Programação de Computadores

LAÇOS E LISTAS

Renato Dourado Maia
Instituto de Ciências Agrárias
Universidade Federal de Minas Gerais

Exemplo Inicial – Uma Tabela

- Suponha que você deseja construir uma tabela com valores de temperatura em graus Celsius e Fahrenheit:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  5  41.0
 10  50.0
 15  59.0
 20  68.0
 25  77.0
 30  86.0
 35  95.0
 40 104.0
```

Como fazer?

Exemplo Inicial – Uma Tabela

- Para **uma linha** da tabela:

```
C = -20
F = 9.0/5*C + 32
print C, F
```

- Solução **ingênua**:

```
C = -20; F = 9.0/5*C + 32; print C, F
C = -15; F = 9.0/5*C + 32; print C, F
…
C = 35; F = 9.0/5*C + 32; print C, F
C = 40; F = 9.0/5*C + 32; print C, F
```

Exemplo Inicial – Uma Tabela

Programa

```
 1  C = -20;   F = 9.0/5*C + 32;   print C, F
 2  C = -15;   F = 9.0/5*C + 32;   print C, F
 3  C = -10;   F = 9.0/5*C + 32;   print C, F
 4  C =  -5;   F = 9.0/5*C + 32;   print C, F
 5  C =   0;   F = 9.0/5*C + 32;   print C, F
 6  C =   5;   F = 9.0/5*C + 32;   print C, F
 7  C =  10;   F = 9.0/5*C + 32;   print C, F
 8  C =  15;   F = 9.0/5*C + 32;   print C, F
 9  C =  20;   F = 9.0/5*C + 32;   print C, F
10  C =  25;   F = 9.0/5*C + 32;   print C, F
11  C =  30;   F = 9.0/5*C + 32;   print C, F
12  C =  35;   F = 9.0/5*C + 32;   print C, F
13  C =  40;   F = 9.0/5*C + 32;   print C, F
```

Saída (FEIA!)

```
>>>
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
>>>
```

Exemplo Inicial – Uma Tabela

- A solução apresentada no *slide* anterior:
  - É **tediosa** de programar.
  - É **muito** suscetível a **erros** de digitação.
- Quando a programação fica **muito tediosa**, provavelmente há algum **recurso** para **simplificar** a codificação.
- Já vimos que o computador é **excelente** para realizar **tarefas repetitivas**.
  - Para fazer o computador realizar **tarefas repetitivas**, são utilizados **laços** (ou **estruturas/comandos de repetição**).

O Laço `while`

- Um laço `while` executa **repetidamente** um conjunto de comandos **enquanto** uma **condição booleana** for **verdadeira**:

```
while condição:
    <comando 1>
    <comando 2>
    …
<primeiro comando após o laço>
```

- Todos os comandos do laço dever estar **endentados**!
- O laço **termina** quando um comando **não endentado** é encontrado

O Laço `while` – Construindo uma Tabela

