PYTHON
INTRODUÇÃO À LINGUAGEM
PARTE I

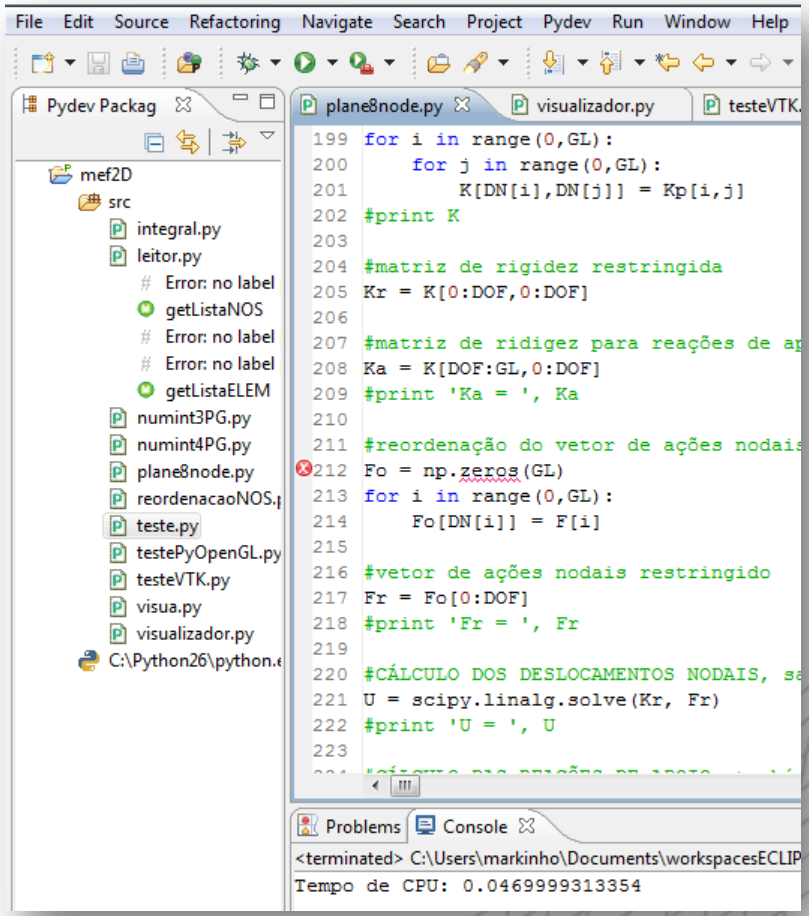
Prof. Marco André Argenta

Grupo de bioengenharia – CESEC/UFPR

Resumo

Neste curso de introdução à linguagem de programação Python, pretende-se trabalhar com os componentes básicos e fundamentais da linguagem, fazendo com que o aluno tenha plena capacitação de iniciar o desenvolvimento em python.

Além do básico e do fundamental da linguagem, serão discutidos alguns aspectos como o ambiente de programação, os pacotes mais úteis para a programação científica, visualização, geração de ambiente, etc. Por fim, uma introdução ao PyCUDA, wrapper para programação em placas de video da nvidia usando o python e o CUDA.



Índice

Tópicos relacionados

Módulo I

1. Um pouco de história
2. Introdução
3. Configurando o Eclipse
4. Introdução à Sintaxe
5. Construções Básicas
6. Tipos Básicos
7. Biblioteca Padrão
8. Referências

Um pouco de história

Nascimento do python, bases do python e o ambiente de concepção, quem usa?

Um pouco de história

Nascimento do Python

- Criada em 1989 pelo holandês Guido van Rossum no Centrum voor Wiskunde en Informatica (CWI), em Amsterdã, Holanda.
- Influenciada pela linguagem ABC, desenvolvida no CWI por Guido e outros nas décadas de 70 e 80. ABC tinha um foco bem definido: ser uma linguagem de programação para usuários inteligentes de computadores que não eram programadores: Físicos, engenheiros, cientistas sociais e até linguistas.
- O projeto de sistema operacional distribuído Amoeba precisava de uma linguagem de script. Nasce o Python.

Um pouco de história

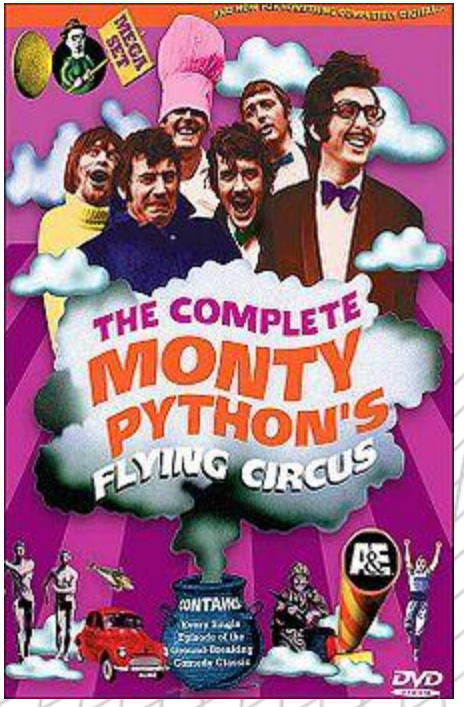
Bases do Python

- Elementos que eram bem sucedidos no ABC.
- Estruturas de dados poderosas inclusas: Listas, Dicionários, Strings.
- Usar indentação para delimitar blocos, eliminando chaves.
- Fácil extensão (lição aprendida com os erros do ABC).
- Fácil de portar: além do Amoeba, gostaria de executar em Unix, Macintosh e Windows.
- Influências de Modula-2 e Modula-3: módulos e namespaces.

Um pouco de história

Ambiente de concepção

- Universidade: pessoas altamente especializadas para desenvolver e opinar sobre os elementos do projeto;
- Descontraído: o nome Python vem da série de humor Monty Python's Flying Circus;
  - Monty Python's Flying Circus (em Portugal também conhecido, mas raramente, por Os Malucos do Circo) foi um série para televisão britânica transmitido pela BBC entre 1969 a 1974. Consistiu de 45 episódios divididos em 4 temporadas.
  - A série, que foi ao ar pela primeira vez em 5 de outubro de 1969, foi criada, roterizada e estrelada pelos Monty Python Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones e Michael Palin.
- Sem prazos, Sem pressão: o desenvolvimento não foi pressionado por estratégias de marketing, prazos, clientes ou qualquer outro fator que pudesse influenciar nas decisões de projeto, resultando em maior qualidade;
- Software Livre: garante a vida da tecnologia.



Um pouco de história

Quem usa?

- NASA: repositório de CAD/CAE/PDM, gerência de modelos, integração e sistema colaborativo: “We chose Python because it provides maximum productivity, code that’s clear and easy to maintain, strong and extensive (and growing!) libraries, and excellent capabilities for integration with other applications on any platform”.
- University of Maryland: ensino: “I have the students learn Python in our undergraduate and graduate Semantic Web courses. Why? Because basically there’s nothing else with the flexibility and as many web libraries”.
- Maior “case” Python da atualidade (Help Gmail, GoogleGroups, etc…): ”Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we’re looking for more people with skills in this language.” — Peter Norvig, director of search quality at Google, Inc.

Introdução

Características básicas, características importantes.

Introdução

Características básicas

- Interpretada: usa máquina virtual, facilita portabilidade.
- Interativa: pode-se programar interativamente, os comandos são executados enquanto são digitados. Facilita testes, desenvolvimento rápido e outros. Facilitadores estão presentes help(obj).
- Orientada a Objetos: tudo1 é objeto: números, strings, funções, classes, instâncias, métodos, ...
- Fortemente Tipada: Não existe casts e nem conversão automática, não se mistura tipos “automagicamente”.
- Tipagem Dinâmica: A tipagem de um objeto é feita em
- tempo de execução. Um objeto tem tipo, uma variável não.

Introdução

Características importantes

- Sintaxe clara, sem caracteres “inúteis”:
  - blocos são marcados por indentação
  - parênteses são opcionais, só precisam ser utilizados para
  - eliminar ambiguidades.
  - palavras-chave (keywords) e formações que ajudam na leitura,
  - como for ... in ....
- Fácil extensão: codificar nos módulos é muito fácil, podendo
  - utilizar bibliotecas nativas, aproveitando desempenho,
  - características nativas das plataformas, etc.
  - API Python/C é bem simples
  - Diversos conversores automáticos (SWIG, SIP, ...)
  - Jython: usando Python em Java e vice-versa.
  - PyRex: pseudo linguagem para facilitar integração Python + C/C++.

Introdução

Características importantes

- Tipos básicos poderosos: listas, dicionários (hash tables), strings, ... otimizados e de fácil uso.
- Baterias Inclusas: biblioteca padrão contém diversos recursos úteis: Interface Gráfica (Tk), XML, Servidores (TCP, UDP, HTTP, ...), HTML, protocolos de internet (email, http, ...), xmlrpc, ...
- Grande base de código e bibliotecas de terceiros
- Grande comunidade de desenvolvedores
- Software Livre: liberdade de uso, distribuição. Licença própria, compatível com GPL, porém pode distribuir somente o binário.
- Independente: a entidade sem fins lucrativos Python Software Foundation cuida da propriedade intelectual do Python.

Configuração do Eclipse

Python(x,y), interpretador, projeto.

