



Listas, Sequências(2)

Listas Multidimensionais

Em alguns casos, será necessário que relacionemos um tipo de dado de forma à expressar uma modelagem multidimensional (estrutura “linha X coluna”).

Um exemplo simples: Um tabuleiro de Jogo da Velha!

Em python, um artifício comumente usado para isto é o uso de uma “lista de listas”.

```
1 tabuleiro = [] # Inicializa como uma lista vazia "[]"
2 for cont in range (3): # Desejamos construir um tabuleiro "padrao" para o jogo da velha
3     tabuleiro.append([0,0,0]) # A cada iteracao, adiciona uma lista preenchida por "0"
4     print tabuleiro
5 #Com apenas 3 linhas de codigo, fomos capazes de criar o tabuleiro!
```

- Jogo completo:
(Não se preocupe
em tentar
entendê-lo por
completo ainda)

```
1  tabuleiro = [] # Inicializa como uma lista vazia []
2  for cont in range(3): # Desejamos construir um tabuleiro "padrao" para o jogo da velha
3      tabuleiro.append([0,0,0]) # A cada iteracao, adiciona uma lista preenchida por "0"
4  print tabuleiro
5  #Com apenas 3 linhas de codigo, fomos capazes de criar o tabuleiro!
6
7  def vitoria(tabuleiro_a_analisar):
8      for cont in range(3):
9          if sum(tabuleiro_a_analisar[cont]) == 3 or sum(tabuleiro_a_analisar[cont]) == -3:
10             return True # Alguem venceu horizontalmente
11      somavertical = 0
12      for contx in range(3):
13          for conty in range(3):
14             somavertical += tabuleiro_a_analisar[conty][contx]
15             if(somavertical == 3 or somavertical == -3):
16                 return True # Alguem venceu verticalmente
17      somavertical = 0
18      #Diagonais
19      somadiagonal = 0
20      for cont in range(3):
21          somadiagonal += tabuleiro_a_analisar[cont][cont]
22      if(somadiagonal == 3 or somadiagonal == -3):
23          return True
24      somadiagonal = 0
25      for cont in range(3):
26          somadiagonal += tabuleiro_a_analisar[cont][2 - cont]
27      if(somadiagonal == 3 or somadiagonal == -3):
28          return True
29      return False
30
31  def imprimetabuleiro(tabuleiro_a_imprimir):
32      for linha in range(3):
33          print tabuleiro[linha]
34
35  turno = 0
36  while (not(vitoria(tabuleiro)) and turno < 9):
37      imprimetabuleiro(tabuleiro)
38      if(turno%2 == 0):
39          x = int(raw_input("Jogador 1, escolha a coordenada x: "))
40          y = int(raw_input("Jogador 1, escolha a coordenada y: "))
41
42          if(tabuleiro[x][y] == 0):
43              tabuleiro[x][y] = 1
44              turno += 1
45          else:
46              turno += 1
47      else:
48          x = int(raw_input("Jogador 2, escolha a coordenada x: "))
49          y = int(raw_input("Jogador 2, escolha a coordenada y: "))
50
51          if(tabuleiro[x][y] == 0):
52              tabuleiro[x][y] = -1
53              turno += 1
54          else:
55              turno += 1
56      imprimetabuleiro(tabuleiro)
57  print "Fim de jogo"
```

Mutabilidade

Python, em sua essência, admite dois tipos, no que tange à mutabilidade. São eles os tipos mutáveis e os tipos imutáveis.

- Imutáveis: (Dada sua construção), Um tipo de dados em que o elemento que compõe sua estrutura não pode ser modificado. Ex.: string
- Mutáveis: Tipo de dados onde o elemento pode ser modificado. Ex.: list

Tuplas

Tuplas são um tipo de dados bastante peculiar em python.

Apesar de possuir características de uma lista, uma tupla é um tipo imutável, tal qual uma string. Se comparadas à listas, tuplas são um tanto quanto “mais restritivas”.

