MC-102 — Aula 10
Formatação e tuplas

Instituto de Computação – Unicamp

4 de Setembro de 2019

Listas - remover elementos

- O comando **del lista[posição]** remove da lista o item da posição especificada.

```
>>> x = [40, 99, 30, 10, 40]
>>> del x[2]
>>> x
[40, 99, 10, 40]
```

- Também podemos remover um item da lista utilizando o método **remove**.

```
>>> x.remove(10)
>>> x
[40, 99, 40]
```

Listas - sort

- o metodo **sort** modifica a lista para que ela seja ordenada
- a função **sorted** retorna uma nova lista ordenada

```
>>> a= [67, 23,12,75]
>>> id(a)
4442180424
>>> a.sort()
>>> a
[12, 23, 67, 75]
>>> id(a)
4442180424   # o mesmo
>>> a=sorted(a)
>>> a
[12, 23, 67, 75]
>>> id(a)
4442469960   # mudou
```

- **sort** e **sorted** podem conter o parametro **reverse=True** que indica que a ordenação deve ser reversa , ou seja decrescente.

Formataçao

- o objetivo de formatação é construir um string onde partes dele vem do valor de variavies.
- por exemplo, a variável **nom** contem o nome do usuário e **saldo** o saldo da conta. e voce quer gerar o string "Bom dia NNNNN! Voce tem SSSSS no banco" onde o NNNNN é o valor da variável **nom** e SSSS o valor do saldo.
- até agora isso era feito com um **print** com vários atributos

```
print('Bom dia',nom,'! Voce tem',saldo,'no banco')
```

- Os comandos e operadores de formatação permitem gerar o string, que voce pode ou não imprimir

Formataçao

- O Python tem duas familias de formatadores, o novo e o velho.
- o novo usa **.format** e o velho **%**

```
nom = 'Diana'
saldo = 45.34
velho = 'Bom dia %s!  Voce tem %f no banco' % (nom,saldo)
novo = 'Bom dia {}!  Voce tem {} no banco'.format(nom,saldo)
```

Formatação velha

- Há o string de controle (e texto fixo) seguido do operador de formatação **%**, seguido da tupla (a ser vista na aula de hoje) dos valores a serem substituidos no string de controle.
- o **%** dentro do string de controle indica onde os valores serão inseridos e o caracter seguinte indica que tipo de valor será inserido
- **%s** para um string, **%d** para um inteiro, **%f** para um float.
- entre o **%** e a letra pode haver números e outros caracteres para controlar como o valor será convertido para string. Há dezenas de opções para esse controle.
- A mais util é o **%N.Df** que indica que o valor float usará *N* posições total, *D* das quais depois da casa decimal

```
velho = 'Bom dia %s!  Voce tem %.2f no banco' % (nom,saldo)
```

Formatação nova

- O string de controle (e texto fixo) é o argumento externo da função **format** (da mesma forma que a lista é o argumento externo do **append**) e os valores a serem substituidos são os argumentos do **format**
- o {} dentro do string de controle indica onde os valores serão inseridos
- não é preciso indicar o tipo do valor a ser substituido
- dentro do {} pode haver números e outros caracteres para controlar como o valor será convertido para string. Há ainda mais opçoes de controle na formatação nova.
- A mais util é o {**:N.Df**} que indica que o valor float usará *N* posições total, *D* das quais depois da casa decimal

```
novo = 'Bom dia {}!  Voce tem {:.2f} no banco'.format(nom,saldo)
```

Formatação velha e nova

- A documentação do Python para a formatação velha esta em `https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting`. Oficialmente a formatação velha é chamada de printf-style String Formatting
- A documentação do Python para a formatação nova esta em `https://docs.python.org/3/reference/lexical_analysis.html#f-strings`. Oficialmente ela é chamada de Formatted string literals
- o site `https://pyformat.info` contem uma boa comparação linha por linha das duas formatações

Tuplas

- Tuplas são similares a listas, isto é, uma sequência de dados de qualquer tipo.
- Porém, ao contrário de listas, as tuplas são imutáveis.
- Tuplas são representadas por uma sequência de valores separados por vírgula, e entre um abre e fecha parênteses.
  - `(1, 2.3, 5, ''aaa'')` é uma tupla de 4 elementos.
- As operações para acessar os elementos ou sub-sequências de um lista e de um string, também funcionam em tuplas.

```
>>> a = (1,2.3,5,'aaa')
>>> a[2]
5
>>> a[1:3]
(2.3, 5)
```

Tuplas

- Como strings, tuplas são imutáveis.

```
>>> a=(1,2,5,"aa")
>>> a[2]=9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuplas - empacotamento e desempacotamento

- Os elementos de uma tupla podem ser acessados de uma forma implícita na atribuição (conhecido como desempacotamento).