```
1  print '-------------------'      # table heading
2  C = -20                        # start value for C
3  dC = 5                         # increment of C in loop
4  while C <= 40:                 # loop heading with condition
5      F = (9.0/5)*C + 32         # 1st statement inside loop
6      print C, F                 # 2nd statement inside loop
7      C = C + dC                 # 3rd statement inside loop
8  print '-------------------'      # end of table line (after loop)
```

Vamos simular o laço “no braço”!

Estruturas de Dados

- **Estruturas de dados** são maneiras de **organizar dados** de maneira a **facilitar** o seu **acesso**.
- Algumas formas são **clássicas**:
  - Listas.
  - *Arrays* (vetores e matrizes).
  - Tuplas (registros).
  - Árvores.
- Linguagens de programação frequentemente possuem **primitivas** para a construção dessas estruturas de dados (estruturas de dados **embutidas**).

Estrutura de Dados Abstrata

- É uma **especificação matemática** que define uma **coleção de dados** e uma série de **operações** que podem ser realizadas sobre ela.
- É **abstrata** por não especificar **como** as operações são feitas, mas somente os **dados de entrada** e o **resultado**.
- Numa linguagem de programação, essa coleção de operações é chamada de **interface** ou **API** (***Application Programming Interface***).
- Usuários devem se **preocupar** com a **interface** e **não** com a **implementação**, que **pode mudar** como tempo.

Listas

- Considerando o que estudamos **até agora**, uma variável pode se referir a um **número** ou a uma ***string***.
- E se for necessário trabalhar com uma **coleção de números**?
- Uma solução **simples**, mas **tediosa** e **sem nenhuma elegância**, é utilizar **uma variável para cada valor**.
  - Isso é **impraticável** para uma **grande quantidade** de valores!
- Solução **elegante**: utilizar uma **lista**!

Listas (Clássicas)

- São **arranjos sequenciais** de informações mais simples.
- Caracterizam-se por permitir o **acesso eficiente** aos seus elementos em **ordem sequencial**.
- A **definição clássica** de uma lista como estrutura de dados abstrata compreende as seguintes **operações**:
  - Construir de uma **lista vazia**.
  - **Testar** se uma dada lista é vazia.
  - Obter o **primeiro elemento** de uma lista.
  - **Adicionar** um novo elemento no **início** de uma lista.
  - **Retirar** o elemento **inicial** de uma lista.

Listas em Python

- A estrutura conhecida como lista em Python é **mais geral** do que a estrutura de dados abstrata lista **clássica**:
  - Pode ser vista como uma implementação **tanto de lista como de *arrays***, suportando, além de **acesso sequencial**, o **acesso direto** por meio de **índices**.