Talvez lhe venha a pergunta “mas se ela é mais restritiva que a lista, então é um tipo inútil?”. Não é bem assim.

Dois detalhes que temos de levar em consideração:

- Tuplas, em python, são muito mais “leves” no quesito consumo de recursos, do que listas.
- Tuplas são parte essencial de muitas funções preexistentes de python.

Novamente, quando e como usar, dependerá do problema e do programador!

```

1  tupla_vazia = () # Inicializa uma tupla vazia.
2
3  tupla_numerica = (1, 2) # Tupla numerica de 2 elementos
4
5  tupla_numerica = (1, 2, 3, 4) # Tupla numerica com 4 elementos
6
7  print tupla_numerica
8
9  print len(tupla_numerica) # Lembra-se de len()? Aqui, nos dara o tamanho da tupla
10
11 tupla_aeds = ("AEDs 1", "AEDs 2", "AEDs 3") # Tupla de strings
12
13 print tupla_aeds[0] # Acessando e imprimindo um elemento da tupla
14
15 #Tuplas tambem podem ser definidas sem os "()". Na realidade, python
16 #ira avaliar qualquer elemento separado por "," e sem outro delimitador
17 #como uma Tupla.
18
19 tupla_sem_parenteses = 1,2,3
20
21 print type(tupla_sem_parenteses) # type() nos permite obter o tipo associado a uma variavel, da forma como python o "enxerga"
22
23 #Tuplas sao bastante usadas em python para a troca de valores.
24 #Esta operacao, alem de "economica", eh bastante simples de se compreender
25
26 a = 0
27 b = 1
28
29 print a, b
30
31 a, b = b, a # Troca por tupla ou "Tuple Swapping"
32
33 print a, b

```

Mais sobre strings e tipos sequenciais

Tal qual listas, strings também podem ser acessadas pela notação de índice:

```
1 string_a_acessar = "Cappuccino"
2 for i in range(len(string_a_acessar)):
3     print string_a_acessar[i] # Acessando via notacao de indice
```

```
antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBásico/Slides/Exemplos/Sequences$ python stringsequences.py
C
a
p
p
u
c
c
i
n
o
```


Strings são imutáveis em python. No entanto, existe um tipo oriundo do módulo “**UserString**”, conhecido como `MutableString`, que é, como o nome diz, mutável. Esta classe é atualmente, não recomendada pela comunidade python.

Como uma alternativa, pode-se trabalhar com uma lista de caracteres, por exemplo.

Slicing (Fatiou, passou! Boa 06!)

Slicing (Fatiamento) é uma característica de tipos baseados em sequencias, como string e list.

O slicing permite extrair uma parte dos dados definidos em uma estrutura, de forma à seguir uma “regra”. É bastante útil, em especial no processamento de textos. De fato, as capacidades de slicing são um dos motivos pelo qual python é altamente usado para scripts de automação.

Slicing é feito através do operador slice: “[x:y]”, com x e y sendo os limites/trecho desejado.

- Funcionamento do slicing:

```
1  #Slicing em strings
2  bolo = "The cake is a lie!"
3
4  #[inicio : fim : passo]
5
6  print bolo[-1] #Contagem negativa, ou seja, parte da direita para esquerda
7
8  print bolo[4:8] #Slicing basico
9
10 print bolo[10:15]
11
12 print bolo[14:] #A partir do indice 14
13
14 print bolo[:3] #Ate o indice 3
15
16 print bolo [0:15:2] # Similar a sintaxe operacao de acesso
17
18 #Slicing em lista
19 lista_jantar = ['Torta', 'Pave', 'Empada', 'Folhado', 'Desfiado']
20
21 print lista_jantar[0:4:2]
22
```

Exercício #12 (5 min.)