Configuração Eclipse

Python(x,y)

Python (x,y) é um software livre para computação numérica, científica, de engenharia, para análise de dados e visualização de dados baseado na linguagem de programação Python, usando as interfaces gráficas do usuário Qt e o ambiente de desenvolvimento integrado Eclipse.

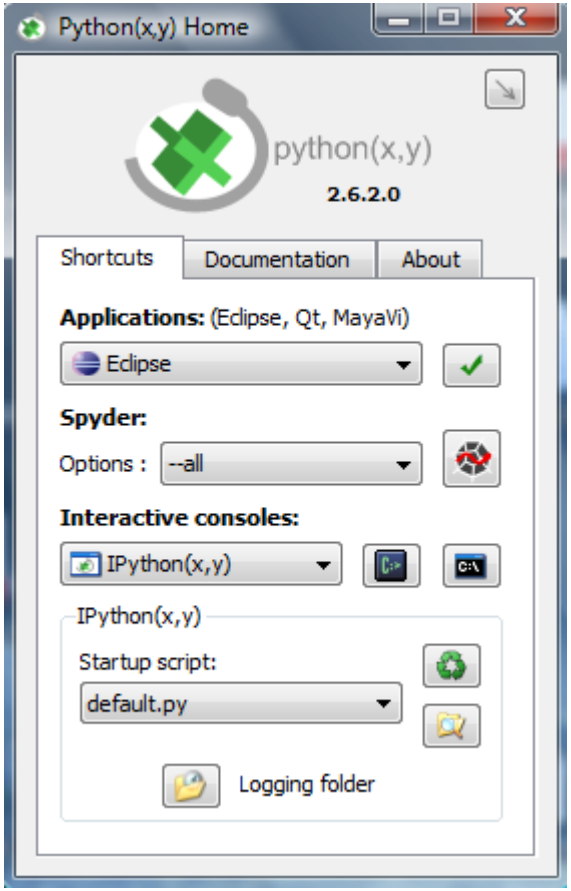
O que é exatamente o Python (x,y)?

Python (x, y) é uma distribuição do científica-orientada do Python baseada no Qt e Eclipse. Sua finalidade é ajudar os programadores científicos que utilizam linguagens interpretadas (como MATLAB ou IDL) ou linguagens compiladas (C/C++ ou Fortran), a mudar para Python. Programadores de C/C++ ou Fortran vão apreciar poder reutilizar seus códigos "tal como estão" usando um wrapper e podendo assim chamá-los diretamente a partir de scripts em Python.

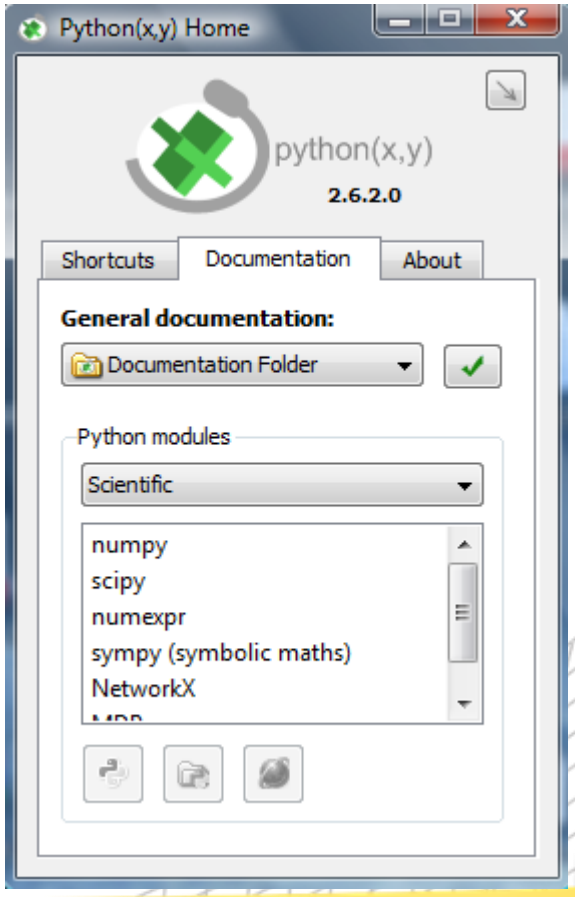
Configuração Eclipse

Python(x,y)

Atalhos:



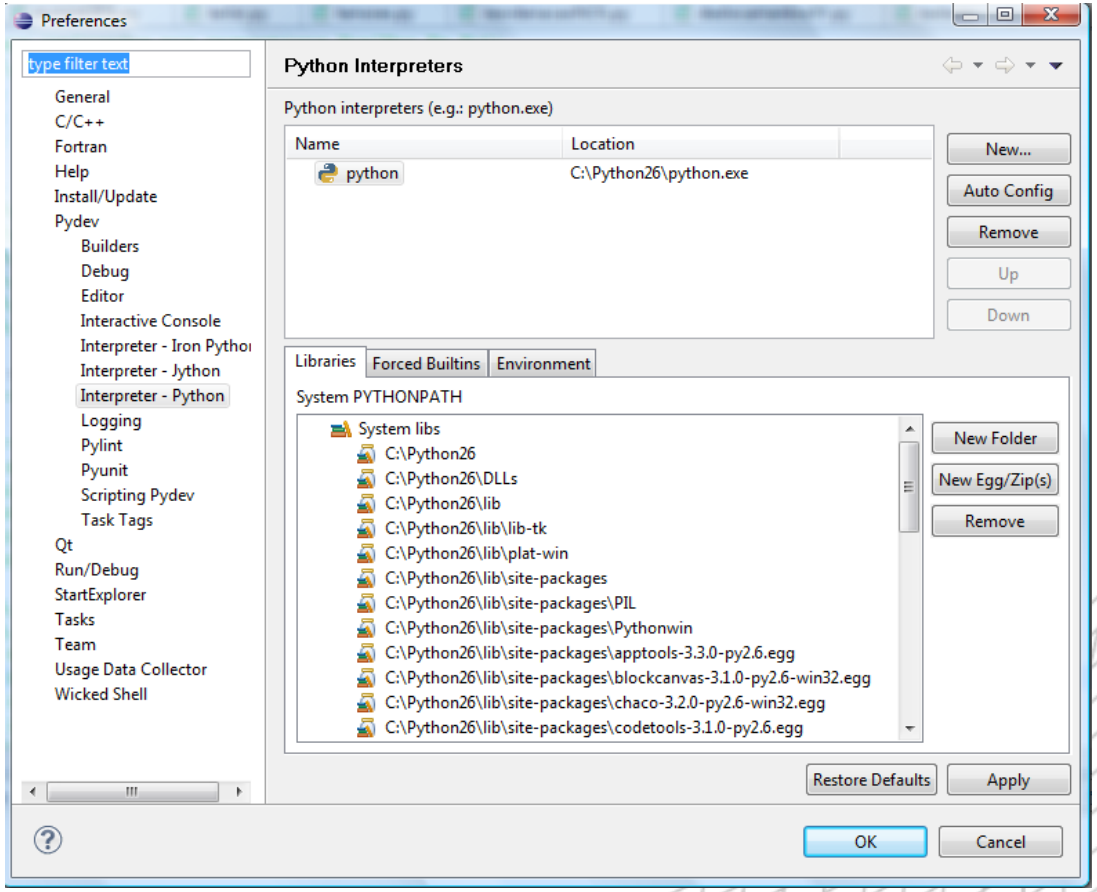
Documentação:



Configuração Eclipse

Configurando o interpretador

- Após iniciar um novo projeto, deve-se configurar o interpretador Python.
- Isso é feito indo em windows > preferences > PyDev > interpreter Python
- Basicamente, clica-se em “Auto Config” e após “OK”.



Introdução à Sintaxe

Sintaxe básica, comantários, identação, literais e operadores.

Introdução à sintaxe

Sintaxe básica

Um programa em Python é constituído de linhas lógicas:

- Linhas Lógicas são terminadas com uma nova linha;
- Exceto quando explicitamente continuadas na próxima linha física, para isto usa-se o "\".
- Ou implicitamente continuadas na próxima linha física por expressões que usam parênteses, colchetes ou chaves.

```
a = b + c
x = x * f + \
    x0 * f
l = [ 1, 2,
      3, 4 ]
d = { 'a': 1, 'b': 2}
if a > x and \
    b < y:
    print "text"
```

Introdução à sintaxe

Comentários e comentários funcionais

- Após o caractere "#" até o final da linha, tudo é considerado um comentário e ignorado, exceto pelos comentários funcionais.
- Comentários funcionais geralmente são usados para:
  - alterar a codificação do arquivo fonte do programa acrescentando um comentário com o texto "#-*- coding: <encoding> -*#-" no inicio do arquivo, aonde <encoding> é a codificação do arquivo (cp1251 para português usando o windows PT-BR, utf-8 no linux). Alterar a codificação é necessário para suportar caracteres que não fazem parte da linguagem inglesa, no código fonte do programa.
  - definir o interpretador que será utilizado para rodar o programa em sistemas UNIX, através de um comentário começando com "#!" no inicio do arquivo, que indica o caminho para o interpretador (geralmente a linha de comentário será algo como "#!/usr/bin/env python").

```
#!/usr/bin/env python
# -*- coding: cp1252 -*-

#uma linha de código que mostra o resultado de 7 vezes 3
print 7 * 3
```

Introdução à sintaxe

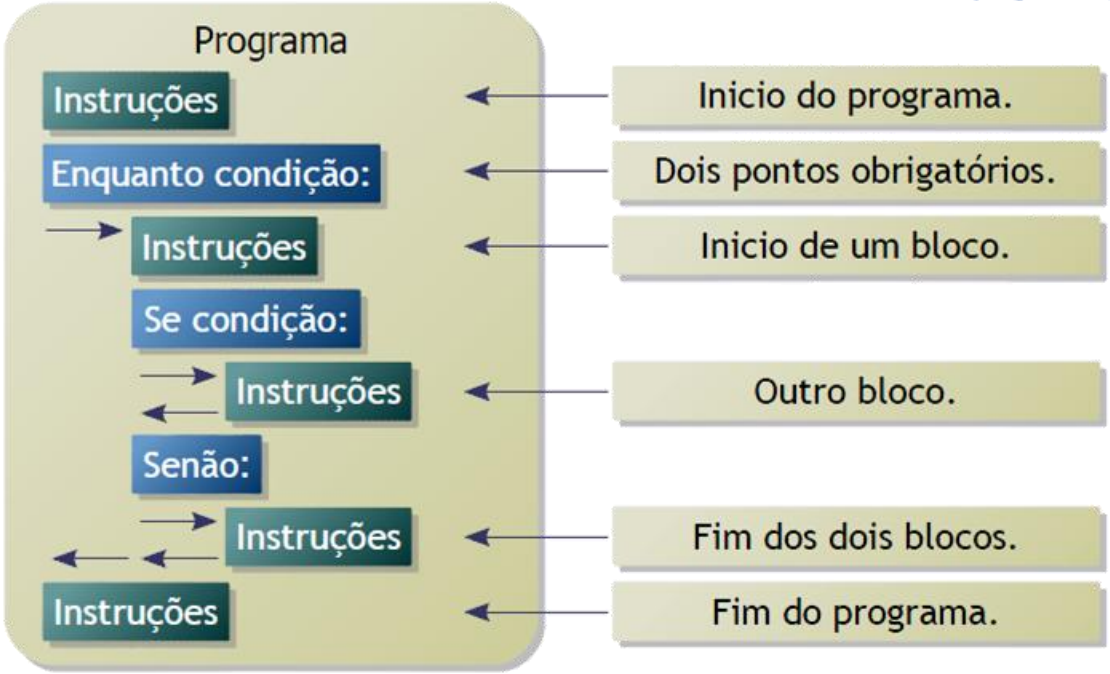
Identação

- Em Python, os blocos de código são delimitados pelo uso de indentação.
- A indentação não precisa ser consistente em todo o arquivo, só no bloco de código, porém é uma boa prática ser consistente no projeto todo.
- Cuidado ao misturar TAB e Espaços: configure seu editor!
- Utilitário tabnanny (geralmente está na instalação do seu python, como /usr/lib/python2.3/tabnanny.py) verifica indentação.

```
a = 1
b = 2
if a < b:
    print "a é menor"
else:
        print "b é menor" #apesar de ok, evite inconsistências!
```

Introdução à sintaxe

Identação




```
#para i na lista 104, 234, 654, 378, 798:
for i in [ 104, 234, 654, 378, 798 ]:

    #se o resto dividido por 3 for igual a zero:
    if i % 3 == 0:

        #imprime...
        print i, '/ 3 =', i / 3
```

saída

```
234 / 3 = 78
654 / 3 = 218
378 / 3 = 126
798 / 3 = 266
```

Introdução à sintaxe

Identificadores

- Devem começar com uma letra a-z, sem acentuação ou underline "_".
- Depois pode ser seguido por uma letra a-z, sem acentuação, dígitos e underline "_".
- Alguns identificadores são palavras-chave reservadas:

```
and, del, for, is, raise, assert, elif, from,
lambda, return, break, else, global, not, try,
class, except, if, or, while, continue, exec,
import, pass, yield, def, finally, in, print
```

- Python é case sensitive, ou seja, Maiúsculas e Minúsculas são diferentes!

Introdução à sintaxe

Literais

- Strings
  - Convencional: `'texto'` ou `"texto"`
  - Multi-Line: `'''texto várias linhas'''` ou `"""texto várias linhas"""`
  - Unicode: `u'texto unicode'` ou `u"texto"`, ...
  - Raw: `r'texto bruto\n'`
  - Strings em várias linhas são concatenadas.
- Números parecido com outras linguagens, C, C++, Java:
  - Inteiro: `123` (decimal), `0632` (octal), `0xff00` (hexadecimal)
  - Longo: `123L` ou `123l`
  - Ponto Flutuante: `3.14`, `10.`, `.12`, `1.23e-9`
  - Complexos: `10.0 + 3j`

Introdução à sintaxe

Operadores

Operador	Descrição	Método correspondente
+	adição	__add__
-	subtração	__sub__
*	multiplicação	__mul__
**	potenciação	__pow__
/	divisão	__div__
//	divisão por baixo (floor)	__floordiv__
%	módulo (resto)	__mod__
<<	deslocamento à esquerda	__lshift__
>>	deslocamento à direita	__rshift__
&	“e” lógico (and) bit-a-bit	__and__
|	“ou” lógico (or) bit-a-bit	__or__
^	“ou exclusivo” (xor) bit-a-bit	__xor__
~	Inverte	__inv__
<	menor	__lt__
>	maior	__gt__
<=	menor ou igual	__le__
>=	maior ou igual	__ge__
==	igual lógico	__eq__
!=	diferente lógico	__ne__

Para maiores informações `import operator; help(operator)`.

Construções Básicas

Variáveis, controles de fluxos, laços e funções.

Construções básicas

Variáveis

- Python usa tipagem dinâmica: uma variável não tem tipo fixo, ela tem o tipo do objeto que ela contém.
- Para criar um novo conteúdo para a variável é necessário apenas uma atribuição.
- Um conteúdo é destruído e recolhido pelo coletor de lixo quando nenhuma variável ou estrutura aponta mais para ele.

```
a = " texto " #a contém uma string, então é do tipo string (str)
a = 123 #a contém um inteiro, então é do tipo inteiro (int)
a = [ 1 , 2 , 3 ] #a é uma lista
b = [ a , " 123 " , 333 ] #b é uma lista
d = { " chave ": " valor " , " teste ": a , "b" : 12345 } #d é um dicionário
```

Construções básicas

Variáveis - exemplo

- Utilizar o Python no modo interativo como calculadora e calcular:

$$2^{50}$$

$$2^{50}.3$$

$$2^{50}.3 - 1000$$

$$\frac{400.0}{2^{50}} + 50$$

- Respostas:

```
a = 2 ** 50, a == 1125899906842624L
b = a * 30, b == 33776997205278720L
c = b - 1000, c == 33776997205277720L
d = 400.0 / a + 50, d == 50.000000000000355
```

Construções básicas

Controle de fluxo

Executando instruções de forma condicional do tipo

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

Sendo:

<condição>: sentença que possa ser avaliada como verdadeira ou falsa.

<bloco de código>: seqüência de linhas de comando.

As clausulas elif e else são opcionais e podem existir vários elifs para o mesmo if.

Parênteses só são necessários para evitar ambigüidades.

Construções básicas

Controle de fluxo

Em Python

```
idade = int ( raw_input ( " Idade :" ) )
if idade < 2:
    print " Bebe "
elif 2 <= idade <= 13:
    print " Criança "
elif 14 <= idade <= 19:
    print " Adolescente "
else :
    print " Adulto "
```

Sendo raw_input( ) o comando para a entrada pelo teclado em tempo de execução.

Usando diversas condições:

```
if a and ( b or c ):
    print " verdade "
```

Apartir da versão 2.5, python suporta também o seguinte tipo de expressão:

<variável> = <valor 1> if <condição> else <valor 2>

Construções básicas

O que avalia para verdadeiro e falso

- Os operadores lógicos são: and, or, not, is e in.
  - and: retorna verdadeiro se e somente se receber duas expressões que forem verdadeiras.
  - or: retorna falso se e somente se receber duas expressões que forem falsas.
  - not: retorna falso se receber uma expressão verdadeira e vice-versa.
  - is: retorna verdadeiro se receber duas referências ao mesmo objeto e falso em caso contrário.
  - in: retorna verdadeiro se receber um item e uma lista e o item ocorrer uma ou mais vezes na lista e falso em caso contrário.
- Os seguintes valores são considerados falsos:
  - Os seguintes valores são considerados falsos:
  - ▪ False (falso).
  - ▪ None (nulo).
  - ▪ 0 (zero).
  - ▪ '' (string vazia).
  - ▪ [] (lista vazia).
  - ▪ () (tupla vazia).
  - ▪ {} (dicionário vazio).
  - ▪ Outras estruturas com o tamanho igual a zero.

Construções básicas

Exemplo

- Implementar o seguinte conjunto de regras em Python:
  - Se a for verdadeiro e b for falso, imprima “Caso 1”
  - Senão, Caso a for falso e b for verdadeiro, imprima “Caso 2”
  - Caso contrário:
    - Caso c for maior que 10 e d estiver entre 0.0 e 100.0, imprima “Caso 3”
    - Caso e estiver na lista lst, imprima “Caso 4”
    - Senão imprima “Caso 5”

```
if a and not b:
    print " Caso 1"
elif not a and b:
    print " Caso 2"
else :
    if c > 10 and 0.0 <= d <= 100.0:
        print " Caso 3"
    if e in lst :
        print " Caso 4"
    else :
        print " Caso 5"
```

Construções básicas

Laços

- São dois tipos de laços em python:

  while CONDICAO :

    BLOCO_DE_CODIGO

  for VARIAVEL in SEQUENCIA :

    BLOCO_DE_CODIGO

Em python:

```
from time import time
start = time ()
while time () - start < 3.0:
    print " esperando ... "

for fruta in [ " Banana " , " Ma¸ca " , " Uva " ]:
    print " Fruta :" , fruta

d = { "a": 1 , "b" : 2 }
for chave , valor in d. iteritems ():
    print " Chave :" , chave , " , Valor :" , valor
for i in xrange ( 100 , 200 , 10 ):
    print "i:" , i
```

Construções básicas

Laços: próxima iteração e saída forçada

- **continue** interrompe a execução da iteração atual e vai para a próxima, se esta existir.
- **break** interrompe a execução do laço.

```
contador = 0
for contador in range(1,10):
    letra = raw_input ( " Entre com a letra :" )
    if letra == 'c':
        continue
    elif letra == 'b':
        break
print " contador :" , contador
```

Saída:

```
Entre com a letra :c
Entre com a letra :c
Entre com a letra :c
Entre com a letra :b
contador : 4
```

Construções básicas

Laços: usando o else

- Visando facilitar a vida do programador, Python fornece a cláusula else para os laços. Esta será executada quando a condição do laço for falsa, eliminando a necessidade do programador manter uma variável de estado.

```
lista = ['saída', 'stop', 'ida', 'voar']
for elemento in lista :
    if elemento == 'parada':
        break
    print elemento
else :
    print " Laço chegou ao fim "
```

Saída:

```
saída
stop
ida
voar
 Laço chegou ao fim
```

```
lista = ['saída', 'stop', 'ida', 'parada', 'voar']
for elemento in lista :
    if elemento == 'parada':
        break
    print elemento
else :
    print " Laço chegou ao fim "
```

Saída:

```
saída
stop
ida
```

No exemplo acima, a mensagem “Laço chegou ao fim” só é imprimida caso não existir um elemento que seja igual a “parada”.

Construções básicas

Exemplo

- Dada uma lista de palavras “lista” e uma palavra “chave” imprimir o índice do elemento que encontrou a palavra, senão imprimir “Palavra não encontrada”.

```
lista = ['um', 'dois', 'três']
chave = 'três'

for indice , palavra in enumerate(lista):
    if palavra == chave :
        print indice
        break
else :
    print " Palavra não encontrada "
```

Saída:

```
2
```

Construções básicas

Exemplo: Explicação

- enumerate(sequencia) é um iterador que retorna pares (indice, sequencia[indice])
- Em python, a construção a seguir é válida:

```
x , y = 1 , 2
print x # 1
print y # 2
par = 1 , 2 # idem a par = (1 , 2)
print par # ( 1 , 2 )
x , y = par # idem a x , y = 1 , 2
```

Saída:

```
1
2
(1, 2)
```

- Então for `índice, palavra in enumerate(lista)` também é válido!

Construções básicas

Funções

- 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).

- Doc Strings são strings que estão associadas a uma estrutura do Python. Nas funções, as Doc Strings são colocadas dentro do corpo da função, geralmente no começo. O objetivo das Doc Strings é servir de documentação para aquela estrutura.

- Sintaxe:

  def NOME_DA_FUNCAO (LISTA_DE_PARAMETROS):

  BLOCO_DE_CODIGO

Em python:

```
def fatorial ( numero ):
    if numero <= 1:
        return 1
    else:
        return ( numero * fatorial ( numero - 1 ) )

print fatorial(4)
```

Saída:

```
24
```

Construções básicas

Funções: parâmetros com valor padrão

- Pode-se ter parâmetros com valores padrão, estes devem vir depois dos parâmetros sem valor padrão.

```
def f( a , b , c =3 ):
    print "a:" , a , "b:" , b , "c:" , c

f ( 1 , 2 )
f ( 1 , 2 , 0 )
```

Saída:

```
a: 1 b: 2 c: 3
a: 1 b: 2 c: 0
```

Cuidado!

- O valor do padrão para um parâmetro é calculado somente uma vez quando o programa é carregado, caso você use um objeto mutável, todas as chamadas usarão a mesma instância.

Construções básicas

Funções: parâmetros com valor padrão

- Exemplo do objeto mutável:

```
def f(v , l =[]): # l é instanciado quando o python lê esta linha
    l.append(v)
    return l

print f(1)
print f(2)
```

Saída:

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

- Talvez seja este o comportamento que você quer, mas talvez não. Caso deseje que uma nova instância seja criada para cada chamada, utilize algo como:

```
def f( v , l= None ):
    if l is None :
        l = []
        l. append ( v )
    return l
```

ou

```
def g( v , l= None ):
    l = l or []
    l. append ( v )
    return l
```

```
print f(1)
print f(2)
print g(1)
print g(2)
```

Saída:

```
[1]
[2]
[1]
[2]
```

Construções básicas

Funções: número variável de argumentos

- **Argumentos sem nome:** os argumentos são passados para a função na forma de uma lista, na ordem em que foram digitados:

```
def arg_sem_nome( * args ):
    for arg in args :
        print " arg :" , arg

arg_sem_nome( 'a' , 'b' , 123 )
```

Saída:

```
arg : a
arg : b
arg : 123
```

- **Argumentos com nome:** os argumentos são passaos para a função na forma de um dicionário, o nome do argumento é a chave.

```
def arg_com_nome ( ** kargs ):
    for nome , valor in kargs . iteritems ():
        print nome , "=" , valor

arg_com_nome( a=1 , b=2 , teste =123 )
```

Saída:

```
a = 1
b = 2
teste = 123
```

- **Usando Ambos:**

```
def f( a , b , * args , ** kargs ):
    print "a:" , a , " , b:" , b , \
        " args :" , args , " , kargs :" ,\
        kargs

f (1 , 2); f (1 , 2 , 3); f(1 , 2 , t =9);
```

Saída:

```
a: 1  , b: 2  args : ()  , kargs : {}
a: 1  , b: 2  args : (3,)  , kargs : {}
a: 1  , b: 2  args : ()  , kargs : {'t': 9}
```

Construções básicas

Exemplo

- Faça uma função que dado um número, retorne o próximo na sequência de Robert Morris (http://www.ocf.berkeley.edu/~stoll/number_sequence.html)

  1, 11, 21, 1211, 111221, ...

```
def next_morris ( number ):
    number = str ( number )
    r = []
    i = 0
    last = number [ 0 ]
    for c in number :
        if c == last :
            i += 1
        else :
            r.append( str (i ) + last )
            last = c
            i = 1
    r.append( str (i ) + last )
    return "". join ( r )

print next_morris(111221)
```

Saída:

```
312211
```

Construções básicas

Peculiaridades de um bloco de código

- Um bloco vazio é criado com o keyword `pass`

```
while True :
    pass # Bloco vazio
```

- Qualquer string "solta" (não atribuéda a variáveis) é considerada uma "docstring" e contribui para a documentação do bloco, no atributo __doc__:

```
def f ():
    '''
    Documentação da função.
    Esta função faz bla bla bla ...
    '''
    corpo ()
    return valor
```

Tipos Básicos

Números, Sequências (listas, tuplas, dicionários), Mapeamento e Strings.

Tipos Básicos

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
- Além dos números inteiros convencionais, existem também os inteiros longos, que tem dimensão arbitrária e são limitados pela memória disponível. As conversões entre inteiro e longo são realizadas de forma automática. A função *builtin int()* pode ser usada para converter outros tipos para inteiro, incluindo mudanças de base.

Tipos Básicos

Números

- Exemplo:

```
print 'int(3.14) =', int(3.14) #Convertendo de real para inteiro
print 'float(5) =', float(5) #Convertendo de inteiro para real
print '5.0 / 2 + 3 = ', 5.0 / 2 + 3 #Cálculo entre inteiro e real resulta em real

# Inteiros em outra base
print 127 #base 10 (decimal)
print 0177 #base 8 (octal)
print 0x7f #base 16 (hexadecimal)

#Operações com números complexos
c = 3 + 4j
print 'c =', c
print 'Parte real:', c.real
print 'Parte imaginária:', c.imag
print 'Conjugado:', c.conjugate()
```

Saída:

```
int(3.14) = 3
float(5) = 5.0
5.0 / 2 + 3 =  5.5
0x7f
127
127
127
c = (3+4j)
Parte real: 3.0
Parte imaginária: 4.0
Conjugado: (3-4j)
```

- Os números reais também podem ser representados em notação cientifica, por exemplo: 1.2e22.

Tipos Básicos

Números

- O Python tem uma série de operadores definidos para manipular números, através de cálculos aritméticos, operações lógicas (que testam se uma determina condição é verdadeira ou falsa) ou processamento bit-a-bit (em que os números são tratados na forma binária).

- 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 (<=).
  - Maior ou igual (>=).
  - Igual (==).
  - Diferente (!=).

- Operações bit-a-bit (bitwise):
  - Deslocamento para esquerda (<<).
  - Deslocamento para direita (>>).
  - E bit-a-bit (&).
  - Ou bit-a-bit (|).
  - Ou exclusivo bit-a-bit (^).
  - Inversão (~).

Tipos Básicos

Números

- Durante as operações, os números serão convertidos de forma adequada (exemplo: (1.5+4j) + 3 resulta em 4.5+4j).
- Além dos operadores, também existem algumas funções *builtin* para lidar com tipos numéricos: *abs()*, que retorna o valor absoluto do número, *oct()*, que converte para octal, *hex()*, que converte para hexadecimal, *pow()*, que eleva um número por outro e *round()*, que retorna um número real com o arredondamento especificado.

```
a = 60.456 + 4.j
b = -34.2354
c = 2435

print abs(a), abs(b)
print oct(c) #c deve ser inteiro
print hex(c) #c deve ser inteiro
print pow(b, 3)
print round(b, 1) #segundo parâmetro número de casas decimais
```

Saída:

```
60.5881831383 34.2354
04603
0x983
-40126.0323866
-34.2
```

Tipos Básicos

Números – Operadores adicionais

- Operadores de atribuição:

  - = Operador de atribuição simples (c = a + b vai atribuir o valor de a + b em c).
  - += Adiciona e atribui (c += a é equivalente a c = c + a).
  - -= Subtrai e atribui (c-= a é equivalente a c = c – a)
  - *= Multiplica e atribui (c *= a é equivalente a c = c * a)
  - /= Divide atribui (c /= a é equivalente a c = c / a)
  - %= Usa o módulo e atribui (c %= a equivale a c = c % a)
  - **= Calcula o expoente atribui (c **= a é equivalente a c = c ** a)
  - //= Divisão por baixo e atribui (c //= a é equivalente a c = c // a)

Tipos Básicos - Sequências

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.
- Criação de uma lista:

```
lista = [ 10, 30., 'texto', 5, 'texto2']
```

- Acessando elementos pelo índice:

```
print lista[1]
```

Saída:

```
30.0
```

- Mudando elementos já existentes (a posição já deve existir):

```
lista[0] = 123
```

Mostrando a lista:

```
[123, 30.0, 'texto', 5, 'texto2']
```

- Acessando pedaços da lista:

```
sub_lista = lista [ 2: 4 ]
sl1 = lista [ : 3 ]
sl2 = lista [ 3 : ]
```

Mostrando as novas listas:

```
['texto', 5]
[123, 30.0, 'texto']
[5, 'texto2']
```

Tipos Básicos - Sequências

Listas

- Acrescentando mais um item ao final da lista:

```
lista.append(1)
```

Saída:
```
[123, 30.0, 'texto', 5, 'texto2', 1]
```

- Estendendo a lista com outra:

```
lista.extend([10,20,30])
lista += [40,50,60]
```

Saída:
```
[123, 30.0, 'texto', 5, 'texto2', 1, 10, 20, 30]
[123, 30.0, 'texto', 5, 'texto2', 1, 10, 20, 30, 40, 50, 60]
```

- Ordenando a lista (altera as posições na lista):

```
lista.sort()
lista.sort(lambda x,y: cmp(x,y))
```

Saída:
```
[1, 5, 10, 20, 30.0, 30, 40, 50, 60, 123, 'texto', 'texto2']
```

- Invertendo a lista (também altera as posições):

```
lista.reverse()
lista.sort(lambda x,y: cmp(y,x))
```

Saída:
```
['texto2', 'texto', 123, 60, 50, 40, 30.0, 30, 20, 10, 5, 1]
```

- Contar ocorrências de um elemento:

```
lista.count(30)
```

Saída:
```
2
```

Tipos Básicos - Sequências

Listas

- Mostrando a posição do elemento na lista CORRIGIR index():

```
lista.count(30)
```

Saída: `4`

- Inserir um elemento em certa posição na lista:

```
lista.insert ( 0 , 'abc' )
```

Saída: `['abc', 'texto2', 'texto', 123, 60, 50, 40, 30, 30.0, 20, 10, 5, 1]`

- Apagando um elemento da lista:

```
del lista[2]
```

Saída: `['abc', 'texto2', 123, 60, 50, 40, 30, 30.0, 20, 10, 5, 1]`

- Apagando um pedaço da lista:

```
del lista[3:7]
```

Saída: `['abc', 'texto2', 123, 30.0, 20, 10, 5, 1]`

- Apagando certo elemento na lista:

```
lista.remove('texto2')
```

Saída: `['abc', 123, 30.0, 20, 10, 5, 1]`

- Mudando um pedaço da lista:

```
lista[2:4] = [10, 20]
```

Saída: `['abc', 123, 10, 20, 10, 5, 1]`

Tipos Básicos - Sequências

Listas

- Repetindo uma lista:

```
l = [1, 2]*5
```

Saída:
```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

- Mostrando o tamanho da lista:

```
print len(lista)
```

Saída:
```
7
```

- Criando uma lista numerada com as posições:

```
lista = range(10)
```

Saída:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Criando uma lista numerada com as posições em formato string:

```
l1 = [ str(i) for i in range(10) ]
```

Saída:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- Criando uma lista com o índice ao quadrado, mas somente se for múltiplo de 2:

```
l2 = [ i**2 for i in range(5) if i % 2 == 0 ]
```

Saída:
```
[0, 4, 16]
```

Tipos Básicos - Sequências

Listas

- Criando uma lista de de 0 a 30 indo de 10 em 10:

```
l2 = range( 0, 30, 10 )
```

Saída:
```
[0, 10, 20]
```

- Juntando duas listas formando tuplas (as listas não precisam ter o mesmo tamanho, no entanto as tuplas se formarão até o tamanho da menor lista):

```
zip(l1, l2)
```

Saída:
```
[(0, 0), (1, 10), (2, 20)]
```

- Somando os elementos de uma lista:

```
sum(l1), sum(l2)
```

Saída:
```
3 30
```

- Esses são as principais operações com listas. Para mais informações:
  - `help(list)` no console do python ou
  - http://www.python.org/doc/2.3/lib/typesseq-mutable.html ou
  - http://docs.python.org/tutorial/datastructures.html

Tipos Básicos - Sequências

Listas - Exercícios

1. Crie uma lista “lista”, verifique se “valor” está dentro dela, caso verdade imprima “Sim”, senão imprima “Não”.

2. Crie outra lista. Uma lista “lista”, itere sobre a lista, imprimindo cada um de seus elementos.

3. Usando a lista de 10 componentes, cada componente com o valor do seu indice, crie uma nova lista “rotaciona_3” que cada posição está rotacionada, ou corrida, ou caminhada na lista em 3 posições para a esquerda, por exemplo, o índice “zero” ficaria:

   0 = 0 + 3

   Dica: existe uma forma simples de se fazer isso em python.

Tipos Básicos - Sequências

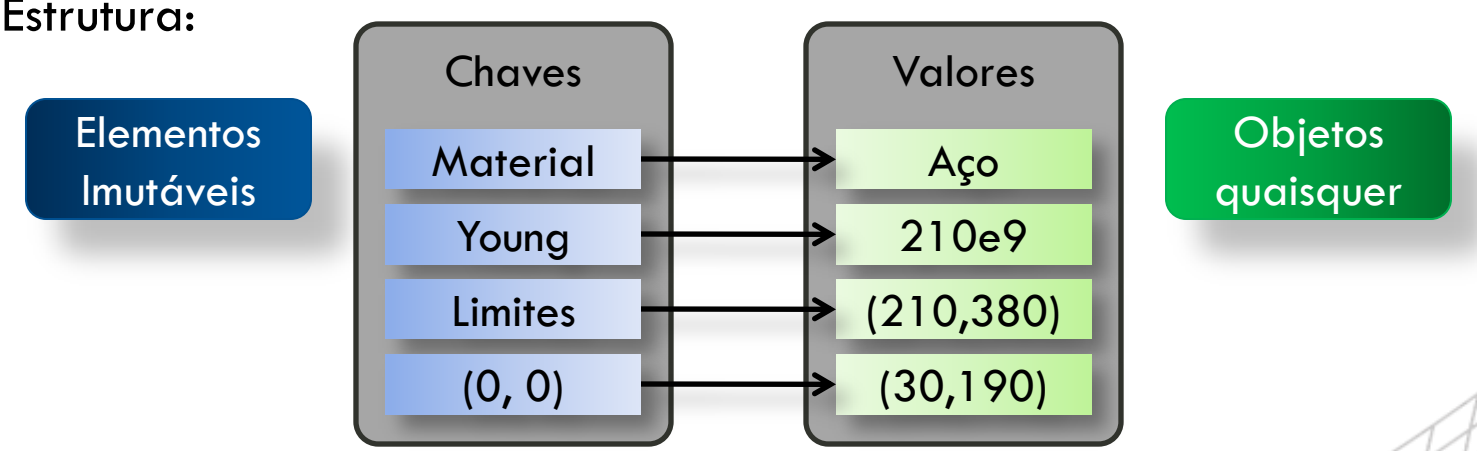
Tuplas

- Semelhantes as listas, porém são imutáveis: não se pode acrescentar, apagar ou fazer atribuições aos itens.
- Criação da tupla:
```
tupla = ( 1,  2, 'abc ')
```
- Mostrando elemento:
```
tupla[0]
```
Saída:
```
1
```
- Outros exemplos:
```
tupla[:2]
tupla[2:]
len(tupla)
t = 1, 2, 3
```
Saídas:
```
(1, 2)
('abc ',)
3
(1, 2, 3)
```
- Tuplas podem ser convertidas em listas e vice-versa:
```
lista = list(tupla)
tupla = tuple(lista)
```
Saídas:
```
[1, 2, 'abc ']
(1, 2, 'abc ')
```
- As tuplas são mais eficientes do que as listas convencionais, pois consomem menos recursos computacionais.

Tipos Básicos - Mapeamento

Dicionários (hash tables)

- Um dicionário é uma lista de associações compostas por uma chave única e estruturas correspondentes. Dicionários são mutáveis, tais como as listas.
- A chave tem que ser de um tipo imutável, geralmente strings, mas também podem ser tuplas ou tipos numéricos. O dicionário do Python não fornece garantia de que as chaves estarão ordenadas.
- Estrutura:



- Criação:

```
material = {'Material':'aço', 'Young':210e9, 'Limites':(210,380), (0,0):(30,190)}
```

Tipos Básicos - Mapeamento

Dicionários (hash tables)

- Acessando elementos:

```
material['Young']
```

Saída:

```
210000000000.0
```

- Adicionando elementos (a chave precisa sem imutável, p. ex. n pode ser uma lista):

```
material['chave'] = 'valor'
material[1] = 10
material[(1,5)] = 'coord'
lista = [1, 5, 10.0]
material['listagem'] = lista
```

Saída:

```
{1: 10, (0, 0): (30, 190), 'listagem': [1, 5, 10.0], 'Material': 'aco', 'Young': 210000000000.0, 'chave': 'valor', (1, 5): 'coord', 'Limites': (210, 380)}
```

- Apagar um elemento do dicionário:

```
del material['chave']
```

- Apagar completamente o dicionário:

```
material.clear()
```

Tipos Básicos - Mapeamento

Dicionários (hash tables)

- Obtendo itens chaves e valores:

```
dici = {'a':1, 'b':2, 3:10, (1,2):(4,1)}
itens = dici.items()
chaves = dici.keys()
valores = dici.values()
```

Saída:

```
[('a', 1), (3, 10), ('b', 2), ((1, 2), (4, 1))]
['a', 3, 'b', (1, 2)]
[1, 10, 2, (4, 1)]
```

- Obtendo iteradores (otimizado para `for`):

```
for chave in dici.iterkeys():
    print chave
for valor in dici.itervalues():
    print valor
for chave, valor in dici.iteritems():
    print chave , '=' , valor
```

Saída:

```
a
3
b
(1, 2)
1
10
2
(4, 1)
a = 1
3 = 10
b = 2
(1, 2) = (4, 1)
```

Tipos Básicos - Mapeamento

Dicionários - Exercícios

- Crie um dicionário “simulação” e coloque nele os dados de uma simulação de elementos finitos: material, elemento, dof, força, apoio

- Usando o dicionário criado anteriormente, imprima material e acrescente “modos” ao dicionário com o valor de 5 e exclua a chave “dof”.

- Também usando “simulação”, imprima todos os itens do dicionário no formato “chave : valor”, ordenado pela chave.

Tipos Básicos - Strings

Texto

- As strings no Python são *buitins* para armazenar texto. São imutáveis, sendo assim, não é possível adicionar, remover ou mesmo modificar algum caractere de uma string. Para realizar essas operações, o Python precisa criar um nova string.

- Tipos:
  - String padrão: s = 'Led Zeppelin'
  - String unicode: u = u'Björk'

  A string padrão pode ser convertida para unicode através da função unicode().

- 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', aonde s conterá os caracteres "\" e "n").

Tipos Básicos - Strings

Texto - Operações

- Criação:

```
texto1 = 'primeiro texto'
texto2 = "outro texto"
texto3 = '''este texto
tem varias
linhas'''
```

Saída:

```
primeiro texto
outro texto
este texto
tem varias
linhas
```

- Acesso a elementos pelo índice:

```
print texto1[2]
print 'GTO'[1]
```

Saída:

```
i
T
```

- Acesso a pedaços da string:

```
print texto1[ :2]
print texto1[2: ]
print texto1[2:4]
```

Saída:

```
pr
imeiro texto
im
```

Tipos Básicos - Strings

Texto - Operações

- Funções de procura por substrings:
  - No início:

```
texto1.startswith('pri')
texto1.startswith('cba')
```

Saída:

```
True
False
```

  - No final:

```
texto1.endswith('to')
texto1.endswith('xy')
```

Saída:

```
True
False
```

  - Em qualquer posição:

```
texto1.find('iro')
texto1.find('gto')
```

Saída:

```
5
-1
```

Tipos Básicos - Strings

Texto - Operações

- Funções de verificação de conteúdo:
  - Se são somente letras:

```
'abc'.isalpha()
'123'.isalpha()
'a12'.isalpha()
```

Saída:

```
True
False
False
```

  - Se são somente números:

```
'abc'.isdigit()
'123'.isdigit()
'a12'.isdigit()
```

Saída:

```
False
True
False
```

  - Se são letras e números:

```
'abc'.isalnum()
'123'.isalnum()
'a12'.isalnum()
```

Saída:

```
True
True
True
```

  - Se são somente espaços:

```
'  '.isspace()
' \t\n'.isspace()
'a  '.isspace()
```

Saída:

```
True
True
False
```

Tipos Básicos - Strings

Texto - Operações

- Transformar sequência em texto e texto em sequência:
  - Juntando textos de uma lista em um texto só:

```
print '-'.join(['a','b','cde'])
print 'AB'.join(['1','teste'])
```

Saída:

```
a-b-cde
1ABteste
```

  - Quebrando um texto em uma lista:

```
print 'nome:senha'.split(':')
print 'a/|b/|c'.split('/|')
```

Saída:

```
['nome', 'senha']
['a', 'b', 'c']
```

- Transformar a caixa do texto (será criada uma nova instância):
  - Para maiúscula:

```
print 'abC'.upper()
```

Saída:

```
ABC
```

  - Para minúscula:

```
print 'aBC'.lower()
```

Saída:

```
abc
```

  - Inverter maiúsculas e minúsculas:

```
print 'aBc'.swapcase()
```

Saída:

```
AbC
```

Tipos Básicos - Strings

Texto - Operações

- Verificar se está em maiúscula, minúscula, com a primeira letra maiúscula:

```
'abc'.islower()
'AbC'.islower()
'ABC'.isupper()
'AbcTeste'.istitle()
'Abcteste'.istitle()
```

Saída:

```
True
False
True
False
True
```

- Retirada de caracteres (será criada uma nova instância):
  - Da esquerda:

```
print 'abc'.lstrip()
print '_*_abc_*_'.lstrip('*_')
```

Saída:

```
abc
abc_*_
```

  - Da direita:

```
print 'abc'.rstrip()
print '_*_abc_*_'.rstrip('*_')
```

Saída:

```
abc
_*_abc
```

  - De ambos os lados:

```
print 'abc'.strip()
print '_*_abc_*_'.strip('*_')
```

Saída:

```
abc
abc
```

Tipos Básicos - Strings

Texto - Operações

- Troca de caracteres (será criada uma nova instância):
  - Troca de pedaços:

```
print 'este é um teste'.replace('ste','ABC')
```

Saída:
```
eABC é um teABC
```

  - Troca de caracteres individuais baseados em uma tabela:

```
import string
tabela = string.maketrans('oea','431')
txt = 'hoje o dia é de sol'
print txt.translate(tabela)
```

Saída:
```
h4j3 4 di1 é d3 s4l
```

- Formatação de strings:
  - Operador % é usado para fazer a formatação da string.
  - O formato deve seguir a convenção do printf() do C.
  - Pode-se usar valores nomeados, passando um dicionário.
  - Operadores extra, como o "%r" são usados para a representação do objeto.
  - Mais informações em http://docs.python.org/lib/typesseq-strings.html (item 6.6.2).

Tipos Básicos - Strings

Texto - Formatação

- Operador "%" é usado para fazer interpolação de strings. A interpolação é mais eficiente no uso de memória do que a concatenação convencional.
- Símbolos usados na interpolação:
  - %s: string.
  - %d: inteiro.
  - %o: octal.
  - %x: hexacimal.
  - %f: real.
  - %e: real exponencial.
  - %%: sinal de percentagem.
- Os símbolos podem ser usados para apresentar números em diversos formatos.

Tipos Básicos - Strings

Texto - Formatação

- Exemplos:

```
#Zeros a esquerda
print 'Agora são %02d:%02d.' % (8, 3)
#Real (número após o ponto controla as casas decimais)
print 'Percentagem: %.1f%%, Exponencial:%.2e' % (5.333, 0.00314)
#Octal e hexadecimal
print 'Decimal: %d, Octal: %o, Hexadecimal: %x' % (10, 10, 10)
```

Saída:

```
Agora são 08:03.
Percentagem: 5.3%, Exponencial:3.14e-03
Decimal: 10, Octal: 12, Hexadecimal: a
```

Tipos Básicos - Strings

Texto - Formatação

- A partir da versão 2.6, está disponível outra forma de interpolação além do operador "%", o método de string e a função chamados format().

```
musicos = [('Page', 'guitarrista', 'Led Zeppelin'),
           ('Fripp', 'guitarrista', 'King Crimson')]
#Parâmetros identificados pela ordem
msg = '{0}é {1}do {2}'
for nome, funcao, banda in musicos:
    print(msg.format(nome, funcao, banda))
#Parâmetros identificados pelo nome
msg = '{saudacao}, são {hora:02d}:{minuto:02d}'
print msg.format(saudacao='Bom dia', hora=7, minuto=30)
#Função builtin format()
print 'Pi =', format(3.14159, '.3e')
```

Saída:

```
Page é guitarrista do Led Zeppelin
Fripp é guitarrista do King Crimson
Bom dia, são 07:30
Pi = 3.142e+00
```

Tipos Básicos - Strings

Texto - Exercícios

1. Converta uma string para maiúscula e imprima.
2. Dado o texto “abacate” troque as letras “a” por “4” e imprima.
3. Dado o texto “bin:x:1:1:bin:/bin:/bin/false”, quebre-o na ocorrência de ':'.
4. Dado uma tupla ('a', 'b', 'c'), transforme-a em uma string, separada por '*'.
5. Uma mensagem está “criptografada” usando o ROT13:

   “fr ibpr rfgn yraqb rfgr grkgb, cnenoraf. pnfb anb graun hgvyvmnqb b genafyngr(), gragr qrabib!”.

   Decodifique essa mensagem considerando somente letras minúsculas.

   ROT13 ou “rotate by 13 places” é um esquema de substituição de cifras, bastante usado em foruns on-line. Mais sobre esse esquema em:

   http://en.wikipedia.org/wiki/ROT13

Biblioteca Padrão

Matemática, Arquivos, Sistemas de arquivo, Arquivos compactados, Arquivos de dados, Sistema operacional, Tempo, Expressões regulares.

Biblioteca Padrão

Visão Geral

É comum dizer que o Python vem com “baterias inclusas”, em referência a vasta biblioteca de módulos e pacotes que é distribuída com o interpretador.

Alguns módulos importantes da biblioteca padrão:

- **Matemática**: math, cmath, decimal e random.
- **Sistema**: os, glob, shutils e subprocess.
- Threads: threading.
- Persistência: pickle e cPickle.
- XML: xml.dom, xml.sax e elementTree (a partir da versão 2.5).
- Configuração: ConfigParser e optparse.
- **Tempo**: time e datetime.
- Outros: sys, logging, traceback, types e timeit.

Biblioteca Padrão

Matemática

- Além dos tipos numéricos builtins do interpretador, na biblioteca padrão do Python existem vários módulos dedicados a implementar outros tipos e operações matemáticas.
  - O módulo *math* define funções logarítmicas, de exponenciação, trigonométricas, hiperbólicas e conversões angulares, entre outras.
  - O módulo *cmath*, implementa funções similares, porém feitas para processar números complexos.

```
import math
import cmath

print math.sqrt(2) #raiz quadrada
print math.log(5) #logaritmo

cpx = -2 - 2j #Complexos
plr = cmath.polar(cpx) #Conversão para coordenadas polares

print 'Complexo:', cpx
print 'Forma polar:', plr, '(em radianos)'
print 'Amplitude:', abs(cpx)
print 'Ângulo:', math.degrees(plr[1]), '(graus)'
```

Biblioteca Padrão

Random

Saída:

```
1.41421356237
1.60943791243
Complexo: (-2-2j)
Forma polar: (2.8284271247461903, -2.3561944901923448) (em radianos)
Amplitude: 2.82842712475
Ângulo: -135.0 (graus)
```

- O módulo random traz funções para a geração de números aleatórios.

```
import random
import string

print random.choice(string.ascii_uppercase) #Escolha uma letra
print random.randrange(1, 11) #Escolha um número de 1 a 10
print random.random() #Escolha um float no intervalo de 0 a 1
```

Saída:

```
X
3
0.362231373619
```

Biblioteca Padrão

Decimal

- Na biblioteca padrão ainda existe o módulo decimal, que define operações com números reais com precisão fixa.

```
import decimal

t = 5.
for i in range(50):
    t = t - 0.1
print 'Float:', t

t = decimal.Decimal('5.')
for i in range(50):
    t = t - decimal.Decimal('0.1')

print 'Decimal:', t
```

Saída:

```
Float: 1.02695629778e-15
Decimal: 0.0
```

- Com este módulo, é possível reduzir a introdução de erros de arredondamento originados da aritmética de ponto flutuante.

Biblioteca Padrão

Frações

- Na versão 2.6, também está disponível o módulo fractions, que trata de números racionais.

```
from fractions import Fraction

#Três frações
f1 = Fraction('-2/3')
f2 = Fraction(3, 4)
f3 = Fraction('.25')

print "Fraction('-2/3') =", f1
print "Fraction('3, 4') =", f2
print "Fraction('.25') =", f3

#Soma
print f1, '+', f2, '=', f1 + f2
print f2, '+', f3, '=', f2 + f3
```

Saída:

```
Fraction('-2/3') = -2/3
Fraction('3, 4') = 3/4
Fraction('.25') = 1/4
-2/3 + 3/4 = 1/12
3/4 + 1/4 = 1
```

- As frações podem ser inicializadas de várias formas: como string, como um par de inteiros ou como um número real. O módulo também possui uma função chamada *gcd()*, que calcula o maior divisor comum (MDC) entre dois inteiros.

Biblioteca Padrão

Arquivos

- Os arquivos no Python são representados por objetos do tipo *file*, que oferecem métodos para diversas operações de arquivos. Arquivos podem ser abertos para leitura ('r', que é o default), gravação ('w') ou adição ('a'), em modo texto ou binário('b').
- Em Python:
  - sys.stdin representa a entrada padrão.
  - sys.stdout representa a saída padrão.
  - sys.stderr representa a saída de erro padrão.
- A entrada, saída e erro padrões são tratados pelo Python como arquivos abertos. A entrada em modo de leitura e os outros em modo de gravação.
- Os objetos do tipo arquivo também possuem um método seek(), que permite ir para qualquer posição no arquivo.
- Na versão 2.6, está disponível o módulo io, que implementa de forma separada as operações de arquivo e as rotinas de manipulação de texto.

Biblioteca Padrão

Arquivos

- Exemplo de escrita:

```
import sys

temp = open('temp.txt', 'w') #Criando um objeto do tipo file
#Escrevendo no arquivo
for i in range(5):
    temp.write('%03d\n' % i)
temp.close() #Fechando

temp = open('temp.txt')
#Escrevendo no terminal
for x in temp:
    #Escrever em sys.stdout envia
    #o texto para a saída padrão
    sys.stdout.write(x)
temp.close()
```

Saída:

```
000
001
002
003
004
```

Biblioteca Padrão

Arquivos

- Exemplo de leitura:

```
import sys
import os.path

#raw_input() retorna a string digitada
fn = raw_input('Nome do arquivo: ').strip()
#strip() retira o caractere de nova linha
if not os.path.exists(fn):
    print 'Tente outra vez...'
    sys.exit()

#Numerando as linhas
for i, s in enumerate(open(fn)):
    print i + 1, s,

#Imprime uma lista contendo linhas do arquivo
print open('temp.txt').readlines()
```

Saída:

```
Nome do arquivo: teste.txt
Tente outra vez...
```

```
Nome do arquivo: temp.txt
1 000
2 001
3 002
4 003
5 004
['000\n', '001\n', '002\n', '003\n', '004\n']
```

Biblioteca Padrão

Sistema de arquivos

- Os sistemas operacionais modernos armazenam os arquivos em estruturas hierárquicas chamadas sistemas de arquivo (file systems).

- Várias funcionalidades relacionadas a sistemas de arquivo estão implementadas no módulo os.path, tais como:
  - os.path.basename(): retorna o componente final de um caminho.
  - os.path.dirname(): retorna um caminho sem o componente final.
  - os.path.exists(): retorna True se o caminho existe ou False em caso contrário.
  - os.path.getsize(): retorna o tamanho do arquivo em bytes.

- O glob é outro módulo relacionado ao sistema de arquivo. A função glob.glob() retorna uma lista com os nomes de arquivo que atendem ao critério passado como parâmetro. (semelhante ao “dir” no prompt do windows, ou “ls” no terminal do linux)

Biblioteca Padrão

Sistema de arquivos

- Exemplo:

```
import os.path
import glob

#Mostra uma lista de nomes de arquivos
#e seus respectivos tamanhos
#mostrando os arquivos .py
for arq in sorted(glob.glob('*.py')):
    print arq, os.path.getsize(arq)

#para todos os arquivos do diretório
for arq in sorted(glob.glob('*.*')):
    print arq, os.path.getsize(arq)
```

Saída:

```
Main.py 398
Main.py 398
temp.txt 25
```

Biblioteca Padrão

Arquivos compactados

- O Python possui módulos para trabalhar com vários formatos de arquivos compactados.

- Exemplo de gravação de um arquivo ".zip":

```
import zipfile

texto = """
**************************************
Esse é o texto que será compactado e...
... guardado dentro de um arquivo zip.
**************************************
"""
#Cria um zip novo
zip = zipfile.ZipFile('arq.zip', 'w', zipfile.ZIP_DEFLATED)
#Escreve uma string no zip como se fosse um arquivo
zip.writestr('texto.txt', texto)
#Fecha o zip
zip.close()
```

Biblioteca Padrão

Arquivos compactados

- Exemplo de leitura de um arquivo ".zip":

```
import zipfile

#Abre o arquivo zip para leitura
zip = zipfile.ZipFile('arq.zip')
#Pega a lista dos arquivos compactados
arqs = zip.namelist()
for arq in arqs:
    #Mostra o nome do arquivo
    print 'Arquivo:', arq
#Pegando as informações do arquivo
zipinfo = zip.getinfo(arq)
print 'Tamanho original:', zipinfo.file_size
print 'Tamanho comprimido:', zipinfo.compress_size
#Mostra o conteúdo do arquivo
print zip.read(arq)
```

Saída:

```
Arquivo: texto.txt
Tamanho original: 168
Tamanho comprimido: 84

****************************************

Esse é o texto que será compactado e...

... guardado dentro de um arquivo zip.

****************************************
```

Biblioteca Padrão

Arquivos de dados

- Na biblioteca padrão, o Python também fornece um módulo para simplificar o processamento de arquivos no formato CSV (Comma Separated Values). No formato CSV, os dados são armazenados em forma de texto, separados por vírgula, um registro por linha.

```
import csv

#Dados
dt = (('temperatura', 15.0, 'C', '10:40', '2006-12-31'),
      ('peso', 42.5, 'kg', '10:45', '2006-12-31'))
#A rotina de escrita recebe um objeto do tipo file
out = csv.writer(file('dt.csv', 'w'))
#Escrevendo as tuplas no arquivo
out.writerows(dt)

#A rotina de leitura recebe um objeto arquivo
dt = csv.reader(file('dt.csv'))
#Para cada registro do arquivo, imprima
for reg in dt:
    print reg
```

Saída:

```
['temperatura', '15.0', 'C', '10:40', '2006-12-31']
['peso', '42.5', 'kg', '10:45', '2006-12-31']
```

Biblioteca Padrão

Sistema operacional

```
import os
import sys
import platform

def uid():
    """
    uid() -> retorna a identificação do usuário
    corrente ou None se não for possível identificar
    """
    #Variáveis de ambiente para cada sistema operacional
    us = {'Windows': 'USERNAME',
          'Linux': 'USER'}
    u = us.get(platform.system())
    return os.environ.get(u)

print 'Usuário:', uid()
print 'plataforma:', platform.platform()
print 'Diretório corrente:', os.path.abspath(os.curdir)

exep, exef = os.path.split(sys.executable)
print 'Executável:', exef
print 'Diretório do executável:', exep
```

- Além do sistema de arquivos, os módulos da biblioteca padrão também fornecem acesso a outros serviços providos pelo sistema operacional.

Saída:

```
Usuário: markinho
plataforma: Windows-post2008Server-6.1.7600
Diretório corrente: C:\Users\markinho\Documents\Python\workspace\Aula\Aula\src
Executável: python.exe
Diretório do executável: C:\Python26
```

Biblioteca Padrão

Tempo

- O Python possui dois módulos para lidar com tempo:
  - time: implementa funções que permitem utilizar o tempo gerado pelo sistema.
  - datetime: implementa tipos de alto nível para realizar operações de data e hora.

```
import time

#localtime() Retorna a data e hora local no formato
#de uma estrutura chamada struct_time, que é uma
#coleção com os itens: ano, mês, dia, hora, minuto,
#segundo, dia da semana, dia do ano e horário de verão
print time.localtime()

#asctime() retorna a data e hora como string, conforme
#a configuração do sistema operacional
print time.asctime()

#time() retorna o tempo do sistema em segundos
ts1 = time.time()

#gmtime() converte segundos para struct_time
tt1 = time.gmtime(ts1)
print ts1, '->', tt1
```

Saída:

```
time.struct_time(tm_year=2010, tm_mon=3, tm_mday=7, tm_hour=18, tm_min=1, tm_sec=22, tm_wday=6, tm_yday=66, tm_isdst=0)
Sun Mar 07 18:01:22 2010
1267995682.75 -> time.struct_time(tm_year=2010, tm_mon=3, tm_mday=7, tm_hour=21, tm_min=1, tm_sec=22, tm_wday=6, tm_yday=66, tm_isdst=0)
```

Biblioteca Padrão

Tempo

- Continuando...

```
#Somando uma hora
tt2 = time.gmtime(ts1 + 3600.)

#mktime() converte struct_time para segundos
ts2 = time.mktime(tt2)
print ts2, '->', tt2

#clock() retorma o tempo desde quando o programa
#iniciou, em segundos
print 'O programa levou', time.clock(), \
        'segundos até agora...'

#Contando os segundos...
for i in xrange(5):
    #sleep() espera durante o número de segundos
    #especificados como parâmetro
    time.sleep(1)
    print i + 1, 'segundo(s)'
```

Saída:

```
1268010082.0 -> time.struct_time(tm_year=2010, tm_mon=3, tm_mday=7, tm_hour=22, tm_min=1, tm_sec=22, tm_wday=6, tm_yday=66,
                tm_isdst=0)
O programa levou 3.831203312e-07 segundos até agora...
1 segundo(s)
2 segundo(s)
3 segundo(s)
4 segundo(s)
5 segundo(s)
```

Biblioteca Padrão

Tempo

- Em datetime, estão definidos quatro tipos para representar o tempo:
  - datetime: data e hora.
  - date: apenas data.
  - time: apenas hora.
  - timedelta: diferença entre tempos.

```
import datetime

#datetime() recebe como parâmetros:
#ano, mês, dia, hora, minuto, segundo
#e retorna um objeto do tipo datetime
dt = datetime.datetime(2020, 12, 31, 23, 59, 59)

#Objetos date e time podem ser criados
#a partir de um objeto datetime
data = dt.date()
hora = dt.time()

#Quanto tempo falta para 31/12/2020
dd = dt - dt.today()

print 'Data:', data
print 'Hora:', hora
print 'Quanto tempo falta para 31/12/2020:', \
        str(dd).replace('days', 'dias')
```

Saída:

```
Data: 2020-12-31
Hora: 23:59:59
Quanto tempo falta para 31/12/2020: 3952 dias, 5:45:22.123000
```

Biblioteca Padrão

Expressões regulares

- Expressão regular é uma maneira de identificar padrões em sequências de caracteres.

- Principais caracteres:
  - Ponto (.): Em modo padrão, significa qualquer caractere, menos o de nova linha.
  - Circunflexo (^): Em modo padrão, significa inicio da string.
  - Cifrão ($): Em modo padrão, significa fim da string.
  - Contra-barra (\): Caractere de escape, permite usar caracteres especiais como se fossem comuns.
  - Colchetes ([]): Qualquer caractere dos listados entre os colchetes.
  - Asterisco (*): Zero ou mais ocorrências da expressão anterior.
  - Mais (+): Uma ou mais ocorrências da expressão anterior.
  - Interrogação (?): Zero ou uma ocorrência da expressão anterior.
  - Chaves ({n}): n ocorrências da expressão anterior.
  - Barra vertical (|): “ou” lógico.
  - Parenteses (()): Delimitam um grupo de expressões.
  - \d: Dígito. Equivale a [0-9].
  - \D: Não dígito. Equivale a [^0-9].
  - \s: Qualquer caractere de espaçamento ([\t\n\r\f\v]).
  - \S: Qualquer caractere que não seja de espaçamento.([^ \t\n\r\f\v]).
  - \w: Caractere alfanumérico ou sublinhado ([a-zA-Z0-9_]).
  - \W: Caractere que não seja alfanumérico ou sublinhado ([^a-zA-Z0-9_]).

Biblioteca Padrão

Expressões regulares

- Exemplo:

```
import re

#Compilando a expressão regular
#Usando compile() a expressão regular fica compilada
#e pode ser usada mais de uma vez
rex = re.compile('\w+')

#Encontra todas as ocorrências que atendam a expressão
bandas = 'Yes, Genesis & Camel'
print bandas, '->', rex.findall(bandas)

#Identifica as ocorrências de Björk (e suas variações)
bjork = re.compile('[Bb]j[öo]rk')
for m in ('Björk', 'björk', 'Biork', 'Bjork', 'bjork'):
    #match() localiza ocorrências no inicio da string
    #para localizar em qualquer parte da string, use search()
    print m, '->', bool(bjork.match(m))

#Substituindo texto
texto = 'A próxima faixa é Stairway to Heaven'
print texto, '->', re.sub('[Ss]tairway [Tt]o [Hh]eaven',
                          'The Rover', texto)
#Dividindo texto
bandas = 'Tool, Porcupine Tree e NIN'
print bandas, '->', re.split(',?\s+e?\s+', bandas)
```

Saída:

```
Yes, Genesis & Camel -> ['Yes', 'Genesis', 'Camel']
Björk -> True
björk -> True
Biork -> False
Bjork -> True
bjork -> True
A próxima faixa é Stairway to Heaven -> A próxima faixa é The Rover
Tool, Porcupine Tree e NIN -> ['Tool, Porcupine Tree', 'NIN']
```

Biblioteca Padrão

Exercícios

1. Crie um arquivo compactado contendo um arquivo de texto com uma lista de dados de plataforma, usuário, o diretório de trabalho e a data e hora completas do momento da operação no formato do sistema, em cada linha do arquivo.

2. Converta os segundos armazenados no exercício 1 para hexadecimal e octal.

3. Converta o número complexo 3.45 + 2.3j para a forma polar, com o ângulo em graus, multiplique esse ângulo a um numero real randômico entre 0 e 1 e mostre na tela com 2 casas decimais.

4. Salve os dados dos exercícios 2 e 3 em um arquivos de dados do tipo CSV, com a descrição de cada dado.

FINAL DA PARTE I

Referências:

Python para Desenvolvedores / Luiz Eduardo Borges
Rio de Janeiro, Edição do Autor, 2010
ISBN 978-85-909451-1-6

Site oficial

A edição mais recente está disponível no formato PDF em:
http://ark4n.wordpress.com/python/