- Listas e ***strings*** são **variedades de sequências** e, portanto, têm APIs **semelhantes**:
  - Podem ser **indexadas** e **fatiadas**.
  - Podem ser **concatenadas** (+) e **repetidas** (*).
  - É Importante destacar que os elementos de **listas podem** ser alterados **individualmente**, enquanto de ***strings***, **não**.

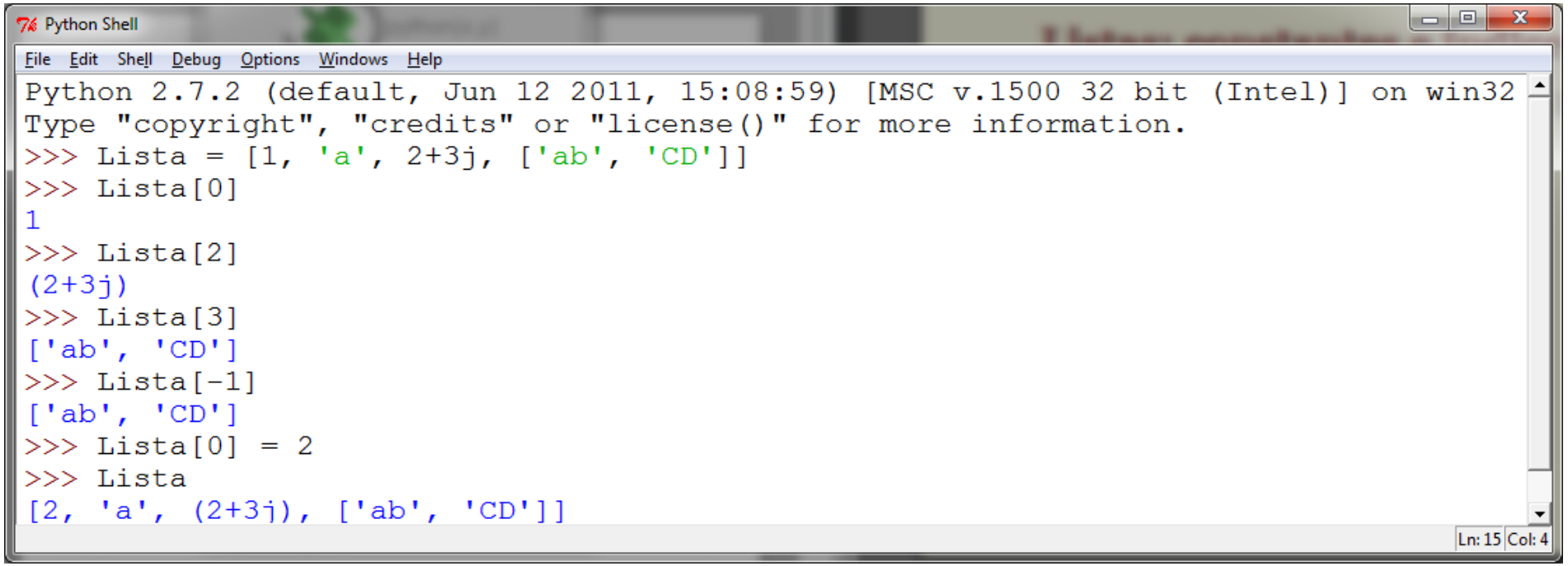
Listas em Python

- **Listas** constituem o tipo de **agregação de dados** mais **versátil** e **comum** da linguagem Python:
  - Podem, por exemplo, ser utilizadas para implementar estruturas de dados mais complexas, como **matrizes** e **árvores**.
- Uma lista é **inicializada** separando-se os elementos com uma **vírgula** e utilizando-se **colchetes**:

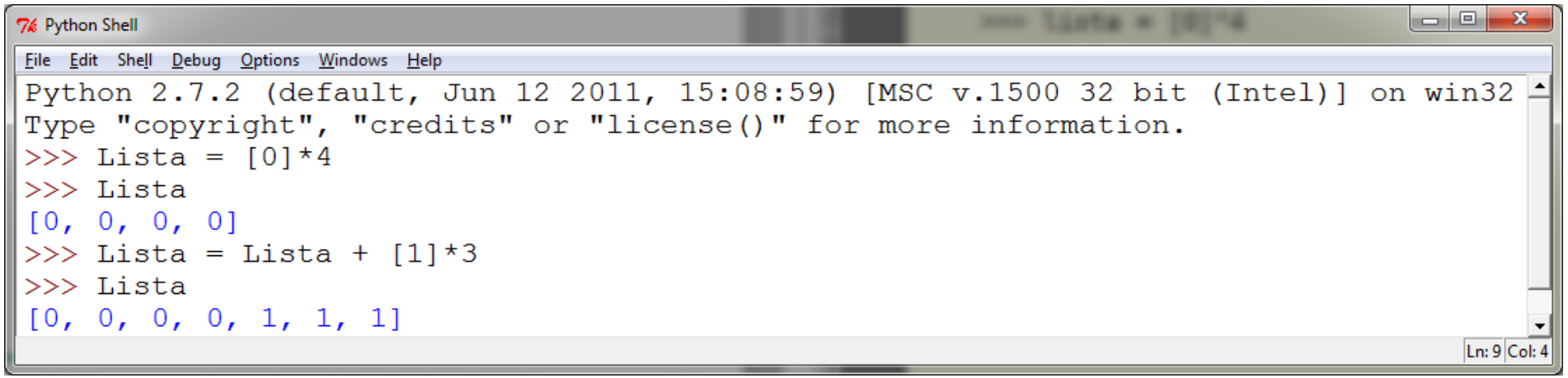
```
L1 = [-91, 'uma string', 7.2, 0]
```

- Os elementos, que podem ser de **qualquer tipo** (inclusive **outras listas**), podem ser acessados utilizando-se um **índice** (tal como vimos para ***strings***).

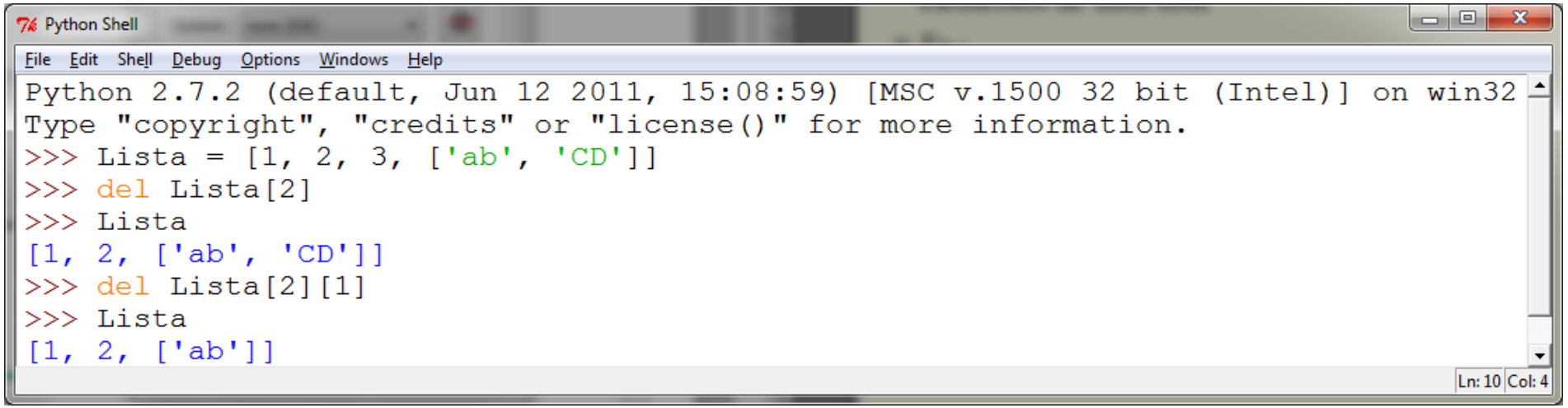
Listas – Inicialização e Índices



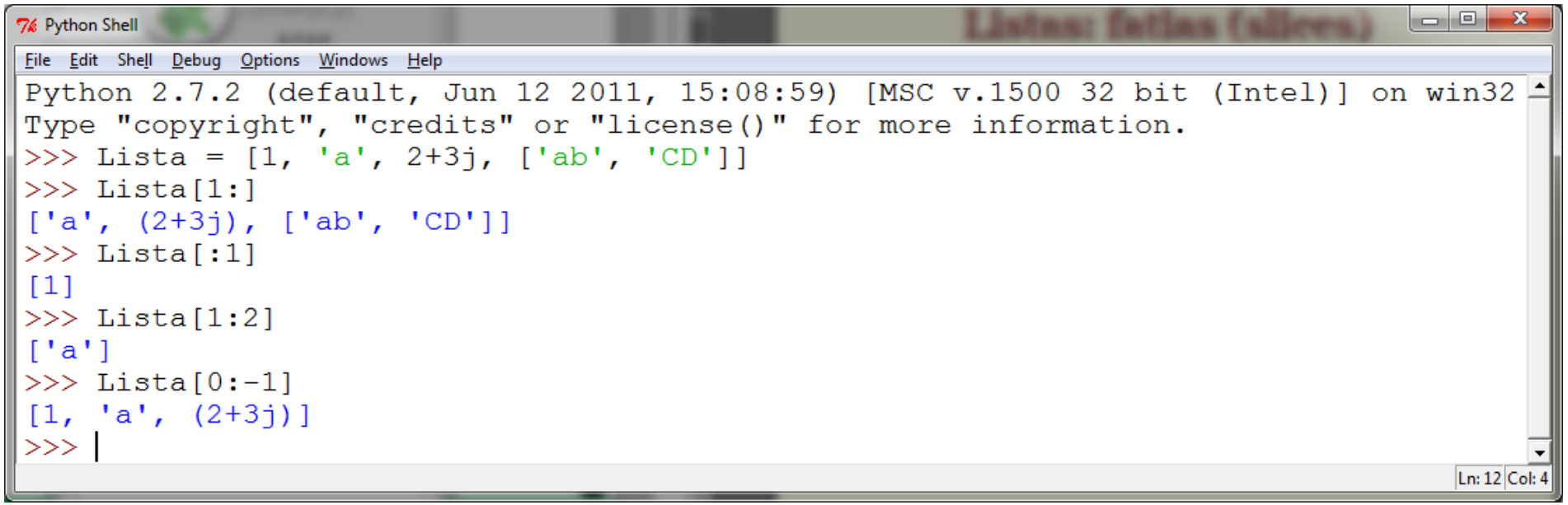
Listas – Concatenação e Repetição



Listas – Remoção

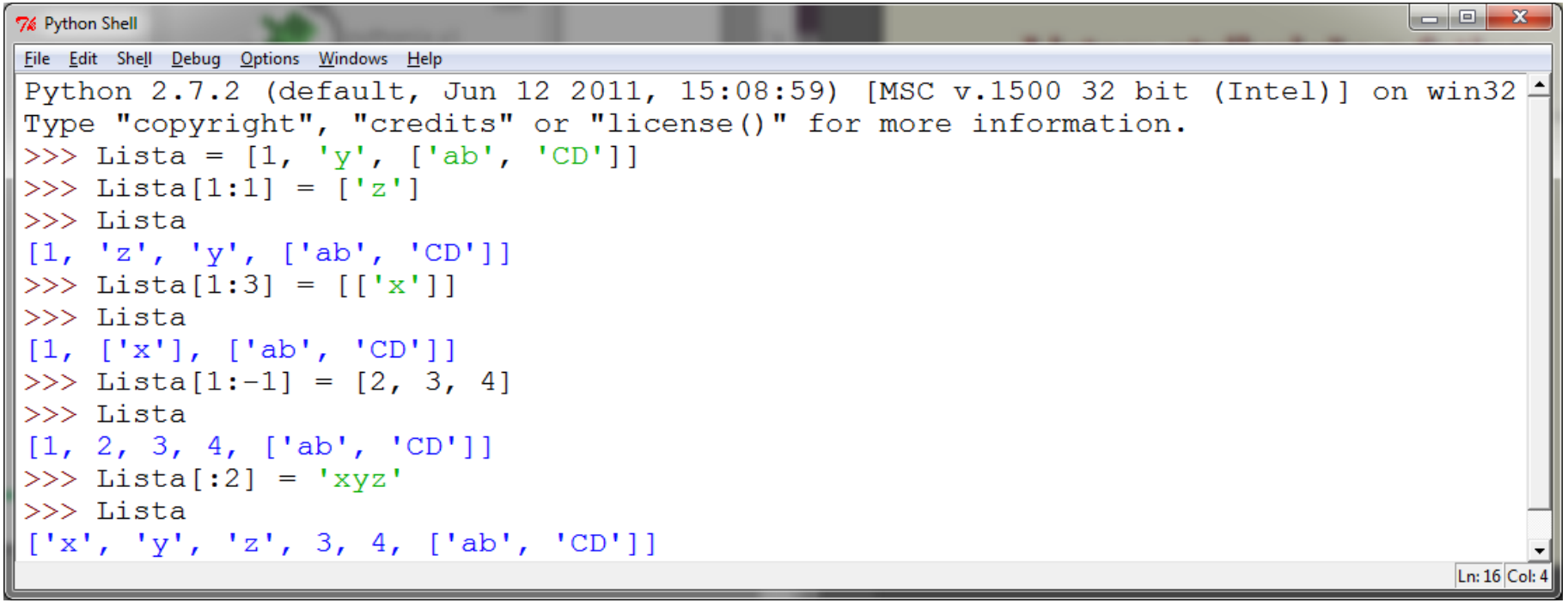


Listas – Fatiamento



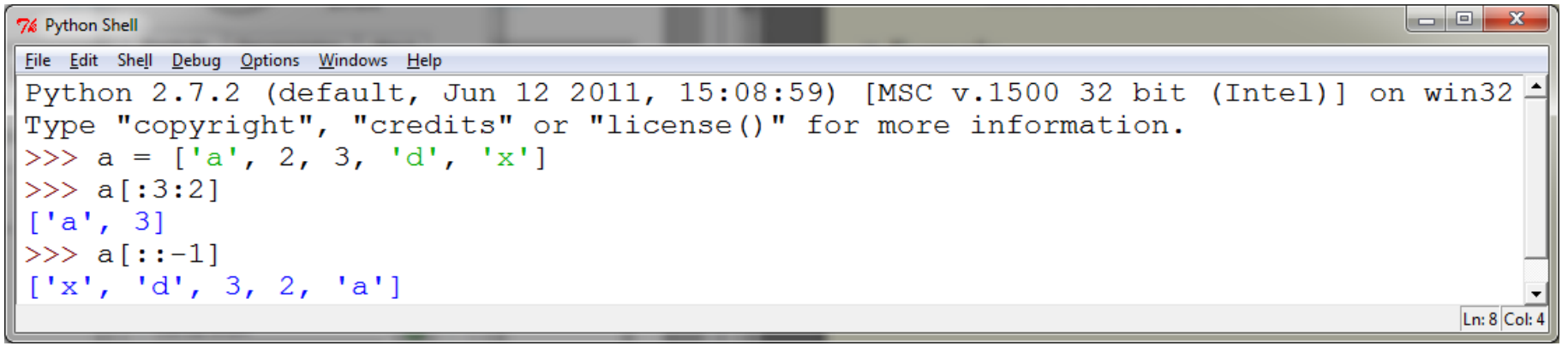
Listas – Atribuição a Fatias

- A **atribuição a uma fatia** exige que o valor atribuído seja uma **sequência** (lista ou *string*, por exemplo):

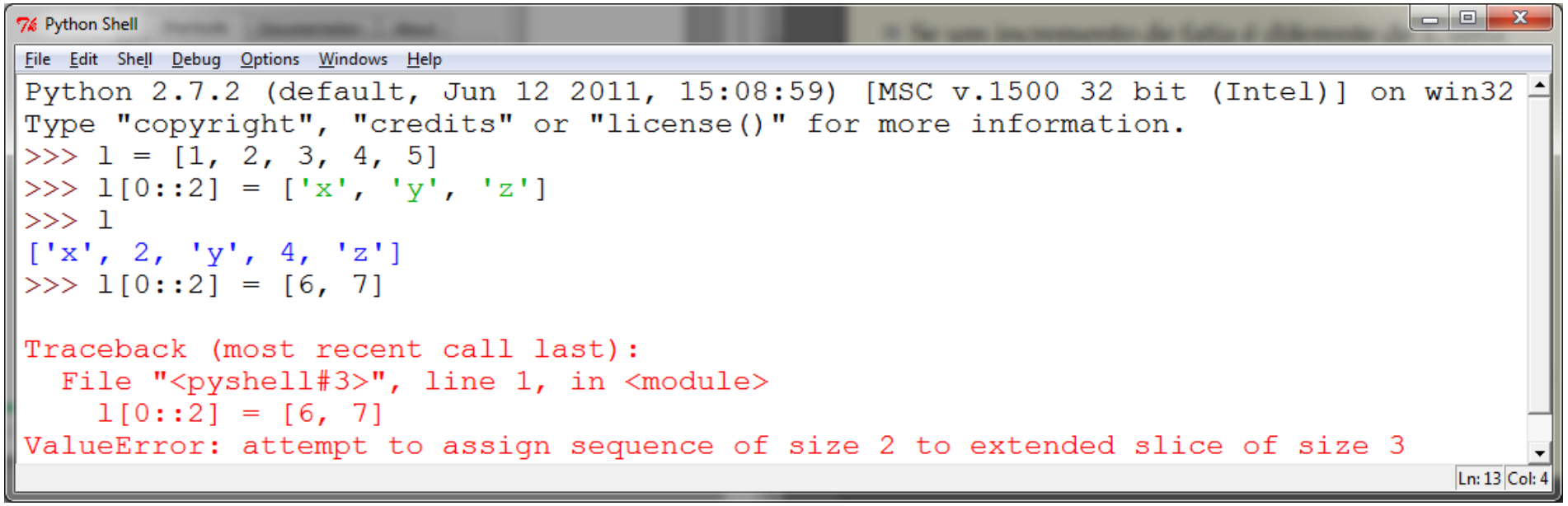


Listas – Incrementos em Fatias

- É possível utilizar um **terceiro número** na notação de **fatias** designando o **incremento**. O valor ***default*** é 1 e pode-se utilizar qualquer número **inteiro diferente de 0**:
  - `a[0:10:2]` retorna uma lista com os 10 primeiros elementos de `a`, tomados de 2 em 2.
  - `a[5:0:-1]` retorna uma lista com os 5 primeiros elementos de `a`, tomados da esquerda para a direita.

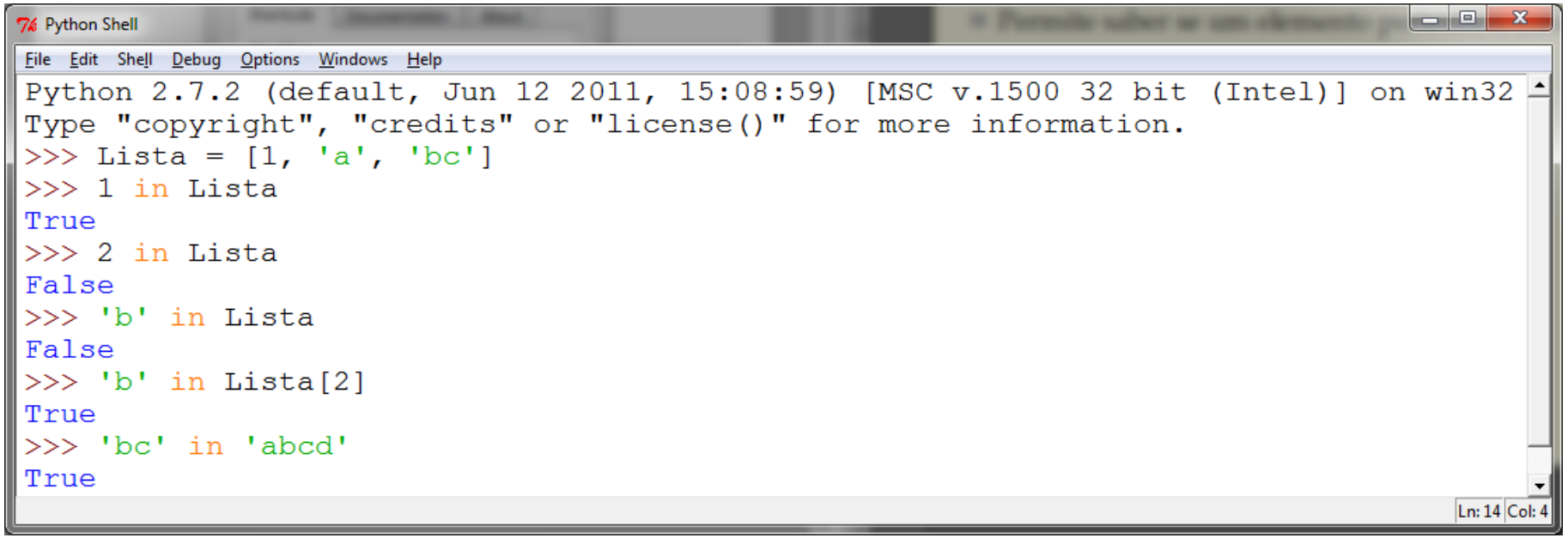


Listas – Incrementos em Fatias



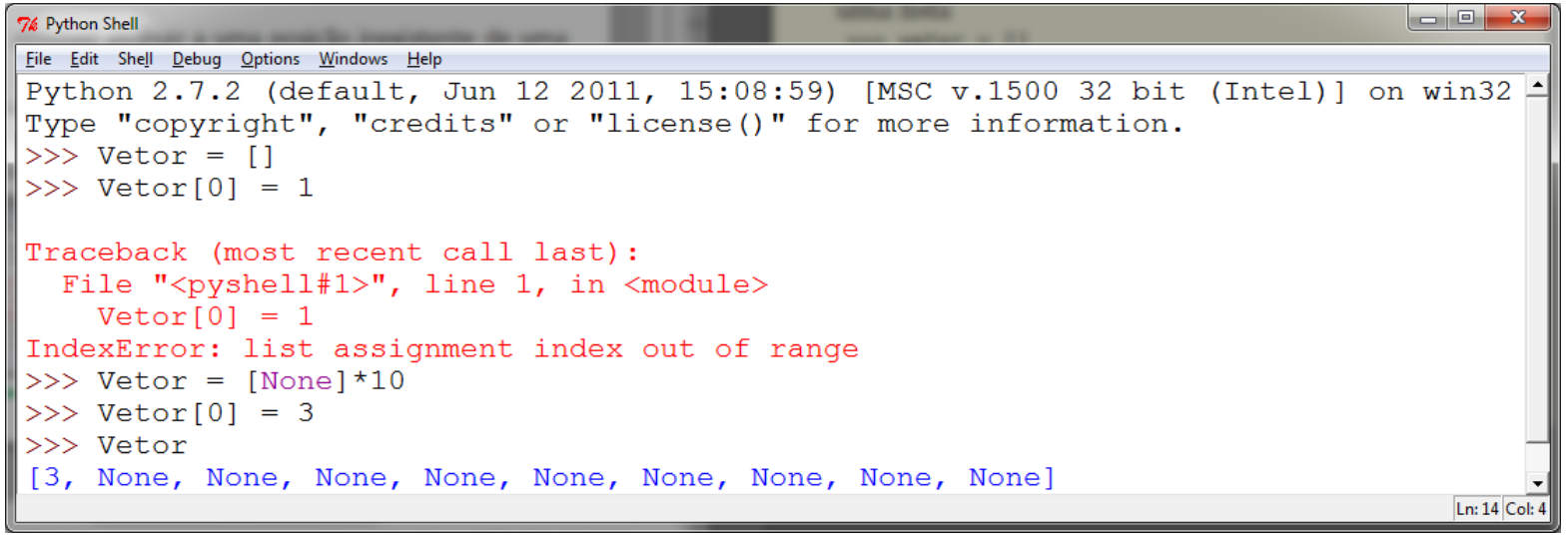
Operador `in`

- Permite saber se um elemento **pertence** a uma lista e serve também para ***strings***.

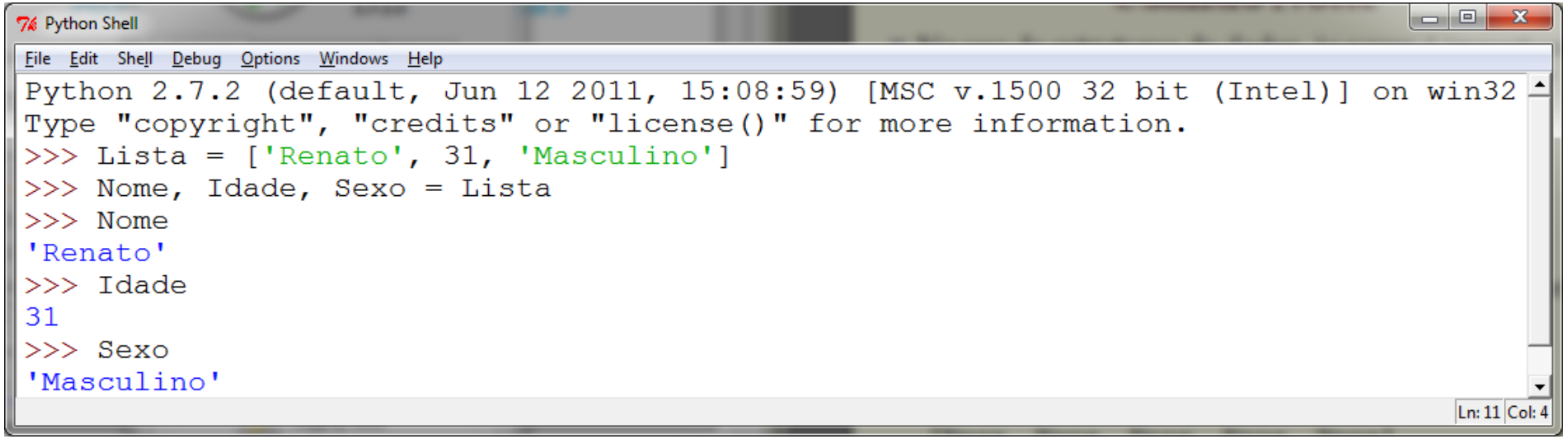


Listas como *Arrays*

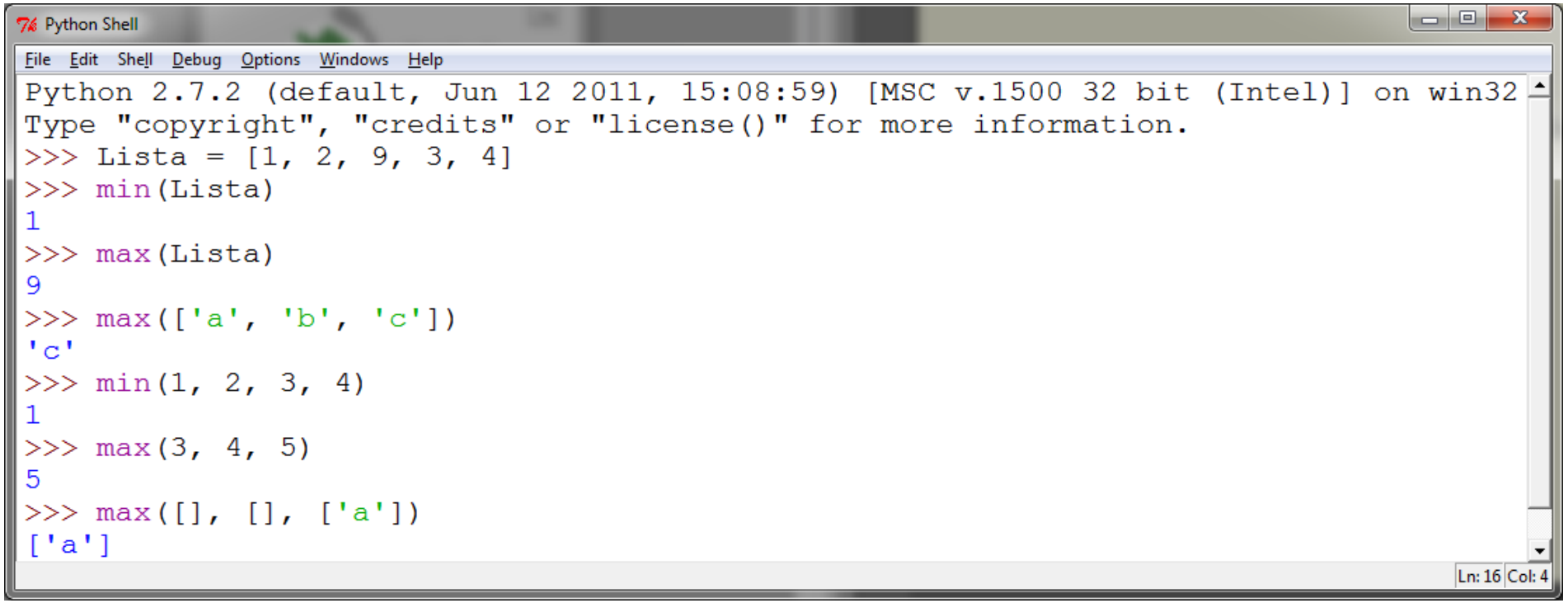
- Não é possível atribuir a uma **posição inexistente** de uma lista.
- Se uma lista vai ser utilizada como um ***array***, ou seja, vai conter um **número conhecido** de elementos, é **conveniente** inicializá-la.



Atribuição a partir de Listas



Funções `len`, `min` e `max`



O Laço `for`

- Cada elemento de uma lista pode ser **visitado** e **processado** utilizando-se um laço `for`:

```
1  degrees = [0, 10, 20, 40, 100]
2  for C in degrees:
3      print 'list element:', C
4  print 'The degrees list has', len(degrees), 'elements'
```

Endentação!

O Laço `for` – Construindo uma Tabela

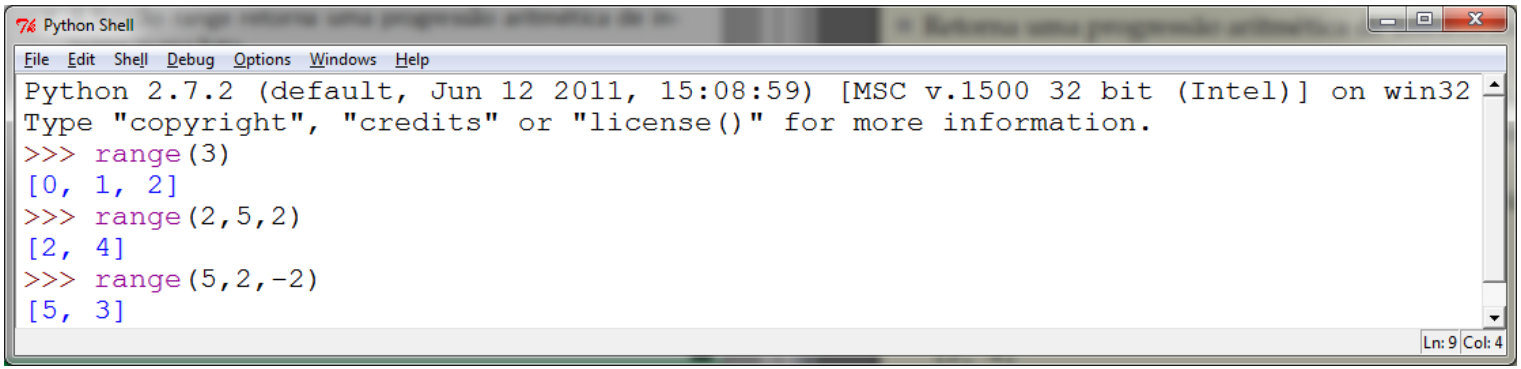
```
1  Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2  for C in Cdegrees:
3      F = (9.0/5)*C + 32
4      print '%5d %5.1f' % (C, F)
```

