Introdução à Programação

Aula07 – Técnicas de Programação

Eduardo Corrêa Gonçalves

02/04/2023

Sumário

Strings – Sequências de Escape

- **Sequências de Escape**
  - A barra invertida “ **** ” dentro de uma string é chamada de **caractere de escape** e possui uma função especial.
    - Ela permite com que seja especificados caracteres especiais como por exemplo nova linha (**\n**) e tabulação (**\t**). Veja alguns exemplos abaixo:

- \' → aspas simples
- \" → aspas duplas
- \\ → barra invertida
- \n → nova linha
- \t → tabulação
- \r → retorno de carro

```
>>> s1 = "samba\nbossa\tnova"
>>>
>>> print(s)
samba
bossa   nova

"""
para imprimir uma barra invertida tempos que escrever duas, já que ela é usada como caractere de escape
"""
>>> s2 = 'barra invertida: \\'
>>> print(s2)
barra invertida: \
```

Strings – Interpolação (1/3)

- **Interpolação**
  - O processo de interpolação de strings consiste em inserir valores de variáveis dentro de **marcações especiais** (denominadas *placeholders*) especificadas em uma string.
  - A interpolação simplifica bastante a definição de strings em situações onde é necessário misturar texto fixo (mensagens) com valores de variáveis.
  - No Python há algumas formas diferentes de interpolar. Nesse curso, adotaremos o uso de **f-strings**.
    - São definidas com o uso da letra **f** (ou **F**) precedendo a string.
    - Para interpolar, utiliza-se a sintaxe {valor:formatação} dentro da string (*a formatação é opcional*).
    - No slide a seguir mostramos um exemplo.

Strings – Interpolação (2/3)

- **f-strings – formatted string literals (1/2)**

```
# imprime conteúdo das variáveis “nome” e “idade”, sem formatação
>>> nome = "Hal"; idade = 55;
>>> print(f"Meu nome é {nome}, tenho {idade} anos.")
Meu nome é Hal, tenho 55 anos.

# imprime conteúdo da variável “valor”. Formatação: 2 casas decimais
>>> valor = 9500.3366
>>> print(f"valor com 2 decimais: {valor: .2f}")
valor com 2 decimais:  9500.34

 # imprime conteúdo da variável “t”. Formatação: centralizado usando 30 posições
>>> t = "2001: uma Odisséia"
>>> print(f"| {t:^30} |")
|       2001: uma Odisséia       |
```

Strings – Interpolação (3/3)

- **f-strings – formatted string literals (2/2)**
  - Algumas opções de formatação:

  - **Alinhamento**
    - *n*<, *n*> e *n*^ → força expressão a ser alinhada à esquerda (<), direita (>) ou centralizada (^) usando *n* posições na string.
  - **Tipo de Dados**
    - s → string (default)
    - d → inteiro
    - e → notação científica
    - f → float (real) com precisão (número de casas decimais) padrão = 6
    - % → porcentagem. Multiplica o número por 100 e exibe com o formato f
  - **Casas decimais**
    - *n*.*w*f → exibe número real com *w* casas decimais (corretamente arredondado), reservando *n* posições (opcional).

  - Podem ser **mescladas**. Por exemplo: {v:>5.2f} formata o valor da variável v alinhado à direita, usando 5 posições dentro da string, com 2 casas decimais.

Parâmetros da função print() (1/2)

- A função **print()** possui dois **parâmetros opcionais** descritos a seguir:
  - **sep**:
    - Quando usamos o **print()** para escrever várias informações, por padrão elas serão apresentadas separadas por espaço.
    - Com o uso do parâmetro **sep**, você poderá especificar outro caractere qualquer.
  - **end**:
    - Por padrão, a função **print()** imprime uma mensagem e em seguida gera uma quebra linha (faz um “\n” automaticamente).
    - Com o uso do parâmetro **end**, você poderá fazer com que todo comando **print()** não encerre com uma quebra de linha, mas com o uso de algum caractere, como espaço em branco por exemplo.
  - Para deixar tudo mais claro, veja a forma de utilizar esses parâmetros no exemplo do slide a seguir.

Parâmetros da função print() (2/2)

- A seguir, diversas possibilidades de utilização da função **print()**.

```
a=1; b=2; c=3
#imprime os valores de a, b, c separados p/ espaço
print(a,b,c)                #1 2 3

#parâmetro "sep": troca espaço p/ separador especificado
print(a,b,c,sep=';')        #1;2;3

#parâmetro "sep": troca espaço por nada (string vazia)
print(a,b,c,sep='')         #123

#parâmetro "end": por padrão, termina uma linha com \n
#(quebra de linha). Mas usando "end" você pode mudar
print("Ada",end=' ')
print("Lovelace")           #Ada Lovelace
```

Interpolação e escape na função print()

- Possibilidades de utilização da função **print()** para: (i) **interpolação**; (ii) escape p/ definir aspas, tabulação (**\t**) e quebra de linha (**\n**) e aspas.
  - Digite e analise esse exemplo !!!

```
# A interpolação facilita a inserção e formatação de
# conteúdo de variáveis em strings
nome='PI'
v=3.14159
print(f"{nome} com duas casas={v:.2f}") #PI … 3.14

# o sinal de escape (\) também permite a impressão de aspas
print("Imprimindo aspas \"") #imprimindo aspas "
print('Imprimindo aspas \'') #imprimindo aspas ’

# adicionando quebras de linha com \n
print("linha1\nlinha2\nlinha3")

# separando valores por tabulação com \t
print("coluna1\tcoluna2\tcoluna3")
```

Sintaxe Compacta (1/6)

- **Escreva um programa que imprima a palavra 'ENCE' 15 vezes na tela usando o while.**

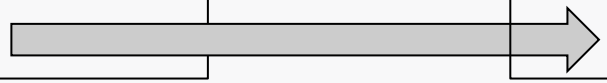
```
x = 0

while x < 15:
    print('ENCE')
    x = x + 1
```

```
>>> %Run ence15x.py

ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
ENCE
```

Sintaxe Compacta (2/6)

- **Incremento**
  - No programa, usamos a variável x para controlar o laço. A cada iteração, seu valor é incrementado em 1 (instrução x = x + 1).
  - Esse tipo de instrução – incrementar o valor de uma variável – é tão **comum em programas** que possui um nome: **incremento**.
  - Além de um nome específico, o incremento também possui uma sintaxe alternativa, mais compacta e preferida pelos programadores:
    - x = x + 1, também pode ser escrito como **x += 1**.
    - Sendo assim, poderíamos reescrever o programa da seguinte maneira:

```
x = 0

while x < 15:
    print('ENCE')
    x = x + 1
```



```
x = 0

while x < 15:
    print('ENCE')
    x += 1
```

Sintaxe Compacta (3/6)

- **Incremento**
  - Na verdade, podemos utilizar a sintaxe compacta para incrementar uma variável em qualquer valor, não apenas 1.

```
k += 5              # soma 5 ao valor corrente de "k"

v1 += 100           # soma 100 ao valor corrente de "v1"

soma += nota        # soma o valor armazenado em "nota"
                    # ao valor corrente de "soma"
```

Sintaxe Compacta (4/6)

- **Decremento**
  - Além do incremento, existe também o decremento, que significa subtrair algum valor de uma variável.
  - O decremento também tem uma sintaxe alternativa: v = i - 1 também pode ser escrito como v -= 1.

```
sal -= 50          #subtrai 50 do valor de "sal" em 50

temp -= x          #subtrai x do valor de "temp"
```

Sintaxe Compacta (5/6)

- **Sintaxe Compacta para outras operações aritméticas**
  - Também é possível utilizar a sintaxe curta para definir operações de multiplicação, divisão e potência.
  - Veja os exemplos abaixo:

```
w *= 4          # faz "w" receber seu valor atual vezes 4

u /= 3          # faz "u" receber seu valor atual dividido por 3

z **= 2         # faz "z" receber o quadrado de seu valor atual

t %= 2          # faz "t" receber o resto da divisão de seu
                # valor atual por 2

r //= 2         # faz "r" receber o quociente da divisão de seu
                # valor atual por 2
```

Sintaxe Compacta (6/6)

- **Testando Faixas de Valores**
  - Muitas vezes temos que implementar um teste lógico (seja para o **if**, **elif** ou **while**) onde é preciso testar se um valor está dentro de uma faixa.
    - **Exemplo**: testar se o valor da variável idade é entre 21 e 30
    - O jeito padrão é usar o operador lógico **and**

```
if idade > 20 and idade <= 30:
```

    - Porém, nessa situação, o Python também permite que você use a **sintaxe compacta** a seguir:

```
if 20 < idade <= 30:
```

Técnicas Úteis (1/8)

- **Estudo de Caso – programa que diz se número é primo**
  - Neste último exemplo, vamos apresentar algumas **técnicas gerais de programação** que você pode adotar na resolução de diversos problemas.
  - Como “estudo de caso”, vamos criar um programa que recebe como entrada um número e diz se ele é primo ou não.
  - Definição de número primo (Wikipedia):
    - “Um **número primo** é um número natural maior que 1 que não pode ser formado pela multiplicação de outros dois naturais menores”

Técnicas Úteis (2/8)

- **Número primo**
  - "Um **número primo** é um número natural maior que 1 que não pode ser formado pela multiplicação de outros dois naturais menores"
  - Veja que:
    - Nenhum número negativo é primo. (caso básico)
    - 0 e 1 não são primos. (caso básico)
    - 2 é o único primo que é par (exceção)
    - Os demais primos são ímpares (caso normal difícil)
  - Podemos tirar proveito dessas informações para simplificar o desenvolvimento de nosso programa.

