



Minicurso de Python Básico

Ana Lívia S. S. Almeida,
Guilherme Giordano P. Guimarães

10 de março de 2013

Introdução à Linguagem Python

O que é Python?

Python é uma linguagem de altíssimo nível, **orientada a objetos**, de **tipagem dinâmica** forte, interativa e interpretada. Possui uma sintaxe clara e concisa, que favorece a legibilidade do código, tornando a linguagem mais produtiva. É possível integrar o Python a outras linguagens como C e Fortran.

Por que usar Python?

- Fácil e simples
- Sintaxes limpas
- Diversas bibliotecas inclusas
- Mais expressiva do que muitas linguagens (C/C++, Perl, Java)
- Interativa
- Protótipos rápidos
- Alta produtividade
- Interfaces para outras linguagens como C/C++ e Fortran

Informações sobre o Python

- Os arquivos são identificados pela extensão “.py” (arquivo.py)
- Não há declaração de tipo de variáveis, nem abertura e fechamento de chaves. Porém, a indentação é extremamente importante!!
- Aceita os tipos de dados básicos (int, float, long int, boolean, char...)
- Para comentar uma linha, basta inserir “#” no início da linha
- Variáveis no interpretador Python são criadas através da atribuição e destruídas pelo coletor de lixo (garbage collector), quando não existem mais referências a elas.
- Nomes das variáveis devem começar com letra (sem acentuação) ou sublinhado (_) e seguido por letras (sem acentuação), dígitos ou sublinhados (_), sendo que maiúsculas e minúsculas são consideradas diferentes.

Tipos de Dados

Além disso, existem tipos que funcionam como coleções. Os principais são:

- Lista
- Tupla
- Dicionário

Os tipos no Python podem ser

- Mutáveis: permitem que os conteúdos das variáveis sejam alterados
- Imutáveis: não permitem que os conteúdos das variáveis sejam alterados

Tipos - números

Python oferece alguns tipos numéricos na forma de builtins:

- Inteiro (int): $i = 1$
- Real de ponto flutuante (float): $f = 3.14$
- Complexo (complex): $c = 3 + 4j$

```
# -*- coding: latin1 -*-  
  
# Convertendo de real para inteiro  
print 'int(3.14) =', int(3.14)  
  
# Convertendo de inteiro para real  
print 'float(5) =', float(5)  
  
# Calculo entre inteiro e real resulta em real  
print '5.0 / 2 + 3 = ', 5.0 / 2 + 3  
  
# Inteiros em outra base  
print "int('20', 8) =", int('20', 8) # base 8  
print "int('20', 16) =", int('20', 16) # base 16  
  
# Operações com números complexos  
c = 3 + 4j  
print 'c =', c  
print 'Parte real:', c.real  
print 'Parte imaginária:', c.imag  
print 'Conjugado:', c.conjugate()
```

Tipos - números

Saída:



```
float(5) = 5.0
5.0/2+3 = 5.5
int(20,8) = 16
int(20,16) = 32
c = (3+4j)
Parte Real: 3.0
Parte Imaginária: 4.0
Conjugado: (3-4j)
```


Operações aritméticas:

- Soma (+)
- Diferença (-)
- Multiplicação (*)
- Divisão (/): entre dois inteiros funciona igual à divisão inteira.

Em outros casos, o resultado é real

- Divisão inteira (//): o resultado é truncado para o inteiro imediatamente inferior, mesmo quando aplicado em números reais, porém neste caso o resultado será real também
- Módulo (): retorna o resto da divisão.
- Potência (**): pode ser usada para calcular a raiz, através de expoentes fracionários (exemplo: $100 ** 0.5$)
- Positivo (+)
- Negativo (-)

Operações lógicas:

- Menor ($<$)
- Maior ($>$)
- Menor ou igual (\leq)
- Maior ou igual (\geq)
- Igual ($==$)
- Diferente (\neq)

Tipos - Textos

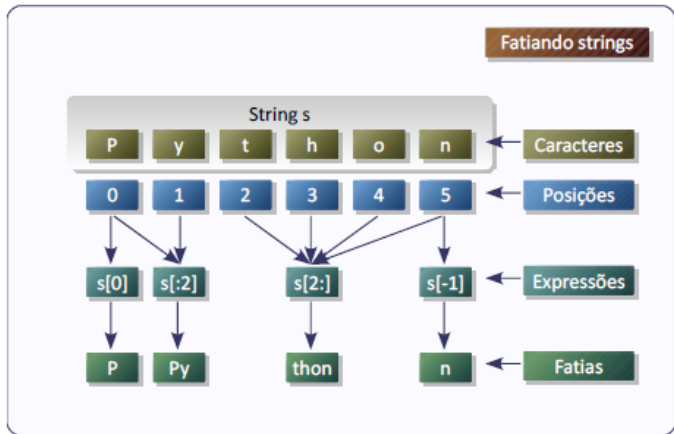
A inicialização de strings pode ser:

