.__rendimento

conta = Poupanca(1)
conta.creditar(200.0)
conta.gerar_rendimento(5)
print(conta.consultar_saldo())
```

Exercícios

- Exercícios de orientação a objetos

Requisições Web

Dúvidas e mais informações

Prof. Anderson Ávila
anderson.avilasantos@gmail.com