Programação de Computadores

Python

@AlexFurtunato

hierarquia conceitual do python

1. *Programas* são compostos por *Módulos*
2. *Módulos* contém *Declarações*
3. *Declarações* contém *Expressões*
4. *Expressões* criam ou processam *Objetos*

Operação de atribuição alternativa

- Para determinados padrões de atribuição existe uma forma alternativa:

X = X + Y

Pode ser rescrita como:

X += Y

- Outras possibilidades são:
  - Ex:
    - var = var + 1
    - var += 1

X += Y	*X &= Y*	*X -= Y*	*X |=Y*
*X *= Y*	*X ^= Y*	*X /= Y*	*X >>= Y*
X %= Y	*X <<= Y*	*X **= Y*	*X //= Y*

Nomenclatura de variáveis

- A regra básica para nomear variáveis é:

Sintaxe:
(Underscore ou Letra) + (Qualquer número de letras, dígitos ou underscore)
Ex: _spam, spam_1, Spam_

- Letras maiúscula e minúscula são diferenciadas
  - Ex: SPAM é diferente de Spam
- Algumas palavras reservadas não podem ser utilizadas
  - False, None, True, and, as, assert, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield

Condições: IF

▸Estrutura de controle para executar de maneira seletiva um bloco de código:

Sintaxe:

```
if teste1:
    bloco1
elif teste2:
    bloco2
elif...
    ...
else:
    blocoX
```

condições: IF

- Ex:

```
>>> x = "coelho"
>>> if x == “vaca”:
...     print(“Leite")
... elif x == “Boi”:
...     print(“Carne”)
... else:
...     print(“Mais Coelho”)
```

O equivalente a “Switch” ou "case"

- Em Python, não existe uma estrutura de controle equivalente a “Switch" ou “Case" utilizados em outras linguagens. Para fazer uma escolha baseada em um valor, ou se utiliza "if... elif... else” ou um dicionário.
  - Exemplo:

```
>>> escolha = “spam"
>>> D = {‘spam’: 1.25, ‘ham’: 1.99, ‘eggs’: 0.99, ‘bacon’: 1.10}
>>> print(D[escolha])
1.25
```

condições: IF

Repetições: while

- Estrutura de repetição que executa um bloco de código "*enquanto*" uma determinada condição se mantém verdadeira.

while teste1:
    bloco

- Em um laço while, podem ser usadas as instrução “break”, “continue”
- Além disso, o laço “while" pode opcionalmente ter um “else"

```
>>> x = 'spam'
>>> while x:
...     print(x, end=' ')
...     x = x[1:]
...
spam pam am m
```

Repetições: while

- ‘break' - Sai fora do laço que o contém
- ‘continue' - Pula para o início do laço para iniciar uma nova iteração
- ‘else' em um laço while - Bloco a ser executado quando o laço é terminado sem o uso do ‘break'
- Ex:

```
>>> x = 10
>>> while x:
...     x -= 1
...     if x % 2 != 0:
...         continue
...     print(x, end=‘ ’)
8 6 4 2 0
```

Repetições: while

- Ex:

```
>>> x = y // 2
>>> while x > 1:
...     if y % x == 0:
...         print(y, 'tem fator', x)
...         break
...     x -= 1
... else:
...     print(y, 'eh primo')
```

Repetições: For

- Estrutura de repetição genérica para iterar sobre itens de uma sequencia ordenada ou outro objetos iteráveis.

```
for alvo in objeto:
    bloco
else:
    blocoX
```

- Em um laço for, podem ser usadas as instrução “break”, “continue” tendo o mesmo efeito do laço “While"
- Além disso, o laço “for" pode opcionalmente ter um “else” cujo bloco é executado ao terminar a iteração e não houver uma saída com o uso da instrução "break"

```
>>> for x in [‘spam’, ‘eggs’, ‘bacon']:
...     print(x, end=‘ ‘)
...
spam eggs bacon
```

Repetições: For

- Iteração sobre sequência de Tuplas. Nesse caso, o alvo pode ser uma Tupla de alvos:

```
>>> T = [(1,2), (3,4), (5,6)]
>>> for (a, b) in T:
...     print(a,”e”,b)
...
1 e 2
3 e 4
5 e 6
```