Dada a seguinte string:

- “Corta para 1, corta para 2”

Através dos operadores de slicing, defina duas novas variáveis, “var1” e “var2”. Faça ambas receberem a respectiva parte da string.

- Dica: Lembra do len()? Isto seria uma boa hora para usá-o.

Exercício #13 (15 min.)

O azar aqui é de quem não quer aprender python...

Dada a seguinte lista:

- [“Hadouken”, “Kamehameha”, “Serious_Punch”, “Astucia”, “Dinheiro”, “Force_Lightning”, “Orar_para_Existencia_X”]

Realize as seguintes operações, em código, sobre esta lista:

- Alterar o item de índice 5 (iniciado de 0) para “Mind_Trick”;
- Crie uma nova lista, contendo apenas os itens iniciados por vogal (faça programaticamente, não faça “na mão”);
- Voltando para a primeira lista, escolha qualquer índice, exceto o último, e substitua todos os valores **à partir** de tal índice por “Sem_Poder”

Dicionários

Um dicionário (em programação), bem como seu correspondente no mundo real, consiste numa estrutura que se baseia em associações.

Por exemplo, no mundo real, um “dicionário de significados” consiste em um livro (ou lista de textos) que contém associação palavra – significado.

Em um programa python, um dicionário assume associações da forma chave – artefato (este artefato sendo uma estrutura).

Os dicionários, por si, são elementos **mutáveis**. As chaves que os mesmos agrupam são **imutáveis**. Já os elementos associados às chaves, podem ser tanto um quanto o outro.

Outra coisa que devemos ter atenção é na questão de chaves com o mesmo identificador.

- Declaração de um dicionário e exemplo de uso:

```
1 dados_pessoa = {"Nome" : "Hannibal", "Sobrenome" : "Lecter", "Profissao" : "Psicologo", "Detalhes" : "Curte uns rango diferenciado..."}
2 #Exemplo de dicionario. Note a estrutura organizada em chaves
3 print dados_pessoa["Nome"]#Imprime o valor cuja chave eh "Nome"
4 print dados_pessoa
5
6 mapeamento_id = {0 : "Alice", 1 : "Bob", 2 : "Carl"}#Mapeamento de ids, uma possivel aplicacao para dicionarios
7
8 #Algumas funcionalidades especificas de dicionarios
9
10 print mapeamento_id.keys()#Somente as chaves
11 print mapeamento_id.values()#Somente os valores
12
13 mapeamento_id2 = {2 : "Charlinho", 42 : "Irineu"}
14
15 mapeamento_id.update(mapeamento_id2) # Update atualiza/complementa valores entre dois dicionarios
16
17 print mapeamento_id
18
19 mapeamento_id.clear()#Remove todos os pares presentes
```

Dicionários são altamente recomendados quando houver alguma relação chave – valor, que realmente necessite ser explicitada em código.

- Um exemplo: Trabalhar com mapeamento pessoa – dados;

A organização é da forma {chave:valor}. Chaves devem ser únicas.

Exercício #14 (10 min.)

Dado o seguinte dicionário:

```
{'Universo_Discreto' : 'Computacao', 'Boson_Treinamentos' :  
'Programacao', 'The_Linux_Foundation' : 'Open Source', 'serpentZA' :  
'Entretenimento', 'GotoConferences' : 'Computacao'}
```

- Acesse e imprima o elemento relacionado à chave “The_Linux_Foundation”;
- Adicione mais um item, com valores chave e conteúdo de sua preferência;
- Teste a função len() sobre este dicionário;

Funções

Em nossa vida, muitas “ações”, após efetuadas uma vez, tornam-se tal como um “padrão em nossa mente”.

Exemplos:

- Pedalar uma bicicleta;
- Tomar banho;
- Efetuar o pagamento de um boleto;



Podemos dizer que estas ações podem ser “contidas” em um escopo, para que sejam feitas quando necessário, ou seja, quando a gente precisar, lembraremos.