Técnicas Úteis (3/8)

- **Técnica 1: tratar os casos básicos separadamente.**
  - Eles são simples, então normalmente vale a pena trata-los a parte logo no início do programa
    - Nenhum número negativo é primo. (caso básico)
    - 0 e 1 não são primos. (caso básico)

```
numero = int(input("Digite um número natural e eu te direi se é primo: "))

# trata os casos básicos
if numero < 0:
    print("Um número negativo não pode ser primo (tem que ser número natural).")
elif 0 <= numero <= 1:
    print(f"{numero} não é primo.")
```

Técnicas Úteis (4/8)

- **Técnica 2: tratar as exceções separadamente.**
  - Elas podem “complicar” a elaboração da solução geral do problema. Sendo assim, pode valer a pena tratar em separado.
    - 2 é o único primo que é par. (exceção)

```
numero = int(input("Digite um número natural e eu te direi se é primo: "))

# trata os casos básicos
if numero < 0:
    print("Um número negativo não pode ser primo (tem que ser número natural).")
elif 0 <= numero <= 1:
    print(f"{numero} não é primo.")
elif numero == 2:
    print(f"2 é primo.")
```

Técnicas Úteis (5/8)

- **Agora vamos tratar situação "normal"**
  - Um método comum para identificar um número primo é:
    - "*Sair dividindo o número por todos os primos menores ou iguais à sua raiz quadrada. Se nenhuma divisão der exata ele é primo*"
  - O problema dessa solução é que ela **difícil de implementar**!
    - Suponha que quero testar se 997 é primo
    - A raiz de 997 é 31,57 → 32 (arredondando para inteiro)
    - Então para testar se 997 é primo, eu teria que:
      - Conhecer todos os primos menores que 32
      - Dividir 997 por todos eles.
      - Se nenhuma divisão der exata, então saberei que 997 é primo
  - *A propósito: 997 é primo e os primos menores que 32 são {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}*

Técnicas Úteis (6/8)

- **Quebrando hipóteses**
  - A solução do slide anterior é muito difícil de implementar.
  - Então podemos fazer o seguinte:
    - Botar o computador para trabalhar pra gente.
      - Ele foi inventado para isso.
      - Ele é extremamente rápido para calcular.
    - E, assim, resolver o problema de forma mais simples.

Técnicas Úteis (7/8)

- **Quebrando hipóteses**
  - Nesse problema, fica fácil implementar se adotarmos a técnica de **quebra de hipótese**.
    - É uma técnica muito usada em programação
  - Podemos **assumir** de início **que o número é primo**
    - É a nossa hipótese inicial.
  - Depois saímos **dividindo** o número por **todos os valores** entre 2 e a raiz do número.
    - Botamos o computador para trabalhar pra gente
  - Se alguma **divisão** der **exata**, **quebramos a hipótese**
    - Saberemos que o numero não é primo.

Técnicas Úteis (8/8)

```
numero = int(input("Digite um número natural e eu te direi se é primo: "))

# trata os casos básicos
if numero < 0:
    print("Um número negativo não pode ser primo (tem que ser número natural).")
elif 0 <= numero <= 1:
    print(f"{numero} não é primo.")
elif numero == 2:
    print(f"2 é primo.")
else: # caso "normal", para testar do 3 em diante
    import math
    primo = True  # assume que é primo
    raiz = math.ceil(math.sqrt(numero)) # tira a raiz do número e arredonda pra cima
    x = 2
    while x <= raiz: # verifica o resto da divisão de numero por todos os inteiros
                     # entre 2 e raiz(numero) p/ verificar se hipótese é quebrada
        if numero % x == 0: primo = False
        x += 1

    # após o while, basta testar o valor da variável primo
    # se estiver False não é primo (hipótese foi quebrada). Se ainda estiver True, é primo.
    if not primo: print(f"{numero} não é primo.")
    else: print(f"{numero} é primo.")
```

Exercício (Desafio)

- **Otimize o programa dos números primos da seguinte forma**
  - Vamos fazer alterações para “poupar processamento” do computador, ou seja, fazer com que ele execute menos cálculos.
    - Isso acelera a velocidade de execução do programa.

  - ALTERAÇÃO 1:
    - Caso o número entrado seja **par diferente de 2**, o programa não deverá executar a rotina de “quebra de hipótese”
      - Deverá logo dizer que “não é primo”

  - ALTERAÇÃO 2 :
    - Na rotina de quebra de hipótese, faça com que o computador divida a variável numero por todos os **ímpares** entre 3 e raiz(numero).

      (*em vez de dividir por todos os números entre 2 e raiz(numero)*)