```
>>> x,y=(9,10)
>>> x
9
>>> y
10
```

- A tupla também pode ser implicitamente criada apenas separando os elementos por virgula (conhecido como empacotamento).

```
>>> 67, 90
(67, 90)
```

Tuplas como valores retornados de funções

- Se uma função precisa retornar 2 ou mais valores, ela pode retornar uma tupla desses valores
- faça uma função que retorna o maior e a posição do maior numa lista

```
def maior (l):
    ma = l[0]
    pos=0
    for i in range(1,len(l)):
          if l[i] > ma:
               ma=l[i]
               pos = i
    return ma,pos
```

- o **return** empacota a tupla dos 2 valores
- na chamada, podemos separar os 2 valores implicitamente (desempacotamento)

```
x, i = maior(notas)
```

Tuplas

- Tuplas permite uma forma facil de trocar o valor de 2 variáveis

```
a, b = b, a
```

Funções - o que sabemos até agora

Funções sem retorno

- Até agora nossas funções parecem com funções matemáticas, no sentido que elas recebem valores e retornam um valor que será usado (normalmente em expressões).
- Usando tuplas, as funções podem "retornar mais de um valor" (na verdade ela so retornam um valor, que é uma tupla).
- Funções podem não retornar nada e neste caso, sua chamada não é parte de uma expressão

```
def imprimeCaixa (numero):
  tamanho=len(str(numero))
  for i in range(12+tamanho):
    print('+',end='',sep='')
  print()
  print('| Número:',numero,'|')
  for i in range(12+tamanho):
    print('+',end='',sep='')
  print()

imprimeCaixa(10)
imprimeCaixa(23456)
```

Variáveis locais e variáveis globais

- Uma variável é chamada local se ela é **criada ou alterada** dentro de uma função. Nesse caso, ela existe somente dentro daquela função, e após o término da execução da mesma a variável deixa de existir.
- **Variáveis parâmetros também são variáveis locais**.
- Uma variável é chamada global se ela for criada fora de qualquer função. Essa variável pode ser visível por todas as funções.

Escopo de variáveis

- O escopo de uma variável determina de quais partes do código ela pode ser acessada, ou seja, de quais partes do código a variável é visível.
- A regra de escopo em Python é bem simples:
  - As variáveis globais são visíveis por todas as funções.
  - As variáveis locais são visíveis apenas na função onde foram criadas.

Variáveis locais e globais

```
def f1(a):
    print(a+x)

def f2(a):
    c=10
    print(a+x+c)


x=4
f1(3)
f2(3)
print(x)
```

- Tanto **f1** quanto **f2** usam a variável **x** que é global pois foi criada fora das funções.
- A saída do programa será:

```
7
17
4
```

Variáveis locais e globais

```
def f1(a):
    x = 10
    print(a+x)

def f2(a):
    c=10
    print(a+x+c)


x=4
f1(3)
f2(3)
print(x)
```

- Neste outro exemplo **f1** cria uma variável local **x** com valor 10. O valor de **x** global permanece com 4.
- A função **f2** por outro lado continua acessando a variável global **x**.
- A saída do programa será:

```
13
17
4
```

Variáveis locais e globais

Veja este outro exemplo:

```
def f1(a):
    print(a+x)

def f3(a):
    x=x+1
    print(a+x)

x=4
f1(3)
f3(3) # este comando vai dar um erro
```

A saída será:

```
7
Traceback (most recent call last):
  File "teste.py", line 10, in <module>
    f3(3) # este comando vai dar um erro
  File "teste.py", line 5, in f3
    x=x+1
UnboundLocalError: local variable 'x' referenced before assignment
```

O que aconteceu???

Variáveis locais e globais

Veja este outro exemplo:

```
def f1(a):
    print(a+x)

def f3(a):
    x=x+1
    print(a+x)

x=4
f1(3)
f3(3) # este comando vai dar um erro
```

- Na função **f3** é alterado o valor de **x**, e pela regra de Python esta variável é portanto uma variável local. O erro ocorre pois está sendo usado uma variável local **x** (no **x+1**) antes dela ser criada!

Variáveis locais e globais

- Para que **f1** use **x** global devemos especificar isto utilizando o comando **global**.