```
1  Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2  index = 0
3  while index < len(Cdegrees):
4      C = Cdegrees[index]
5      F = (9.0/5)*C + 32
6      print '%5d %5.1f' % (C, F)
7      index += 1
```

```
1  Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2  Fdegrees = [] # start with empty list
3  for C in Cdegrees:
4      F = (9.0/5)*C + 32
5      Fdegrees.append(F) # add new element to Fdegrees
6  print Fdegrees
```

A Função `range`

- A função `range` retorna uma **progressão aritmética de inteiros numa lista (em Python 2)**.
- `range(início, parada, incremento)`:
  - Início (opcional) é o **primeiro valor** a ser gerado (*default*: 0).
  - Parada é o **limite da progressão** (a progressão termina no **último valor antes de parada**).
  - Incremento (opcional) é o **passo** da progressão (*default*: 1).



A Função `range`

```
1  Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2  Fdegrees = [] # start with empty list
3  for C in Cdegrees:
4      F = (9.0/5)*C + 32
5      Fdegrees.append(F) # add new element to Fdegrees
6  print Fdegrees
```

```
1  Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2  Fdegrees = [] # start with empty list
3  for i in range(0, len(Cdegrees), 1):
4      C = Cdegrees[i]
5      F = (9.0/5)*C + 32
6      Fdegrees.append(F) # add new element to Fdegrees
7  print Fdegrees
```

Listas – Geração Compacta

- Forma Geral (*list comprehension*):