Voltando para a programação, temos que existem diversas formas de se definir uma funcionalidade que possa ser usada mais de uma vez. A mais comum é através do uso de funções.

Por definição, uma função é um conjunto de instruções sob um identificador comum (nome), que quando invocada, realiza determinada tarefa.

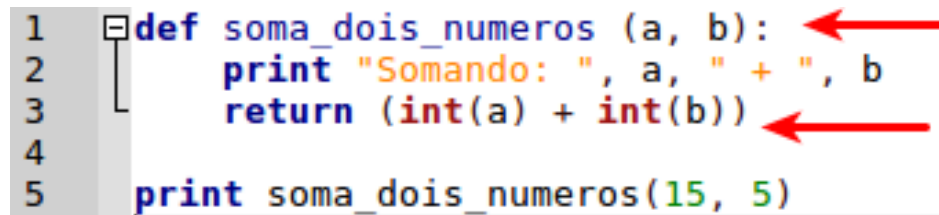
Já fizemos uso de algumas funções, como por exemplo, a “len()”. Porém, agora iremos ser capazes de escrever nossas próprias funções.

A este ato de “conter parte de um código para possível reutilização e afins” damos um nome especial: “Modularização”.

Funções são bastante úteis não só por questão de reaproveitamento de código, mas também pela própria questão da organização do mesmo.

- Sintaxe de uma função em python:

```
1 def soma_dois_numeros (a, b):  
2     print "Somando: ", a, " + ", b  
3     return (int(a) + int(b))  
4  
5 print soma_dois_numeros(15, 5)
```



Vamos trabalhar um pouco mais sobre cada nova terminologia aqui...

“DEF” criando uma função

Em python, uma função é definida pela palavra-chave “def”. Logo em seguida, evocamos o “nome” da função e os parênteses. Por fim, vêm o operador “:”, que delimita todo um bloco para a função.

Uma função é definida total e exclusivamente pelos comandos que se localizam no bloco da mesma.

Dentro da função, podemos “programar normalmente”, seguindo a lógica desejada. Não existe uma “restrição específica” aqui.

```

1  def seleciona_classe(caracteristica): # Outro exemplo
2      if(caracteristica == "Forca"):
3          print "Voce tera alta habilidade de ataque! Hora do Show!"
4      elif(caracteristica == "Astucia"):
5          print "Sua estrategia eh elevada! Nao contava com isso?"
6      elif(caracteristica == "Stealth"):
7          print "Voce eh muito furtivo! Nunca nem vi..."
8      else:
9          print "O que voce esta fazendo, afinal?"
10
11     seleciona_classe("Astucia") # Chamada
12
13  def parabains(): # Exemplo de funcao sem nenhum parametro
14      print 4*"Parabains!"
15
16
17  parabains()

```

Chamando uma função

Declarar e escrever uma função apenas serve para definir a mesma. Para que ela seja, devidamente, executada, é necessário que ela seja **chamada**.

Visto que já fizemos uso de algumas funções, já temos uma ideia de como realizar a chamada.

Sintaxe:

```
1 def chama_o_duda():  
2     print "Duda!"  
3  
4 def pizza (sabor):  
5     print "Entregue pizza de ", sabor  
6  
7 chama_o_duda() #Esta eh a chamada da funcao  
8  
9 pizza("Calabresa") # Chamada passando a funcao uma String
```


Note que nada no bloco da função é executado, sem que haja a chamada da função.
Desta forma, temos que funções também são formas de **alterar o fluxo** de um programa.

```
1 print "Isto sera a primeira linha a ser interpretada"
2
3 def dentro():
4     print "Estou dentro da funcao agora"
5
6 print "Este aqui eh o escopo global do programa"
7 print "Eh bem legal aqui, nao acha?"
8
9 def nuncasera(): # Nao ha uma especificacao para "lugar" da funcao
10     print "Esta frase nunca sera interpretada neste codigo, nao ha chamada de funcao"
11
12     #Apesar disto, muitos programadores preferem coloca-las no inicio
13     #Python somente interpretara uma chamada a uma funcao ja declarada anteriormente
14
15     dentro() # Chamada
16     ultima() # Ira gerar um erro em tempo de execucao
17
18 def ultima ():
19     print "No codigo, estou bem no fim"
20
```

Lembrando que declarar funções no começo do programa, além de aumentar a legibilidade, é uma boa prática de programação.

Exercício #15 (5 min.)

Crie uma nova função. Desta vez, a mesma deve imprimir “Contando de 1 a 50” e em seguida, uma contagem de 1 a 50.

Execute a chamada da mesma abaixo no código.

Escopo de funções

Escopo diz respeito literalmente àquilo que é enxergável pelo código. Até este momento, temos definidos dois escopos para nossos programas: o **escopo global** e o **escopo de uma função**.

Novamente, caímos aqui na questão da indentação.

```
1
2 print "Este aqui eh o escopo global"
3
4 def testador_primo(possivel_primo):
5     m_possivel_primo = int((possivel_primo)/2) # Uma otimizacao: Testa apenas metade do conjunto
6     for cont in range(2,m_possivel_primo,1): # Inicia em 2
7         if(possivel_primo % cont == 0): # Se o resto da divisao eh 0, significa que o numero dividiu, logo, nao eh primo
8             return False
9     return True
10
11 def buscaprime(valor):
12     print """Esta funcao testa se valores são primos
13     (divisivel somente por 1 e ele mesmo)."""
14     if(limite < 2):# Solucao elegante
15         return
16     res = testador_primo(int(valor))
17     if(res):
18         print "EH PRIMO"
19     else:
20         print "NAO EH PRIMO"
21
22 #Multiplas chamadas
23 buscaprime(7)
24 buscaprime(2)
25 buscaprime(6)
26 buscaprime(15557)
27
```

Parâmetros de uma função

É dito que funções possuem um “namespace próprio”. Na prática, isto significa que, definições dentro do bloco da função podem vir a “sobrepor” definições globais do programa. Da mesma forma, uma variável criada dentro de uma função existe unicamente no escopo desta função. Observe:

```

1  palavra_magica = "Alakazam" # Escopo Global
2
3  def magia ():
4      palavra_magica = "Abacadabra" # Mesmo nome, porem escopo diferente
5      print palavra_magica
6
7      print "\nDo escopo da Funcao"
8      magia()
9
10     #Resultados diferentes, escopos diferentes
11
12     print "\nDo Global:"
13     print palavra_magica
14
15
16  def outra_magia():
17      print palavra_magica
18
19      # Eh aqui que a magia acontece!
20      print "\nE aqui, o Python fazendo esforco: "
21      outra_magia()
22
23      palavra_magica = "Abracapocus"
24
25      outra_magia()

```



Existe porém, uma maneira de se passar elementos para o escopo de uma função. Isto é feito através dos parâmetros (neste caso, passados como “argumentos”).

Parâmetros serão tratados, dentro do escopo da função, como variáveis comuns. Um parâmetro que carregue uma string, por exemplo, pode ser impresso através de “print”. Não há restrições aqui, quanto ao tipo. Um exemplo:

```
1 def magia_avancada(palavra_magica): # "palavra_magica" aqui eh um PARAMETRO da funcao
2     print "Hocus " + palavra_magica # variavel esta "disponivel pelo nome do parametro" agora
3
4     palavra = "Abacadabra" # Criamos a palavra em outro escopo, nesse caso, Global
5
6     magia_avancada(palavra) #passamos como ARGUMENTO "palavra" a funcao magia avancada
```