- Com aspas simples ou duplas
- Em várias linhas consecutivas, desde que seja entre três aspas simples ou duplas
- Sem expansão de caracteres (exemplo: `s = r'\n'`, em que `s` conterá os caracteres “\” e “n”)

```
>>> "hello" + "world"      # concetenacao
'helloworld'
>>> "hello"*3              # repeticao
'hellohellohello'
>>> "hello"[0]
'h'
>>> "hello"[-1]            # do final
>>> "hello"[1:4]           # "slicing"
'ell'
>>> len("hello")           # tamanho
5
>>> "hello" < "jello"      # comparacao
True
>>> "e" in "hello"         # busca
True
```

Tipos - Textos

Fatias (slices) de strings podem ser obtidas colocando índices entre colchetes após a string.



Tipos - Textos - Métodos

```
untitled2.py x
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Jul 11 20:53:03 2012
4
5 @author: Dell
6 """
7 print " hello" + "world"    #concatenação
8
9 print "hello"*3 # repeticao
10
11 str = "monty python and the flying circus"
12 print str.split() # uma lista
13 str2=str.split()
14 print str2
15 print str.count("ci") # conta quantas vezes aparece
16 print "/".join(str2)
17
```

Saída

```
helloworld  
hellohellohello  
['monty', 'python', 'and', 'the', 'flying', 'circus']  
1  
monty/python/and/the/flying/circus
```

Tipos - Listas

- Listas são coleções heterogêneas de objetos, que podem ser de qualquer tipo, inclusive outras listas
- As listas no Python são mutáveis, podendo ser alteradas a qualquer momento
- Listas podem ser fatiadas da mesma forma que as strings, mas como as listas são mutáveis, é possível fazer atribuições a itens da lista
- Estruturas de dados nativas: list
- Crescem até o limite da memória
- Métodos para adicionar, remover, ordenar, procurar, contar
- Listas são delimitadas por [e]
- Sintaxe: lista = [a, b, ..., z]

Operações com listas

```
# Uma nova lista: Brit Progs dos anos 70
progs = ['Yes', 'Genesis', 'Pink Floyd', 'ELP']

# Varrendo a lista inteira
for prog in progs:
    print prog

# Trocando o último elemento
progs[-1] = 'King Crimson'

# Incluindo
progs.append('Camel')

# Removendo
progs.remove('Pink Floyd')

# Ordena a lista
progs.sort()

# Inverte a lista
progs.reverse()

# Imprime numerado
for i, prog in enumerate(progs):
    print i + 1, '=>', prog

# Imprime do segundo item em diante
print progs[1:]
```


Operações com listas

Saída:

Yes

Genesis

Pink Floyd

ELP

1 => Yes

2 => King Crimson

3 => Genesis

4 => Camel

['King Crimson', 'Genesis', 'Camel']

Tipos - Tuplas

- Estruturas de dados nativas: tuple
- Coleções de objetos heterogêneos
- Crescem até o limite da memória
- Acesso sequencial, em fatias ou direto
- Métodos para adicionar, remover, ordenar, procurar, contar
- Tuplas são imutáveis, diferentes das listas
- Tuplas são delimitadas por (e)
- Uma tupla é uma coleção de objetos separados por vírgula
- Pode ter ou não parênteses para delimitar a tupla
- Particularidade: tupla com apenas um elemento é representada como: `t1 = (1,)`

Tipos - Tuplas

- Os elementos de uma tupla podem ser referenciados da mesma forma que os elementos de uma lista: `primeiro_elemento = tupla[0]`
- Listas podem ser convertidas em tuplas: `tupla = tuple(lista)`
e tuplas podem ser convertidas em listas: `lista = list(tupla)`
- Embora a tupla possa conter elementos mutáveis, esses elementos não podem sofrer atribuição, pois isto modificaria a referência ao objeto.
- As tuplas são mais eficientes do que as listas convencionais, pois consomem menos recursos computacionais (memória), por serem estruturas mais simples, tal como as strings imutáveis em relação às strings mutáveis.