```
Lista2 = [expressão for elemento in Lista1]
```

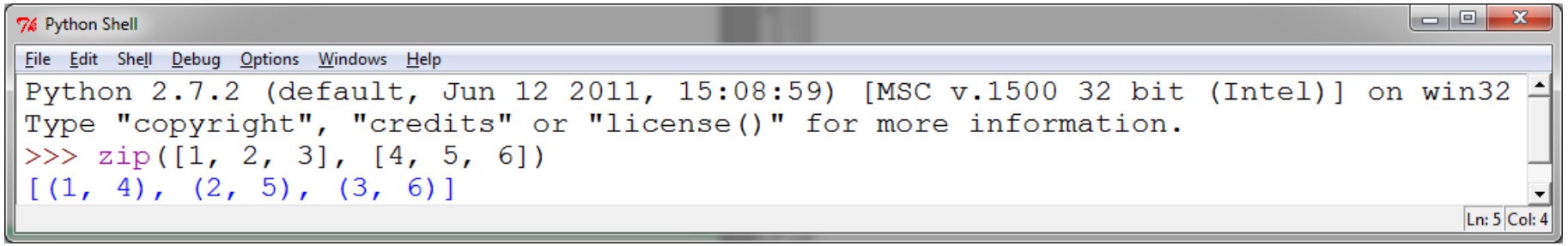
```
1  n = 16
2  Cdegrees = []; Fdegrees = [] # empty lists
3  for i in range(n):
4      Cdegrees.append(-5 + i*0.5)
5      Fdegrees.append((9.0/5)*Cdegrees[i] + 32)
```

```
1  n = 16
2  Cdegrees = [-5 + i*0.5 for i in range(n)]
3  Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
```

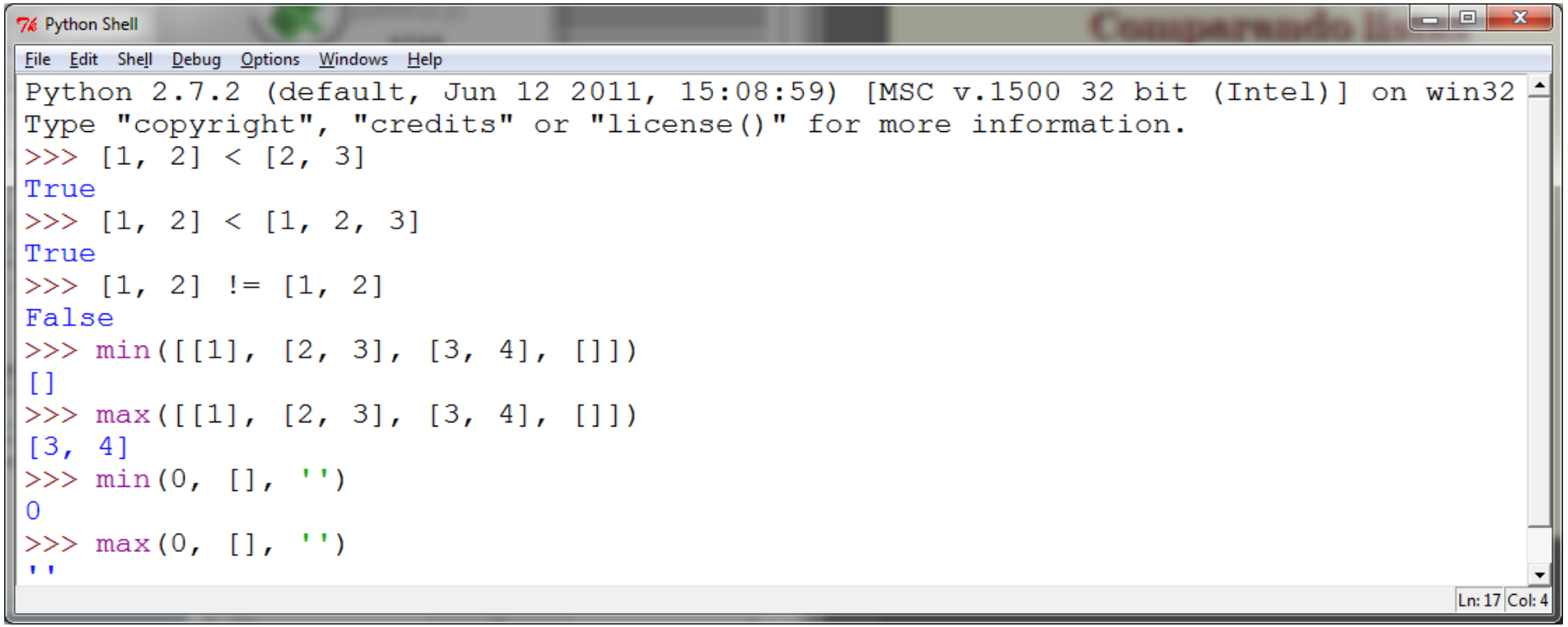
Ser Pythônico...

```
1 for i in range(len(Cdegrees)):
2     print Cdegrees[i], Fdegrees[i]
```

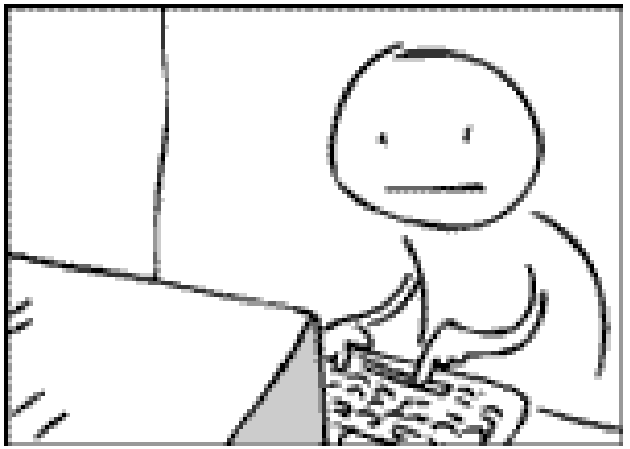
```
1 for C, F in zip(Cdegrees, Fdegrees):
2     print C, F
```



Listas – Comparação



Não se Esqueçam de Praticar!



MAS APRECIEM COM MODERAÇÃO!

Por Hoje é Só!