Repetições: For

- Iteração sobre dicionário, o alvo recebe as chaves do dicionário.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for chave in D:
...     print(chave,"=>",D[chave])
...
a => 1
b => 2
c => 3
```

Repetições: For

- Em Iteração sobre dicionário, pode-se fazer uso de Tuplas para recuperar no alvo as chaves e valores do dicionário.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> list(D.items())     # Para recuperar as chaves e valores em uma lista de tuplas
dict_items([('c', 3), ('a': 1), ('b': 2)])
>>> for (chave,valor) in D.items():
...     print(chave,"=>",valor)
...
a => 1
b => 2
c => 3
```

Repetições: For

- Em Python, os loops com for podem ser feitos sem que haja necessidade de usar contadores. É feita uma iteração sobre itens de uma sequência.
- Em alguns casos, porém, pode ser necessário fazer iterações especializadas que necessitem de instruções adequadas.
- Podemos usar algumas built-ins que tornam o uso do For mais especializado:
  - range - Produz uma sequência de inteiros, que pode ser usados para indexar
  - zip - Retorna uma série de Tuplas sobre sequências
  - enumerate - Gera o par índice,valor de objetos iteráveis

Repetições: For

- Uso do Range

```
>>> list(range(5)), list(range(2,5)), list(range(0, 10, 2))
[0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8]
>>> for i in range(3):
...     print(i,"pythons")
...
0 pythons
1 pythons
2 pythons
```

Repetições: For

- Uso do Range

```
>>> s = 'spam'
>>> for i in range(len(s)):
...     print(i, s[i], sep=`-`, end=` `)
...
0-s 1-p 2-a 3-m
```

Repetições: For

- Uso do zip

```
>>> L1 = [‘a’, ‘b’, ‘c’, ‘d’]
>>> L2 = [1, 2, 3, 4]
>>> zip(L1, L2)
<zip object at 0x02A31300>
>>> list(zip(L1, L2))
[(‘a’, 1), (‘b’, 2), (‘c’, 3), (‘d’, 4)]
>>> for (x, y) in list(zip(L1, L2)):
...     print(x, “-“, y)
...
a - 1
b - 2
c - 3
d - 4
```

Repetições: For

- O zip pode ser usado também para construir dicionário a partir de listas

```
>>> chaves = ['a', 'b', 'c', 'd']
>>> valores = [1, 2, 3, 4]
>>> D1 = dict(zip(chaves, valores))
>>> D1
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> D2 = {k: v for (k, v) in zip(chaves, valores)}
>>> D2
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Repetições: For

- Uso do enumerate

```
>>> s = 'spam'
>>> enumerate(s)
<enumerate object at 0x02A31350>
>>> list(enumerate(s))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
>>> for (offset, valor) in enumerate(s):
...     print(offset, valor)
...
0 s
1 p
2 a
3 m
```

Funções

- Estrutura de programação utilizada em qualquer linguagem. Em outras linguagens pode receber outros nomes: subrotinas, métodos, procedimentos.
- As funções têm como principal objetivo a diminuição de código redundante em um programa. Com funções pode-se “empacotar” um bloco de código que pode ser utilizado em mais de um local no programa.
- As funções também tornam o código mais organizado através da quebra do código em peça “funcionais” e tão imprescindível quanto maior for o código do programa.

definição de funções

- A criação de uma função é feita através do uso da declaração “def"
- Na criação da função são definidos também uma lista de argumentos que são passados para a função

```
def nome([arg1], [arg2], [arg3], […] ):
    código
    return [objeto]
```

- A declaração “return” devolve a execução para o ponto de chamada da função. Opcionalmente, a função pode retornar um valor para o ponto de chamada após a sua execução
- Exemplo:

```
>>> def multiplica(x, y):
...     valor = x * y
...     return valor
>>> multiplica(5, 3)
>>> 15
```

Escopo de variáveis

- Em python, variáveis definidas dentro do bloco de uma função só existem dentro desta função e são chamadas de variareis “locais” da função.
- Portanto, duas variáveis com o mesmo nome podem existir de forma separada dentro e fora de uma função.
- Variáveis definidas fora de qualquer definição de função em um programa são chamadas de variáveis “globais”
- Exemplo:

```
>>> x = 99                  # variável Global x
>>> def func():
...     x = 88              # Variável Local x
```

Escopo de variáveis

- Variáveis globais são automaticamente acessadas dentro das funções, desde que não haja a criação de uma variassem local com o mesmo nome

```
>>> x = 99
>>> def func1(y):
...     z = x * y
...     return z
>>> def func2(y):
...     x = 20            # variável local
...     z = x * y
...     return z
>>> func1(2)
198
>>> func2(2)
40
```

Escopo de variáveis

- Para ter acesso a variáveis globais de dentro da função, utiliza-se a declaração “global”

```
>>> x = 99
>>> def func2(y):
...     global x                # variável Global
...     x = 20
...     z = x * y
...     return z
>>> func2(2)
40
>>> x
20
```

passagem de argumentos

- É possível chamar funções com um número de argumentos variável
- Nessa caso, pode-se utilizar três formas de passagem de argumento
  - Argumentos com valores default
  - Argumentos com passagem por chave
  - Lista arbitrária de argumentos

Passagem de argumentos

- Argumentos com valores default

```
>>> def func(a, b=2, c=3):
...     z = a*b*c
...     return z
>>> func(10)
60
>>> func(10, 5)
150
>>> func(10, 5, 6)
300
```

Passagem de argumentos

- Argumentos com passagem por chave

```
>>> def func(a, b=2, c=3):
...     z = a*b*c
...     return z
>>> func(a=10)
60
>>> func(10, c=5)
100
>>> func(10, b=5)
150
>>> func(b=6, c=20)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    func(b=6, c=20)
TypeError: func() missing 1 required positional argument: 'a'
```

Passagem de argumentos

- Lista arbitrária de argumentos

```
>>> def func(a, *args):
...     print(a)
...     for i in args:
...         print(i)
...     return
>>> func(10)
10
>>> func(10, 5)
10
5
>>> func(10, 5, 6)
10
5
6
```

Exemplo de uso de função

- programa com função que trata data por extenso

```
def data_por_extenso(texto):
    meses=[‘janeiro’,'fevereiro','março','abril','maio','junho','julho',
           ‘agosto','setembro', 'outubro','novembro','dezembro']
    data_num = texto.split('/')
    return data_num[0] + ' de ' + meses[int(data_num[1])-1] + ' de ' + data_num[2]

data_por_extenso(’01/01/1970’)
data_por_extenso(’01/06/2016’)
```

built-in __name__

- Um módulo no Python pode ser importado para outro módulo através do “import" ou ser executado como um programa
- Qualquer módulo em Python possui um atributo denominado “__name__” que o Python cria automaticamente como segue:
  - No arquivo de programa executado (top-level) o atributo __name__ recebe o valor “__main__” quando a execução se inicia
  - No arquivo que está sendo importado, __name__ recebe o nome do módulo
- Portanto, a estrutura de um módulo importável que poderia ser também executado como programa, seria:

```
import x, y, z

def funcao_x():
    código
    return

def main():
    print('Executando...')

if __name__ == '__main__':
    main()
```

Exemplo de uso do __name__

▸ programa com função que trata data por extenso

```
import sys

def data_por_extenso(texto):
    meses=['janeiro','fevereiro','março','abril','maio','junho','julho',
           'agosto','setembro', 'outubro','novembro','dezembro']
    data_num = texto.split('/')
    return data_num[0] + ' de ' + meses[int(data_num[1])-1] + ' de ' + data_num[2]

def main():
    if len(sys.argv) == 2:
        print(data_por_extenso(sys.argv[1]))
    else:
        print('Voce deve passar apenas um argumento com a data a ser convertida!')

if __name__ == '__main__':
    main()
```

Exercício

- Modifique o código anterior, incluindo um método que teste se a data passada como argumento está dentro de algumas regras esperadas:
  - Dia do mês entre 1 e 31
  - Mês entre 1 e 12
  - Ano com quatro dígitos numéricos
  - Separador como “/“