Operações com tuplas

```
untitled2.py x
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Jul 11 20:53:03 2012
4
5 @author: Dell
6 """
7 t = 12345,54321,'hello!' # ou
8 t = (12345,54321,"hello")
9 t [0]
10 u = t , (1,2,3,4,5) # tuplas podem ser aninhadas
11 print u
12 x,y= u # desempacotar tupla
13 print y
```

Operações com tuplas

Saída:

```
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))  
(1, 2, 3, 4, 5)
```

Existem mais tipos!!

- Dicionários
 - Conjuntos
 - Listas como Pilhas
 - Listas como Filas
 - Funções especiais para listas (map, filter, reduce)
 - Funções lambda
 - Técnicas especiais de iteração em listas
- E muito mais!

Estruturas de Controle:

Controle de Fluxo if-else

É muito comum em um programa que certos conjuntos de instruções sejam executados de forma condicional, em casos como validar entradas de dados, por exemplo.

- Sintaxe:

```
if <condição>:  
    <bloco de código>  
elif <condição>:  
    <bloco de código>  
elif <condição>:  
    <bloco de código>  
else:  
    <bloco de código>
```

Estruturas de Controle:

Controle de Fluxo if-else

Onde:

- <condição>: sentença que possa ser avaliada como verdadeira ou falsa
- <bloco de código>: sequência de linhas de comando
- As clausulas elif e else são opcionais e podem existir vários elifs para o mesmo if, porém apenas um else ao final
- Parênteses só são necessários para evitar ambiguidades

Estruturas de Controle:

Controle de Fluxo if-else

```
temp = int(raw_input('Entre com a temperatura: '))

if temp < 0:
    print 'Congelando...'
elif 0 <= temp <= 20:
    print 'Frio'
elif 21 <= temp <= 25:
    print 'Normal'
elif 26 <= temp <= 35:
    print 'Quente'
else:
    print 'Muito quente!'
```

Estruturas de Controle:

Controle de Fluxo if-else

Saída:

```
Entre com a temperatura 23
Normal
```

Estruturas de Controle:

Laços

- Laços (loops) são estruturas de repetição, geralmente usados para processar coleções de dados, tais como linhas de um arquivo ou registros de um banco de dados, que precisam ser processados por um mesmo bloco de código.

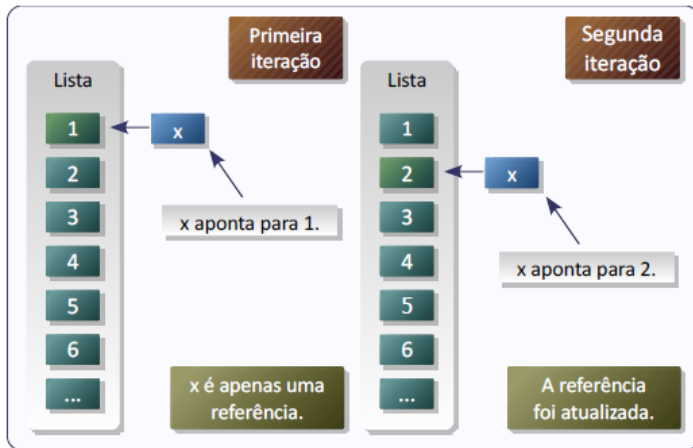
Estruturas de Controle:

Laços - for

- É a estrutura de repetição mais usada no Python. A instrução aceita não só sequências estáticas, mas também sequências geradas por iteradores
- Iteradores são estruturas que permitem iterações, ou seja, acesso aos itens de uma coleção de elementos, de forma sequencial
- Durante a execução de um laço for, a referência aponta para um elemento da sequência
- A cada iteração, a referência é atualizada, para que o bloco de código do for processe o elemento correspondente

Estruturas de Controle:

Laços - for



Estruturas de Controle:

Laços - for

Exemplo:

```
# Soma de 0 a 99  
s = 0  
for x in range(1, 100):  
    s = s + x  
print s
```

Saída:

4950

Estruturas de Controle:

Laços - for

```
untitled2.py x
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Jul 11 20:53:03 2012
4
5 @author: Dell
6 """
7 lst = [10,20,30, ' oi ', ' tchau' ]
8 for i in lst :
9     print i
10 for letra in "python" :
11     print letra
12 for k in range(100):
13     print k
```

A Função range()