```
def f1(a):
    print(a+x)

def f3(a):
    global x
    x=x+1
    print(a+x)

x=4
f1(3)
f3(3) # sem erro
print(x)
```

- A saída neste exemplo será:

```
7
8
5
```

- Note que o valor de **x** global foi alterado pela função **f3**.

Variáveis locais e globais

```
def f3(a):
    c=10
    print(a+x+c)
x=4
print(c)
```

- A variavél **c** foi criada dentro da função e ela só existe dentro desta. Ela é uma variável **local** da função **f3**.

Saída:

```
Traceback (most recent call last):
  File "teste.py", line 5, in <module>
    print(c)
NameError: name 'c' is not defined
```

Variáveis locais e globais

```
def f4(a):
    c=10
    print("c de f4 :",c)
    print(a+x+c)
x=4
c=-1
f4(1)
print("c global:", c)
```

- Neste caso existe uma variável **c** no programa principal e uma variável local **c** pertencente à função **f4**.
- Alteração no valor da variável local **c** dentro da função não modifica o valor da variável global **c**, a menos que esta seja declarada como **global**.

Saída:

```
c de f4 : 10
15
c global: -1
```

Variáveis locais e globais

```
def f4(a):
    global c
    c=10
    print("c de f4 :",c)
    print(a+x+c)
x=4
c=-1
f4(1)
print("c global:", c)
```

- Neste caso a variável **c** de dentro da função **f4** foi declarada como global.
- Portanto é alterado o conteúdo da variável **c** fora da função.

Saída:

```
c de f4 : 10
15
c global: 10
```

Variáveis locais e variáveis globais

- O uso de variáveis globais deve ser evitado pois é uma causa comum de erros:
  - Partes distintas e funções distintas podem alterar a variável global, causando uma grande interdependência entre estas partes distintas de código.
- A legibilidade do seu código também piora com o uso de variáveis globais:
  - Ao ler uma função que usa uma variável global é difícil inferir seu valor inicial e portanto qual o resultado da função sobre a variável global.

Listas em funções

```
def f1(a):
    a.append(3)

a = [1,2]
f1(a)
print(a)
```

- Neste caso mesmo havendo uma variável local **a** de **f1** e uma global **a**, o conteúdo de **a** global é alterado.
- O que aconteceu???

Saída:

```
[1, 2, 3]
```

Listas em funções

```
def f1(a):
    a.append(3)

a = [1,2]
f1(a)
print(a)
```

- Lembre-se que **a** local de **f1** recebe o identificador da lista de **a** global. Como uma lista é mutável, o seu conteúdo é alterado.

Saída:

```
[1, 2, 3]
```

Listas em funções

```
def f1(a):
    a = [10,10]

a = [1,2]
f1(a)
print(a)
```

- O que será impresso neste caso???.

Listas em funções

```
def f1(a):
    a = [10,10]

a = [1,2]
f1(a)
print(a)
```

- Neste caso a variável **a** local de **f1** recebe uma nova lista, e portanto um novo identificador.
- Logo a variável **a** global não é alterada.

Saída:

```
[1, 2]
```

Listas em funções

```
def f1():
    global a
    a = [10,10]

a = [1,2]
f1()
print(a)
```

- O que será impresso???

Listas em funções

```
def f1():
    global a
    a = [10,10]

a = [1,2]
f1()
print(a)
```

- Neste caso **a** de **f1** é global e portanto corresponde a mesma variável fora da função.
- A variável **a** tem seu identificador alterado dentro da função para uma nova lista com conteúdo [10, 10].

Saída:

```
[10, 10]
```