Também trabalhamos com listas, dicionários e etc. em funções:

```
1 lista_magias = ["Abacadabra", "Babacadabra", "Hocus Pocus", "Tome Cevadis", "Apalavris Sensintidus"]
2
3 def super_magia(magias):
4     for magia in magias:
5         print "Atacando com: ", magia
6
7     super_magia(lista_magias)
8
9     print "\n\n\n"
10
11     # Um exemplo mais pratico
12 def busca_binaria(lista_valores, inicio, fim, valor):
13     meio = int((inicio + fim)/2)
14     print meio
15     if(valor == lista_valores[meio]):
16         print "Encontrado", lista_valores[meio]
17         return
18     elif(meio == fim or meio == inicio):
19         print "Fim de execucao, Valor Inexistente no conjunto"
20         return
21     elif(valor < lista_valores[meio]):
22         busca_binaria(lista_valores, inicio, meio-1, valor)
23         return
24     elif(valor > lista_valores[meio]):
25         busca_binaria(lista_valores, meio+1, fim, valor)
26         return
27     else:
28         print "Valor Inexistente no conjunto"
29         return
30
31 lista_a_testar = [1,2,3,5,8,13,21]
32
33 busca_binaria(lista_a_testar, 0, len(lista_a_testar), 13)
```

Em python, podemos trabalhar com valores de escopos distintos (por referência). Definições à parte (AEDs 2) isto significa que, na prática, se algum valor for alterado dentro da função, a alteração também ocorre fora da função, se tal variável estiver definida num escopo “mais externo”.

```
1 valor = 10
2
3 def cinco():
4     valor = 5
5
6     cinco()
7     print valor # 10
8
9 def vinte():
10     global valor
11     valor = 20
12
13 vinte()
14 print valor # 20
15 # Trabalhar diretamente com globais eh uma pratica ruim de programacao
```


Por padrão, todos os valores em python são passados por referência. Nesse caso, “o que for feito dentro de uma função também será feito do lado de fora”, para o caso de valores passados por parâmetro.

```
1 lista_val = [16, 26, 36, 46]
2 def muda (lista):
3     lista.append(56)
4     print lista_val
5     muda(lista_val)
6     print lista_val # Contem o valor 56
7
```

Tipos de argumento de função

Python permite diversos formatos para passagem de parâmetros como argumentos, cada qual com uma funcionalidade.

- Argumentos Obrigatórios (Requeridos):

```
def cumprimentar_requerido (nome, idade): # Argumentos requeridos
    print nome, " meu grande amigo!"
```

- Argumentos Palavra-Chave:

```
def cumprimentar_keyword (nome, idade): # Argumentos palavra chave
    #Sintaticamente, a funcao eh declarada da mesma forma que com args. requeridos
    print nome, " meu grande amigo!"
```

- Argumentos Padrão:

```
def cumprimentar_padrao(nome = "Joao Amorim", idade = 60): # Argumentos padrao
    #Estes valores serao padrao em qualquer chamada
    print nome, " meu grande amigo!"
```

- Argumentos de tamanho de variável:

```
def cumprimentar_tamvar(*nome): # Argumentos de tamanho de variavel
    #Nome sera uma tupla, tao grande quanto se queira
    print nome, " meu(s) grande(s) amigo(s)!"
```

```

1
2 def cumprimentar_requerido (nome, idade): # Argumentos requeridos
3     print nome, " meu grande amigo!"
4
5 def cumprimentar_padrao(nome = "Joao Amorim", idade = 60): # Argumentos padrao
6     #Estes valores serao padrao em qualquer chamada
7     print nome, " meu grande amigo!"
8
9 def cumprimentar_tamvar(*nome): # Argumentos de tamanho de variavel
10    #Nome sera uma tupla, tao grande quanto se queira
11    print nome, " meu(s) grande(s) amigo(s)!"
12
13 def cumprimentar_keyword (nome, idade): # Argumentos palavra chave
14    #Sintaticamente, a funcao eh declarada da mesma forma que com args. requeridos
15    print nome, " meu grande amigo!"
16
17
18 cumprimentar_requerido("Irineu", 36) # Chamada
19 cumprimentar_padrao("Schwarzenegger", 22) # Chamada
20 cumprimentar_padrao() # Utiliza dos valores definidos por padrao
21 cumprimentar_tamvar("Fulano", "Beltrano") # Chamada com mais de um valor associado ao parametro
22 cumprimentar_keyword(idade = 40, nome = "Mikkelsen") # Por palavras chave, os argumentos podem estar em qualquer ordem

```

```

antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBasico/Slides/Exemplos/Funcoes$ python argumentos.py
Irineu meu grande amigo!
Schwarzenegger meu grande amigo!
Joao Amorim meu grande amigo!
('Fulano', 'Beltrano') meu(s) grande(s) amigo(s)!
Mikkelsen meu grande amigo!

```

Exercício #16 (5 min.)

Crie uma nova função. A mesma deve:

- Receber por parâmetro, dois valores numéricos.
- Imprimir a soma, subtração, divisão e multiplicação do primeiro pelo último.

Atenção especial na divisão, pois a função deve tratar da questão da divisão por 0 imprimindo um “erro”.

Retorno de uma função

Após todas as instruções executadas dentro da função, a mesma pode ou não **retornar** algum valor.

Retornar, neste caso, significa passar algum valor significativo para o fluxo do programa onde houve a chamada da função. Esta operação é feita, através da palavra reservada “return”.

Note que, para que haja “certa usabilidade” no retorno da função, muitas vezes o mesmo deverá ser atribuído à uma variável, o que ocorre na chamada da função

- Eis um exemplo simples, de função com retorno.

```
1 def exterminador(municao):
2     while(municao > 0):
3         print "Disparando..."
4         municao -= 1
5     return "Eu vou voltar..."
6
7 """ Keyword return, indica que algo sera "retornado" para o fluxo que
8 originou a chamada desta funcao"""
9
10 fala_personagem = exterminador(10)
11
12 print "Fala: ", fala_personagem
```



Comparando a sintaxe de funções

- Em Python:

```
def soma_dois_numeros (a, b):  
    print "Somando: ", a, " + ", b  
    return (int(a) + int(b))
```

- Em C++:

```
int soma_dois_numeros (int a, int b){  
    cout<<"Somando: "<<a<<" + "<<b<<endl;  
    return a+b;  
}
```


Você (não) pode deixar “PASS”ar!

Quando começamos à abordar funções, nosso primeiro exemplo fazia uso de uma palavra reservada “pass”.

Tal comando serve para dizer ao programa, simplesmente, “não faça nada”.

Serve para adicionar mais legibilidade à alguns códigos, bem como elucidar para os programadores, trechos do código que ainda estão em desenvolvimento.

```
1 def ainda_nao_fiz_nada_mas_vou_fazer():  
2     pass
```

Exercício #17 (15 min.)

Dada a lista de dicionários:

```
[{'Nome': 'John', 'Sobrenome': 'Hammond', 'Profissao': 'Empreendedor'},  
{ 'Nome': 'John', 'Sobrenome': 'Rambo', 'Profissao': ' Militar'},  
{ 'Nome': 'John', 'Sobrenome': 'Wick', 'Profissao': 'Veterinário'},  
{ 'Nome': 'John', 'Sobrenome': 'McClane', 'Profissao': 'Detetive'},  
{ 'Nome': 'John', 'Sobrenome': 'Snow', 'Profissao': 'Patinador'}  
]
```

- Crie uma função para imprimir os dados de uma pessoa, caso o nome dela seja “John”;
- Crie duas funções, uma específica para adicionar e outra para remover itens (dicionários) da lista. A adição deve ser feita na forma de dicionário, com nome, sobrenome e profissão, tal qual os outros. Já a remoção, pode ser feita pelo índice. Adicione 2 pessoas e remova 1.

Dúvidas?
Sugestões?
Críticas?