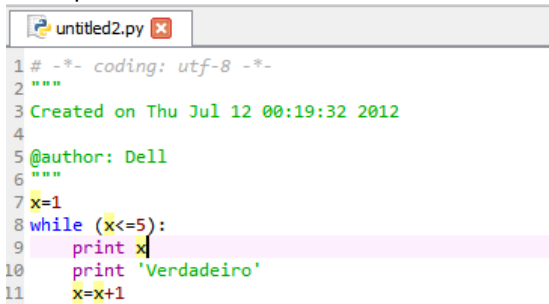
A função `range(m, n, p)`, é muito útil em laços, pois retorna uma lista de inteiros, começando em `m` e menores que `n`, em passos de comprimento `p`, que podem ser usados como sequência para o laço. A função gera uma lista contendo progressões aritméticas

```
>>> range(10)           # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)        # [5, 6, 7, 8, 9]
>>> range(0, 10, 3)     # [0, 3, 6, 9]
>>> range(-10, -100, -30) # [-10, -40, -70, -100]
```


Estruturas de Controle:

while

- Executa um bloco de código atendendo a uma condição
 - O laço while é adequado quando não há como determinar quantas iterações vão ocorrer e não há uma sequência a seguir
- Exemplo:



```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Jul 12 00:19:32 2012
4
5 @author: Dell
6 """
7 x=1
8 while (x<=5):
9     print x
10    print 'Verdadeiro'
11    x=x+1
```

Funções são blocos de código identificados por um nome, que podem receber parâmetros pré-determinados.

No Python, as funções:

- Podem retornar ou não objetos
- Aceitam Doc Strings
- Aceitam parâmetros opcionais (com defaults). Se não for passado o parâmetro será igual ao default definido na função
- Aceitam que os parâmetros sejam passados com nome. Neste caso, a ordem em que os parâmetros foram passados não importa
- Tem namespace próprio (escopo local), e por isso podem ofuscar definições de escopo global
- Podem ter suas propriedades alteradas (geralmente por decoradores)

Funções

```
untitled2.py* x
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Jul 12 00:19:32 2012
4
5 @author: Dell
6 """
7 def par (n):
8     return (n % 2 == 0)
9
10 def mult (x, num=2):
11     return x, x*num
12
13 print par (6) # True
14 a,b = mult (2)
15 print a,b # 2 4
16 a,b = mult (2,10)
17 print a,b
```

Algumas funções úteis:

- `dir()` → lista atributos de um objeto
- `help()` → help interativo ou `help(objeto)`, info. sobre objeto
- `type()` → retorna tipo do objeto
- `raw_input()` → prompt de entrada de dados
- `int()`, `str()`, `float()`... → typecast
- `chr()`, `ord()` → ASCII
- `max()`, `min()` → maior e menor de uma string, lista ou tupla

Procedimentos

```
untitled2.py x
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Jul 12 00:19:32 2012
4
5 @author: Dell
6 """
7 def verifica (n):
8     if(n<0):
9         print'Negativo'
10    elif(n==0):
11        print'Zero'
12    else:
13        print'Positivo'
14    return ''
15
16 print verifica (-5)
17
```

Computação Científica usando Python

Objetivo:

Apresentar os elementos básicos da linguagem Python para escrever programas para solução computacional de problemas científicos, manipular, processar e visualizar os dados

Aplicações:

- Gerar dados (simulação, experimentos)
- Manipular e processar os dados
- Visualizar os resultados
- Para entender, interpretar e validar o que estamos fazendo
- Comunicar os resultados
- Produzir figuras para relatórios e publicações
- Apresentações

NumPy

É uma biblioteca para manipulação de arrays multidimensionais e matrizes.

- Operações rápidas em arrays (funções vetorizadas)
- Diferença com relação a listas tradicionais do Python
 - Vetor homogêneo
 - Muito mais eficientes do que as listas
 - Número de elemento deve ser conhecido a priori. O array pode ser redimensionado posteriormente
 - Muito eficiente (implementado em C)

Criando vetores (arrays) em Numpy

Arrays NumPy podem ser criados a partir de estruturas de dados do Python (listas, tuplas) ou a partir de funções específicas para criação de arrays.

- `zeros((m,n))` → vetor de 0 (zero), com m linhas, n colunas
- `ones((m,n))` → vetor de 1 (um), com m linha e n colunas
- `empty((m,n))` → vetor vazio, com m linhas e n colunas
- `zeros_like(A)` → vetor de 0 (zero), no mesmo formato de A
- `ones_like(A)` → vetor de 1 (um), no mesmo formato de A
- `empty_like(A)` → vetor vazio, no mesmo formato de A

Criando vetores (arrays) em Numpy

- `random.random((m,n))` → vetor $m \times n$ de números aleatórios
- `identity(n)` → matriz identidade de ordem n , ponto flutuante
- `array([[1,2,3],[4,5,6]])` → cria um array a partir de uma lista ou uma tupla
- `arange(i,f,p)` → vetor com início i , final f e passo p
- `linspace(i,f,n)` → vetor com n números de i até f

Criando vetores (arrays) em Numpy

```
9 a = np . array ( [36.4 , 21.6 , 15.6 , 27.5] ) #cria um vetor
10                               #com os valores passados por parâmetro
11 print a
12 az = np.zeros (4) #cria um vetor de zeros, de tam = 4
13 print az
14 a = np.arange(10) #cria um vetor com valores de 0 a 10
15 print a
16 a = np.arange ( 0.0 , 1.0 , 0.2) #vetor de 0 a 1, passo =0.2
17 print a
18 a = np.linspace ( 0.0 , 1.0 , 6) #vetor de 6 números de 0 a 1
19 print a
20 print a.size , a.ndim , a.shape |
```

Console - /home/analivia/Área de Trabalho/untitled0.py

```
[ 36.4  21.6  15.6  27.5]
[ 0.  0.  0.  0.]
[0 1 2 3 4 5 6 7 8 9]
[ 0.  0.2  0.4  0.6  0.8]
[ 0.  0.2  0.4  0.6  0.8  1. ]
6 1 (6,)
```

Criando vetores (arrays) em Numpy

Outros métodos de arrays:

- `a.size` → tamanho do array
- `a.ndim` → dimensão do array
- `a.reshape` → alterar a ordem do array

Criando vetores (arrays) em Numpy

```
38 m = a.reshape (2 ,3) #redimensiona o vetor a
39 print m
40 print m.size, m.ndim, m.shape #imprime o tamanho de m, a dimensão e
41 Z = np.zeros((3,3)) #vetor de 0, de ordem 3 (matriz 3x3)
42 print Z
```

Console - /home/analivia/Área de Trabalho/untitled0.py

 Python 1   untitled0.py 

```
[[ 0.  0.2  0.4]
 [ 0.6  0.8  1. ]]
6 2 (2, 3)
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Acesso a arrays

- `a[x,y]` → acessa o elemento na posição `x,y` de `a`
- `a[x,:]` ou `a[x]` → acessa a linha `x` de `a`
- `a[:,y]` → acessa a coluna `y` de `a`
- `a[-1]` → acessa a última linha de `a` • `a[x:x',:]` → acessa as linhas de `x` até a anterior de `x'`
- `a[::x,::y]` → acessa os elementos nas linhas 0 e `x`, e nas colunas 0 e `y`

Operações com Arrays

NumPy suporta operações entre arrays sem uso de loops (como em c, por exemplo)

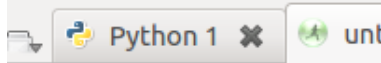
- $a + n \rightarrow$ soma n unidades aos elementos de a
- $a * n \rightarrow$ multiplica os elementos de a por um escalar n
- $a * b \rightarrow$ multiplica um array a por outro array b
- $a ** n \rightarrow$ eleva os elementos de a , a n -ésima potência
- $\text{dot}(a, b) \rightarrow$ produto escalar entre a e b
- $\text{cross}(a, b) \rightarrow$ produto vetorial entre a e b
- $\text{outer}(a, b) \rightarrow$ multiplica cada elemento de a por cada elemento de b , e cria uma matriz onde cada produto é uma linha.

Operações com Arrays

Exemplo de outer:

```
59 a = [4,5,6]  
60 b = [7,8,9]  
61 print np.outer(a,b)
```

Console - /home/analivia/Área



```
[[28 32 36]  
 [35 40 45]  
 [42 48 54]]
```

Métodos dos vetores

- `a.sum()` → retorna a soma dos elementos de `a`
- `a.min()` → retorna o menor elemento de `a`
- `a.max()` → retorna o maior elemento de `a`
- `a.mean()` → retorna a média dos elementos de `a`
- `a.std()` → retorna o desvio padrão dos elementos de `a`
- `a.var()` → retorna a variância dos elementos de `a`
- `a.trace()` → traço de `a`
- `a.copy()` → retorna a cópia de `a`
- `a.conjugate()` → retorna o complexo conjugado de `a`

Os arrays apresentados até agora são do tipo **ndarray**. Há um tipo de array bidimensional chamado **matrix**, que possui algumas propriedades especiais de matrizes:

- `matrix.I` → inversa
 - `matrix.T` → transposta
 - `matrix.H` → conjugada
 - `matrix.A` → transforma a matriz em um array
 - O operador `*` efetua operações usuais de álgebra linear, tipo matriz-matriz, matriz-vetor, vetor-matriz.
 - O módulo **numpy.linalg** possui diversas funções de álgebra linear, como a solução de sistemas de equações lineares por exemplo.
- `x = linalg.solve((A,B))`

O numpy define também um tipo para polinômios, com operações aritméticas, derivação, integração e avaliação de polinômios. É possível também fazer o ajuste de curvas através do Método dos Mínimos Quadrados.

SciPy

É uma coleção de algoritmos matemáticos e funções utilitárias, implementadas sobre o NumPy, divididas em sub-módulos:

- constants: Constantes físicas
- fftpack: Transformada Rápida de Fourier
- integrate: Integração numérica e ODE solvers
- interpolate: Interpolação (Splines)
- stats: Distribuições e funções estatísticas
- optimize: Otimização
- sparse: Matrizes esparsas
- linalg: Álgebra Linear
- io: Entrada e Saída
- signal: Processamento digital de sinais
- ndimage: Processamento digital de imagens

matplotlib

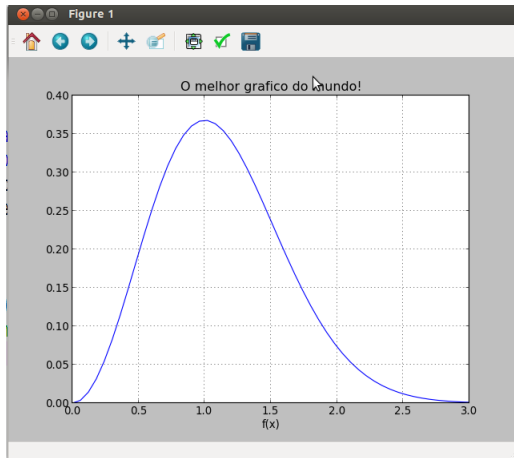
O matplotlib é uma biblioteca que permite a visualização de dados 2D seguindo o estilo do MATLAB. Gera gráficos de qualidade para publicações, exporta para diversos formatos e tem a possibilidade de embutir em interfaces gráficas (Qt, GTK, ...).

- Baseado no NumPy e SciPy
- pylab: módulo com diversas funções para plotar gráficos de
- forma fácil

Exepmlo usando o **plot(x,y)**

```
1 import numpy as np
2 from pylab import *
3
4 x = np.linspace(0,3,51)
5 y = x**2 *np.exp(-x**2)
6
7 plot(x,y)
8 grid(True)
9 xlabel('x')
10 xlabel('f(x)')
11 title ("O melhor grafico do mundo!")
12 show()
```

Gráfico Plotado:



Adicionando mais informações ao gráfico:

```
8 import numpy as np
9 from pylab import *
10 x = np.linspace(-6,6,500)
11 plot(x, sin(x), label='sen(x)') #plota o gráfico de sen(x)
12 plot(x, cos(x), label='cos(x)') #plota o gráfico de cos(x)
13 xlabel('x') #define o nome do eixo abcissas como "x"
14 xlabel('f(x)') #define o nome do eixo das ordenadas como "f(x)"
15 title ("Seno x Cosseno") #Atribui um título para o gráfico
16 axis([-6,6,-2,2]) #determina o intervalo de plotagem
17 legend(loc="upper right") #Posiciona a legenda
18 show() #mostra o gráfico!!
```

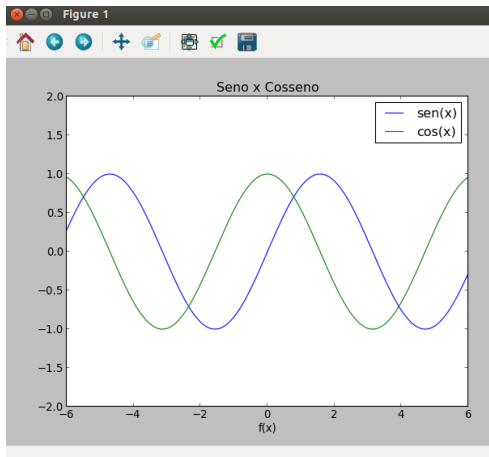


Gráfico Plotado: