Antonio Sérgio Nogueira

Programando em Python®
Do Básico à WEB

1ª Edição
2012

Nota do Autor

Agradeço a todos os que me cercam e que tem compartilhado comigo este sonho de fornecer meus conhecimentos e pesquisas na área de informática.
Aos leitores espero que saibam aproveitar este material didático escrito com amor, carinho e muito trabalho.
A minha filha Vanessa Ota Nogueira o meu especial agradecimento por confeccionar a capa deste material didático.

Antonio Sérgio Nogueira

Sumário

Introdução

Este texto não tem a pretensão de ser uma bíblia da linguagem Python, mas ele aborda a linguagem em sua extensão, mostrando os aspectos mais peculiares como:

- O interpretador Python e o ambiente para edição e execução IDLE;
- A estrutura básica da linguagem e seus tipos de dados;
- A definição de bloco por indentação ao invés de marcadores de início e fim de bloco;
- A orientação a objetos;
- A interface gráfica Tkinter;
- E por fim como desenvolver um programa para Web.

Lançada por Guido van Rossum em 1991, o Python atualmente possui um modelo de desenvolvimento comunitário e aberto, gerenciado pela organização sem fins lucrativos Python Software Foundation. Python é simples de usar, e é uma verdadeira linguagem de programação que oferece tipos nativos de alto nível como flexíveis vetores, matrizes e dicionários. Por ser uma linguagem interpretada de alto nível, orientada a objetos e de tipagem dinâmica e forte. Python é muito mais fácil de usar do que as linguagens compiladas e o interpretador pode ser usado interativamente, o que torna fácil experimentar diversas características da linguagem, escrever e testar programas e funções em um desenvolvimento *bottom-up*.

O nome da linguagem não tem nada a ver com os famosos répteis e sim com o famoso show da BBC "Monty Python's Flying Circus". Mas neste exato momento se você

ainda não se convenceu com a linguagem Python, saiba que a Nasa e a Industrial Light & Magic, a companhia de efeito visual que criou Star Wars, usa Python. Agora que você já se convenceu, que tal continuar a examinar o texto com maior detalhe e testar o interpretador Python com os exemplos mostrados no texto.

O Python e sua extensa biblioteca padrão estão disponíveis na forma de código fonte ou binário para a maioria das plataformas a partir do site, http://www.python.org/, e deve ser distribuídos livremente. No mesmo sítio estão disponíveis distribuições e referências para diversos módulos, programas, ferramentas e documentação adicional contribuídos por terceiros. Acesse as documentações através de http://python.org/doc/ onde você vai encontrar: Python Library Reference, Python Reference Manual, Extending and Embedding the Python Interpreter, Python/C API Reference.

1. O interpretador Python

1.1 Sua história

No final de 1989 Guido van Rossum criou o Python no Instituto de Pesquisa nacional para Matemática e Ciência da Computação (CWI), nos Países Baixos, um sucessor da linguagem chamada de ABC. Tendo como foco principal o aumento de produtividade do programador, em 1991, Guido publicou o código (versão 0.9.0) no grupo de discussão *alt.sources*. Em 1994 formou-se o principal fórum de discussão do Python, *comp.lang.python*, isto foi o marco para o crescimento de usuários da linguagem. A versão 1.0 foi lançada em janeiro de 1994. Além das funcionalidades que já estavam presentes como classes com herança, tratamento de exceções, funções e os tipos de dados nativos, lista, dicionários e strings, sistema de módulos e assim por diante, esta nova versão incluía ferramentas para programação funcional como lambda, `map`, `filter` e `reduce`. A última versão que Guido lançou enquanto estava na CWI foi o Python 1.2. Guido continuou o trabalho, em 1995, no CNRI em Reston, USA, de onde lançou diversas versões. Na versão 1.4 a linguagem ganhou parâmetros nomeados[1] e suporte nativo a números complexos, assim como uma forma de encapsulamento (Versão 1.6 última versão lançada no CNRI)

A partir de 2000, o de desenvolvimento da linguagem se mudou para a BeOpen afim de formar o time PythonLabs. O único lançamento na BeOpen foi o Python 2.0, e após o lançamento o grupo de desenvolvedores da PythonLabs agrupou-se na Digital Creations. O Python 2.0 implementou list

1 A capacidade de passar parâmetro pelo nome e não pela posição na lista de parâmetros.

comprehension, uma relevante funcionalidade de linguagens funcionais como SETL e Haskell, e essa versão 2.0 também introduziu um sistema coletor de lixo capaz de identificar e tratar ciclos de referências. O lançamento incluiu a mudança na especificação para suportar escopo aninhado, assim como outras linguagens com escopo estático. Uma grande inovação da versão 2.2 foi a unificação dos tipos Python (escritos em "C") e classes (escritas em Python) em somente uma hierarquia, com isto o modelo de objetos do Python torna-se consistentemente orientado a objeto e foi adicionado generator, inspirado em Icon. O incremento da biblioteca padrão e as escolhas sintáticas foram fortemente influenciadas por Java em alguns casos: o pacote `logging` introduzido na versão 2.3, o analisador sintático SAX, introduzido na versão 2.0 e a sintaxe de decoradores que usa `@`, adicionadas na versão 2.4. Em 1 de outubro de 2008 foi lançada a versão 2.6, já visando a transição para a versão 3.0 da linguagem. Entre outras modificações, foram incluídas bibliotecas para multiprocessamento, JSON, E/S, além de uma nova forma de formatação de cadeia de caracteres.
Atualmente a linguagem é usada em diversas áreas, como servidores de aplicação e computação gráfica. Está disponível como linguagem script em aplicações como Open Office (Python UNO Bridge) e pode ser utilizada em procedimentos armazenados no sistema gerenciador de banco de dados PostgreSQL(PL/Python). [WIKIPEDIA – 27/11/200].

Existem diversas implementações do PYTON são elas:

Cpython – esta é a versão original escrita em "C".

ActivePython - é um padrão industrial do Python, disponível para Windows, Linux, Mac OS X, Solaris, AIX e HP-UX.

Jython – esta é a versão escrita em Java e pode ser usada em implementações que usam a biblioteca de classes Java. Veja o Website do Jython.

Python for .NET – usa a implementação Cpython, mas é gerenciada por uma aplicação .NET e disponibiliza as bibliotecas .NET. Veja Python for .NET

IronPython – Diferente do Python .Net , ela é uma implementação completa de Python que gera a Linguagem Intermediária, e compila o código Python diretamente para o assembler .NET.

PyPy - Uma implementação do python escrita em Python, até o interpretador de bytecode é escrito em Python. Ela é executada usando o Cpython como um interpretador subjacente. Um dos objetivos do projeto é incentivar a experimentação com a linguagem própria, tornando-a mais fácil de modificar o interpretador (uma vez que é escrito em Python). Informação adicional está disponível na página inicial do projeto PyPy .

1.2 Licença

No que tange as licenças de uso a versão 1.6.1 do Python passou a ser compatível com a GPL, e na versão 2.1 a licença foi renomeada para Python Foundation License, em 2001. Todos os direitos de propriedade intelectual adicionados deste ponto em diante, começando com Python 2.1 e suas versões alfa e beta, são de titularidade da Python Software Foundation (PSF), uma associação sem fins lucrativos organizada sob inspiração da

Apache Software Foundation (ASF). Veja http://www.python.org/psf/ para mais informações sobre PSF.

TERMOS E CONDIÇÕES PARA ACESSAR OU DE OUTRA FORMA UTILIZAR PYTHON

==========

CONTRATO DE LICENÇA PSF

==========

1. Este CONTRATO DE LICENÇA realiza-se entre Python Software Foundation (“PSF”) e o Indivíduo ou Organização (“Licenciado”) acessando ou de outra forma utilizando o programa de computador Python 2.1.1 em forma binária ou código-fonte e sua documentação associada.
2. Nos termos e condições deste Contrato de Licença, PSF outorga ao Licenciado, por este instrumento, uma Licença não exclusiva, sem encargos patrimoniais (royalties), de abrangência mundial para reproduzir, analisar, testar, executar ou expor publicamente, preparar obras derivadas, distribuir e de outra forma utilizar Python 2.1.1, isolado ou em qualquer versão derivada, contanto que, porém, este Contrato de Licença PSF e o aviso de direitos autorais PSF, i.e., “Copyright © 2001 Python Software Foundation, All Rights Reserved” sejam incluídos em Python 2.1.1, isolado ou em qualquer versão derivada preparada pelo Licenciado.
3. Caso o Licenciado prepare uma obra derivada baseada em Python 2.1.1 ou que o incorpore por

inteiro ou qualquer trecho seu, e deseje tornar esta obra derivada disponível a outrem como aqui previsto, então o Licenciado concorda em incluir em tal obra um breve sumário das mudanças feitas sobre Python 2.1.1.

4. PSF torna Python 2.1.1 disponível ao Licenciado "COMO ESTÁ". PSF NÃO OFERECE QUAISQUER GARANTIAS OU DECLARAÇÕES, EXPRESSAS OU TÁCITAS. COMO EXEMPLO, MAS NÃO LIMITAÇÃO, PSF NÃO OFERECE E SE ISENTA DE QUAISQUER GARANTIAS OU DECLARAÇÕES DE COMERCIABILIDADE OU ADEQUAÇÃO A FINALIDADES ESPECÍFICAS, OU DE QUE O USO DE PYTHON 2.1.1 NÃO VIOLARÁ QUAISQUER DIREITOS DE TERCEIROS.

5. PSF NÃO SERÁ RESPONSÁVEL PERANTE O LICENCIADO OU QUAISQUER OUTROS USUÁRIOS DE PYTHON 2.1.1 POR PERDAS E DANOS, SEJAM INCIDENTAIS, ESPECIAIS OU CONSEQUENTES, COMO RESULTADO DE MODIFICAÇÃO, DISTRIBUIÇÃO, O OUTRA FORMA DE UTILIZAÇÃO DE PYTHON 2.1.1, OU QUALQUER DE SUAS OBRAS DERIVADAS, MESMO QUE HOUVESSE SIDO AVISADA DESTA POSSIBILIDADE.

6. Este Contrato de Licença será automaticamente rescindido em caso de violação material de seus termos e condições.

7. Nada neste Contrato de Licença pode ser interpretado de forma a criar qualquer relação de

agência, parceria ou joint-venture entre PSF e o Licenciado. Este Contrato de Licença não outorga permissão para usar marcas ou nomes comerciais de PSF como conjunto distintivo para endossar ou promover produtos ou serviços do Licenciado ou de qualquer terceiro.

8. Ao copiar, instalar ou de outra forma utilizar Python 2.1.1, o Licenciado obriga-se aos termos e condições deste Contrato de Licença.

==========

1.3 Uso do Interpretador Python

1.3.1 Qual Python usar?

O Windows não vem com Python. Mas não se preocupe! Há diversas formas fáceis de entrar no mundo Python usando Windows. Como você pode ver, Python roda em muitos sistemas operacionais. A lista completa inclui Windows, o Mac OS, o Mac OS X e todos os sistemas gratuitos compatível com UNIX como o próprio Linux. Há também versões que rodam no Sun Solaris, AS/400, Amiga, OS/2, BeOS e muitas outras plataformas que você provavelmente nunca ouviu falar.

1.3.2 Python em Windows

No Windows, você tem diversas escolhas para instalar o Python. A ActiveState tem um instalador Python para Windows que inclui a versão completa do Python, um IDE com um editor de código e extensões Windows para o Python que permitem acesso a serviços específicos do Windows, suas APIs e o

registro. O ActivePython pode ser baixado gratuitamente, mas não é open source. Se você realmente precisa da última versão do Python pule para a opção 2. A segunda opção é usar o instalador Python "oficial", distribuído pelos próprios desenvolvedores do Python. Esse instalador pode ser baixado gratuitamente, tem código fonte aberto e está sempre atualizado.

1.3.2.1 Opção 1: Instalando o ActivePython

1. Baixe o ActivePython em http://www.activestate.com/Products/ActivePython/.
2. Se você usa Windows 95, Windows 98 ou Windows ME, deve instalar o Windows Installer 2.0 antes de continuar.
3. Dê um clique duplo no arquivo ActivePython-2.6.1.1-win32-ix86.msi.
4. Siga as instruções na tela.
5. Se seu espaço em disco for limitado, é possível fazer uma instalação personalizada ("custom") e deixar de instalar a documentação, mas isso não é recomendado.
6. Após o término da instalação, feche o instalador e abra Iniciar->Programas->ActiveState ActivePython 2.2->PythonWin IDE.

Example IDE ActivePython

PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>>

1.3.2.2 Opção 2: Instalando o Python do Python.org

1. Baixe o instalador Windows do Python em
http://www.python.org/download/releases/2.7.2/
2. Execute o arquivo Python-2.7.2
3. Siga as instruções na tela.
4. Após o término da instalação, feche o instalador e abra Iniciar->Programas->Python 2.7->IDLE
(Python GUI).

Figura 1: IDLE (Python GUI)

2. Python: Introdução, Tipos e Comandos Básicos

2.1 Interpretador

Python é uma linguagem interpretada, o que significa que o código não precisa ser compilado para que seja executado. Assim, o interpretador lê e executa o código diretamente. Linguagens interpretadas, normalmente, funcionam através de 'Compilação Just- In-Time' ou 'Interpretação pura ou em Bytecode'. Você pode criar seu arquivo Python e salvá-lo com a extensão ".py" ou ainda pode executar no modo shell, ou seja, você digita o código diretamente no interpretador. Python é multi plataforma, roda em Windows, Linux, Unix, Macintosh, etc. Pode ser utilizado um ambiente para edição e execução como o IDLE, que é nativo da linguagem e para acessá-lo em Windows basta ir em INICIAR>PROGRAMAS>PYTHON 2.7>IDLE. Em computadores com Windows, Python é instalado geralmente em 'C:\Python27', apesar de você poder mudar isso enquanto está executando o instalador. Para adicionar esse diretório ao *path*, você pode digitar o seguinte comando no DOS: set path=%path%;C:\python27

2.2 Palavras Reservadas

São palavras que não podem ser usadas para dar nomes aos objetos. São: **and, assert, break, class, continue, del, def, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while.**

2.3 Digitando comandos no IDLE

Podemos usar o IDLE como uma calculadora,

visualizando resultados e usando os resultados em cálculos futuros.

```
>>> a=1
>>> b=2
>>> a+b
>>> a/b
>>> a*b
```

2.4 Tipos de Dados

São categorias de valores que são processados de forma semelhante. Por exemplo, números inteiros são processados de forma diferente dos números de ponto flutuante(decimais) e dos números complexos.
Tipos primitivos: são aqueles já embutidos no núcleo da linguagem
Simples: números (int, long, float, complex) e cadeias de caracteres (strings)

Int: números inteiros de *precisão fixa*
1 , 2 , 15 , 19
Long: números inteiros de *precisão arbitrária*
1L , 10000L , -9999999L

Floats: números racionais de precisão *variável.*
1.0 , 10.5 , -19000.00005 , 15e-5

Complex: números complexos.
1+1j , 20j , 1000+100J

Compostos: listas, dicionários, tuplas e conjuntos.

Tipos definidos pelo usuário: são correspondentes a classes (orientação objeto).

2.5 Número ponto flutuante e complexo

Há total suporte para ponto flutuante; operadores com operandos de diferentes tipos convertem o inteiro para ponto flutuante:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Números complexos também são suportados; números imaginários são escritos com o sufixo ‘j’ ou ‘J’. Números complexos com parte real não nula são escritos como ‘(*real*+*imag*j)’, ou podem ser criados pela chamada de função ‘complex(*real*, *imag*)’.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
```

```
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Números complexos são sempre representados por dois números ponto flutuante, a parte real e a parte imaginária. Para extrair as partes de um número *z*, utilize *z*.real e *z*.imag.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

As funções de conversão para ponto flutuante e inteiro, float(), int() e long() não funcionam para números complexos, pois não existe maneira correta de converter um número complexo para um número real. Utilize abs(*z*) para obter sua magnitude (como ponto flutuante) ou z.real para obter sua parte real.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)
5.0
```

No modo interativo, a última expressão a ser impressa é atribuída a variável[2] "**_**" . Isso significa que ao utilizar Python como uma calculadora, é muitas vezes mais fácil prosseguir com os cálculos da seguinte forma:

```
>>> taxa = 12.5 / 100
>>> preco = 100.50
>>> preco* taxa
12.5625
>>> preco + _
113.0625
>>> round(_, 2)
113.06
```

Essa variável especial deve ser tratada somente para leitura pelo usuário. Nunca lhe atribua explicitamente um valor, do contrário estaria criando uma outra variável (homônima) independente, que mascararia o comportamento mágico da variável especial.

2.6 Strings

Além de números, Python também pode manipular strings[3]. O IDLE não está preparado para trabalhar com caracteres acentuados. Durante a execução de um programa em Python, use este comando na primeira linha para acentuar os caracteres: **# -*- coding: cp1252 -*-**

2 Local de memória usado para guardar valores.

3 Strings são sequências de caracteres entre aspas ou apóstrofo, modificáveis apenas com atribuição de novos valores.

```
>>> ’mais menos’
’mais menos’

>>> “vida's”
"vida's"
```

```
>>> ’”OLA”, BOM DIA.’
’”OLA”, BOM DIA.’

>>> "\"OLA,\" BOM DIA."
’"OLA," BOM DIA.’
```

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Terminadores de linha podem ser embutidos na string com barras invertidas.

```
# -*- coding: cp1252 -*-
#Programa que imprime oi
oi = "Esta é uma string longa contendo\n\
diversas linhas de texto assim como você faria em C.\n\
 Observe que os espaços em branco no inicio da linha são \
significativos."
print oi
```

Resultado da execução:

Esta é uma string longa contendo
diversas linhas de texto assim como você faria em C.
Observe que os espaços em branco no inicio da linha são significativos.

Observe que terminadores de linha ainda precisam ser embutidos na string usando \n; a quebra de linha após a última barra de escape seria ignorada.

No entanto, se a tornarmos uma string "crua" (*raw*), as sequências de \n não são convertidas para quebras de linha. Tanto a barra invertida quanto a quebra de linha no código-fonte são incluídos na string como dados. Portanto, o exemplo:

```
>>>oi = r"Esta eh uma string longa contendo\n\
diversas linhas de texto assim como voce faria em C.\n\
 Observe que os espaços em branco no inicio da linha são \
significativos."
>>>print oi

Resultado:
Esta eh uma string longa contendo\n\
diversas linhas de texto assim como voce faria em C.\n\
  Observe que os espaços em branco no inicio da linha são \
significativos.
```

As strings também podem ser delimitadas por pares de aspas ou apóstrofos tríplices: **"""**ou **'''**. Neste caso não é necessário embutir terminadores de linha, pois o texto da string será tratado fielmente como o original.

```
>>> print """ ola
tudo bem e ai como
vai."""

Resultado:
```

```
 ola
tudo bem e ai como
vai.
```

O interpretador imprime o resultado de operações sobre strings da mesma forma que as strings são formatadas na digitação: dentro de aspas, e com caracteres especiais embutidos em *escape sequences*[4], para mostrar seu valor com precisão a string será delimitada por aspas duplas se ela contém um único caractere de aspas simples e nenhum de aspas duplas, caso contrário a string será delimitada por aspas simples[5]. Strings podem ser concatenadas (coladas) com o operador +, e repetidas com *:

```
>>> palavra='palavra '+'A'
>>> palavra
'palavra A'

>>> '<'+palavra * 5 + '>'
'<palavra Apalavra Apalavra Apalavra Apalavra A>'

>>>print "oi" 'vida'
oi vida
>>> print 'oi' 'vida' # Duas strings literais justapostas são automaticamente concatenadas.
oivida
>>>
```

4 *Caractere especial como \n – nova linha*

5 O comando print, descrito posteriormente, pode ser utilizado para escrever strings sem aspas ou *escape sequences "\n"*.

Strings podem ser indexadas; como em C, o primeiro índice da string é o 0. Não existe um tipo separado para caracteres; um caractere é simplesmente uma string unitária. Assim como na linguagem Icon, substrings[6] podem ser especificadas através da notação *slice*[7]: dois índices separados por dois pontos.

```
>>> palavra[4]
'v'
>>> palavra[0:2]
'pa'
>>> palavra[2:4]
'la'
```

Índices de fatias seguem uma padronização útil; a omissão do primeiro índice equivale a zero, a omissão do segundo índice equivale ao tamanho da string que está sendo fatiada.

```
>>> palavra[:2] # os dois primeiros caracteres
'pa'

>>> palavra[2:]  # todos menos os 2 primeiros
'lavra A'

>>> palavra[-1] # último caractere
'A'

>>> palavra[:-2] # todos menos os 2 últimos
'palavra'
```

6 Uma pedaço da string.

7 Fatiar.

```
>>> palavra[-1::-1] # imprime ao contrário a palavra
'A arvalap'
```

Os principais métodos das strings são:

capitalize - transforma o primeiro caractere em maiúscula.
count - conta o número de ocorrências da substring.
upper - transforma string em maiúscula.

```
>>> a="sergio"

>>> a.capitalize()
'Sergio'

>>> a.count('e')
1

>>> a.upper()
'SERGIO'
```

2.7 Strings Unicode

A partir de Python 2.0 um novo tipo foi introduzido: o objeto Unicode. Ele pode ser usado para armazenar e manipular dados Unicode (veja http://www.unicode.org/) e se integra bem aos demais objetos strings pré existentes, de forma a realizar auto conversões quando necessário. Unicode tem a vantagem de prover um único número ordinal para cada caractere usado em textos modernos ou antigos. Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u”Ola Vida !”
u’Ola Vida !’
```

O ‘u’ antes das aspas ou apóstrofo indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Python *Unicode-Escape*.

```
>>> u’Ola\u0020Vida !’
u’Ola Vida !’
```

O código de escape \u0020 indica a inserção do caracter Unicode com valor ordinal 0x0020[8] na posição determinada. Além dessa codificação padrão, Python oferece um outro conjunto de maneiras de se criar strings Unicode.

A função interna unicode() provê acesso a todos os Unicode codecs[9] registrados. Alguns dos mais conhecidos codecs são : *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*. Os dois últimos são codificações de tamanho variável para armazenar cada caractere Unicode em um ou mais bytes. A codificação padrão é ASCII, que trata normalmente de caracteres no intervalo de 0 a 127 mas rejeita qualquer outro com um erro. Quando uma string Unicode é impressa, escrita em arquivo ou convertida por str(), a codificação padrão é utilizada.

```
>>> u"abc"
u’abc’
```

8 Espaço em branco.
9 Coders and Decoders – Codificadores e decodificadores.

```
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'

>>> str(u"äöü")

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    str(u"äöü")
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

Erro - caractere não é padrão ASCII

Para se converter uma string Unicode em uma string 8-bits, usando uma codificação específica, basta invocar o método encode() de objetos Unicode, passando como parâmetro o nome da codificação destino. É preferível utilizar nomes de codificação em letras minúsculas.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Também pode ser utilizada a função unicode() para efetuar a conversão de uma string em outra codificação. Neste caso, o primeiro parâmetro é a string a ser convertida e o segundo o nome da codificação almejada. O valor de retorno da função é a string na nova codificação.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

2.8 Listas

A mais versátil estrutura de dados do Python é a lista (*list*), que pode ser escrita como uma lista de valores separados por vírgula e entre colchetes. Mais importante, os valores contidos na lista não precisam ser do mesmo tipo e podem ter seus elementos modificados.

```
>>> lista = [1,"um",1.0]
>>> lista.insert(3,"hum")
>>> print lista
[1, 'um', 1.0, 'hum']
```

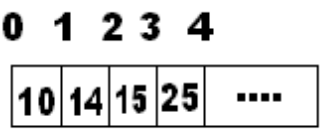
Uma variável chamada lista retorna uma lista com um valor inteiro, uma string e um valor float. Em seguida, foi inserida no final da lista (o número '3' indica a posição três começando do zero) uma outra string. Listas possuem alguns métodos:

append() - adiciona um elemento ao fim da lista.
count() - retorna o número de vezes que um determinado elemento aparece na lista.
extend() - estende a lista com novos elementos passados.
index() - retorna o índice (ou posição) de um determinado elemento da lista.
insert() - insere um determinado elemento numa especificada posição.
pop() - remove o elemento da posição indicada e o retorna.
remove() - remove o primeiro elemento da lista.
reverse() - inverte a ordem dos elementos da lista.

sort() - ordena os elementos da lista.

```
>>> l = [10,56,32,89,25,14]
>>> l.append(90)                    #append
>>> l
[10, 56, 32, 89, 25, 14, 90]

>>> l.insert(3,50)                  #insert
>>> l
[10, 56, 32, 50, 89, 25, 14, 90]

>>> l.pop(7)                        #pop
90
>>> l
[10, 56, 32, 50, 89, 25, 14]

>>> l.index(89)                     #index
4

>>> a = [15,90,68]
>>> a.extend(l)                     #extend
>>> a
[15, 90, 68, 10, 56, 32, 50, 89, 25, 14, '5', '5']

>>> a.sort()                        #sort
>>> a
[10, 14, 15, 25, 32, 50, 56, 68, 89, 90, '5', '5']
```

Da mesma forma que índices de string, índices de lista começam do 0, listas também podem ser concatenadas e sofrer o operadoções de *slice(cortes)*.



Este exemplo é para entender o slice.

```
>>>A[0:3]
[10,14,15]
```

2.9 Tuplas

São objetos como as listas, com a diferença de que tuplas são imutáveis como strings. Uma vez criadas, não podem ser modificadas. Usa-se ou não () como delimitador.

```
>>> tupla = ("Maria", "Pedro", "José")
>>> tupla[0]
'Maria'
>>> tupla[0]="Fátima"
Traceback (most recent call last):
File "<pyshell#25>", line 1, in <module>
tupla[0]="Fátima"
TypeError: 'tuple' object does not support item assignment
Erro - na tentativa de atribuir um novo valor.
>>> tupla= 1,2,3
>>>tupla
(1,2,3)
```

Um problema especial é a criação de tuplas contendo 0

ou 1 itens: a sintaxe tem certos truques para acomodar estes casos. Tuplas vazias são construídas por uma par de parênteses vazios. E uma tupla unitária é construída por um único valor e uma vírgula ou isto entre parênteses (sem a vírgula a tupla não será gerada!). Feio, mas efetivo:

```
>>> vazio = ()
>>> unica = ’oi’,
>>> len(vazio)
0
>>> len(unica)
1
>>> unica
(’oi’,)
```

O comando t = 123,'oi' é um exemplo de empacotamento em tupla[10]: os valores 123 e ’oi’ são empacotados juntos em uma tupla. A operação inversa também é possível desempacotamento de sequência[11] :

```
>>> t=123,'oi'
>>> x,y=t
>>> x
123
>>> y
'oi'
```

2.10 Dicionários

10 Tuple packing.
11 Sequence unpacking.

Dicionários são conjuntos não ordenados de pares, onde o primeiro elemento do par é o índice chamado de chave e o segundo de valor. Um dicionário, em Python, suporta qualquer tipo de objeto, seja ele uma lista ou até mesmo outros dicionários e pode ser modificado. Para criar um dicionário basta declarar pares "chave:valor" separados por vírgula e delimitados por chaves.

```
>>> dic = {1:'um', 2:'dois', 3:'três'}
>>> dic[1]
'um'
>>> dic[4]='quatro'
>>> dic
{1: 'um', 2: 'dois', 3: 'tr\xeas', 4: 'quatro'}
```

Os dicionários possuem alguns métodos, entre eles:

items() - esse método retorna uma lista de tuplas, ou seja, todos os pares chave:valor na forma de tuplas;

```
>>> dic.items()
[(1, 'um'), (2, 'dois'), (3, 'tr\xeas'), (4, 'quatro')]
```

keys() - esse método retorna somente as chaves;

```
>>> dic.keys()
[1, 2, 3, 4]
```

values() - esse método retorna somente os valores;

```
>>> dic.values()
['um', 'dois', 'tr\xeas', 'quatro']
```

get(chave) - retorna o valor da chave passada como parâmetro;

```
>>> dic.get(2)
'dois'
>>> print dic.get(5)
None
```

has_key(chave) - verifica se existe a chave passada como parâmetro, retornando true ou false;

```
>>> dic.has_key(5)
False
>>> dic.has_key(2)
True
```

update(dicionário) - atualiza um dicionário com base em outro passado como parâmetro. Caso elementos do primeiro dicionário existam também no segundo, esse sobrescreve o primeiro, ou se o segundo conter elementos exclusivos, serão adicionados ao primeiro dicionário.

```
>>> dic = {1:'um', 2:'dois', 3:'três'}
>>> dic2 = {1:'one', 2:'two', 5:'five'}
>>> dic.update(dic2)
>>> dic
{1: 'one', 2: 'two', 3: 'tr\xeas', 5: 'five'}
```

dict() - produz dicionários diretamente a partir de uma lista de chaves-valores, armazenadas como tuplas.

```
>>> dict([('sapo', 4139), ('pato', 4127), ('gato', 4098)])
{'sapo': 4139, 'gato': 4098, 'pato': 4127}

>>> dict([(x, x**2) for x in (2, 4, 6)])   # usando construtor de
                                           #processamento de lista
{2: 4, 4: 16, 6: 36}

>>>dict(sapo=10,lua=5,cidade=3)
{'lua': 5, 'cidade': 3, 'sapo': 10}
```

2.11 DocString e comentário

Docstring - São strings que servem para documentar código. Basta criar a string entre sequências de três aspas.

```
>>> def m():
"""
Autor : Nogueira
Data : 23/03/2009
Versao: 0.0
"""
print "Minha função"

>>m.__doc__                    #mostra documentação da função
Autor : Nogueira
Data : 23/03/2009
```

```
Versao: 0.0
```

Comentário(#) - usado para fazer comentário em Python, tudo após este simbolo é considerado como comentário na linha.

```
# primeiro comentário

SPAM = 1      # e esse é o segundo comentário

# ... e ainda um terceiro !

STRING = "# Este não é um comentário."
```

2.12 Operadores Aritméticos

+	adição
-	subtração
*	multiplicação
/	divisão
%	módulo
**	exponenciação

Os operadores + e * fazem concatenação de strings, listas e tuplas.

```
>>> 2**3            # É o mesmo que dois ao cubo
8
```

```
>>> 2**(3+6)      # Dois elevado a nove
512
>>> 7 % 2         # O resto da divisão entre 7 e 2
1
```

O sinal de igual (‘=’) é utilizado para atribuição de um valor a uma variável.

```
>>> largura = 20
>>> altura = 5*9
>>> largura * altura
900
```

Um valor pode ser atribuído a diversas variáveis simultaneamente:

```
>>> x = y = z = 0        # 0 é atribuído a  x, y e z
>>> x
0
>>> y
0
>>> z
0
```

2.13 Operadores de atribuição

+=	Soma com valor atribuído
-=	Subtrai o valor atribuído
/=	Divide pelo valor atribuído

*=	Multiplica pelo valor atribuído

```
>>> a=5
>>> b=6
>>> a+=b     #  a=a+b
>>> a
11
>>> b
6
>>> a-=b     # a=a-b
>>> a
5
>>> a=10
>>> b=5
>>> a/=b     # a=a/b inteiro
>>> a
2
>>> a*=b     # a=a*b
>>> a
10
```

2.14 Operadores de condição

= =	Igual
! =	Diferente
>	Maior
<	Menor
>=	Maior ou igual

<=	Menor ou igual
in	Está dentro da sequência ou do dicionário

```
>>> 5 in (2,3,5)
True
>>> "m" in "Jorge"
False
>>> a=1
>>> b=2
>>> a == b # == testa se a é igual a b
False
>>> a != b # != testa se a é diferente de b
True
>>> a <> b # <> também testa se a é diferente de b
True
>>> a > b # > testa se a é maior que b
False
>>> a < b # < testa se a é menor que b
True
>>> 2*a >= b # testa se o dobro de a é maior ou igual a b
True
```

Em Python, a sintaxe a seguir é válida:

```
>>> if 0<x<10:
        print x
```

2.15 Operadores lógicos

and	e
or	ou
not	não
is	é
Is not	não é

2.15.1 Expressões boolenas

Também chamadas expressões lógicas. Resultam em verdadeiro (True) ou falso (False). São usadas em comandos condicionais e de repetição. Servem para analisar o estado de uma processamento e permitir escolher o próximo passo. A expressão é avaliada da esquerda para a direita. Se o resultado (verdadeiro ou falso) puder ser determinado sem avaliar o restante, este é retornado imediatamente.

```
>>> 1= =1
True
>>> 1= =2
False
>>> 1= =1 or 1= =2
True
>>> 1= =1 and 1= =2
False
>>> 1<2 and 2<3
True
>>> not 1<2
```

```
False
>>> not 1<2 or 2<3
True
>>> not (1<2 or 2<3)
False
>>> "alo" and 1
1
>>> "alo" or 1
'alo'
>>> 0 or 100
100
>>> False or 100
100
>>> "abc" or 1
'abc'
>>> 1 and 2
2
>>> 0 and 3
0
>> False and 3
False
>>> 1 and 2 or 3
2
>>> 0 or 2 and 3
3
>>> 1 and not 0
True
```

2.15.2 Operadores: in , is , is not

```
>>>a=5
>>>b=a
>>>a is b
True
>>>a=5
>>>b=6
>>>a is b
False
>>>a is not b
True
>>>a=[1,2,3]
>>>1 in a
True
>>>a=False
>>>b=True
>>>c= a and b #armazenar em c o resultado de uma expressão booleana
>>>c
False
```

Comparando objetos sequências de mesmo tipo:

```
>>>(1,2,3) < (1,2,4)
True

>>>'ABC'<'C'<'Python'
True
```

É permitido comparar objetos de diferentes tipos. O resultado é determinístico, porém, arbitrário: os tipos são ordenados pelos seus nomes. Então, uma lista é sempre menor do que uma string, uma string é sempre menor do que uma tupla, etc. Tipos numéricos mistos são comparados de acordo com seus valores numéricos, logo 0 é igual a 0.0, etc (esta comparação não é confiável podendo mudar com a versão)

2.16 Variáveis

São nomes dados a áreas de memória: Nomes podem ser compostos de algarismos, letras ou _. O primeiro caractere não pode ser um algarismo. Palavras reservadas (if, while, etc) são proibidas. As variáveis servem para guardar valores intermediários, construir estruturas de dados,etc... Uma variável é modificada usando o comando de atribuição:
var = expressão

É possível também atribuir a várias variáveis simultaneamente:
var1,var2,...,varN = expr1,expr2,...,exprN

Python é uma linguagem dinamicamente tipada, ou seja, suporta uma variável que pode ter diferentes tipos durante a execução do programa. Embora não seja explícita, ela assume um único tipo no momento em que se atribui um valor a ela.

```
>>> var = 3
>>> type(var)
<type 'int'>
>>> var = "3"
>>> type(var)
```

```
<type 'str'>
>>> var = 3.0
>>> type(var)
<type 'float'>
```

Para descobrir o tipo da variável, basta usar a função *type()*. Com linguagem dinâmica, você garante a simplicidade e flexibilidade da função. Em Python, tudo é objeto, que possui atributos e métodos e pode ser atribuído a uma variável ou passado como argumento de uma função. Por exemplo, uma variável que contém uma string é um objeto, pois qualquer string possui um método chamado upper que converte a string para maiúsculo.

```
>>> a = "alo"
>>> a.upper()
'ALO'
```

Os principais tipos de objetos em Python são inteiros, floats(reais), strings(texto), listas, tuplas, dicionários. Pode-se transformar o tipo original de um objeto para inteiro, float e string por meio de funções int, float e string.

2.17 Operador %

Esse operador é muito útil para formatação de texto.

Existem três tipos de formatação:
%s – substitui strings
%d – substitui inteiros
%f – substitui floats


```
>>> nome = "Eliane"
>>> print "Meu nome é %s" % nome
Meu nome é Eliane

>>> x="abacaxi"
>>> y="amarelo"
>>> print 'O %s é %s." %(x, y)
O abacaxi é amarelo.

>>> numInt = 19
>>> print "Eu tenho %d anos" % numInt
Eu tenho 19 anos

>>> numFloat = 1.6
>>> print "Altura: %.2f m" % numFloat
Altura: 1.60 m

>>> numFloat = 54.80
>>> print "Peso: %10.1f kg" % numFloat
Peso:       54.8 kg
```

“%.2f” corresponde a duas casas decimais e “%10.1”, a dez espaços, uma casa decimal.

3. Controle de Fluxo e Estrutura de Dados

3.1 Controle de Fluxo

Assim como nas outras linguagens, Python possui estruturas de controle de fluxo (condição e laço) também.

3.1.1 Instrução: while

Repete uma sequência de comandos enquanto uma dada expressão booleana é avaliada como verdadeira. Em Python, como em C, qualquer valor inteiro não nulo é considerado verdadeiro (*true*), zero tem valor falso (*false)*. A condição pode ser ainda uma lista ou string, na verdade qualquer sequência; qualquer coisa com comprimento não nulo tem valor *true* e sequências vazias tem valor *false*. Os operadores padrão para comparação são os mesmos de C: < (menor que), > (maior que), ==(igual), <= (menor ou igual), >= (maior ou igual) and != (diferente).

Formato: while *expressão***:** **# não se esqueça da indentação**
comando
...
comando

```
>>> a=10
>>> while a>8:
        a-=1
        print a
9
8
>>>
```

Exemplo série de Fibonacci:

```
>>> # Serie de Fibonacci :
... # A soma de dois elementos define o próximo

>>> a,b=0,1
>>> while b<1000:
        print b,
        a,b=b,a+b

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

O corpo do laço é indentado[12]. Python (ainda!) não possui facilidades automáticas de edição de linha. Na prática você irá preparar scripts Python complexos em um editor de texto; a maioria dos editores de texto possui facilidades de indentação automática. Quando comandos compostos forem alimentados ao interpretador interativamente, devem ser encerrados por uma linha em branco (já que o *parser* não tem como adivinhar qual é a última linha do comando). Observe que toda linha de um mesmo bloco de comandos deve possuir a mesma indentação.
Como em todo comando de repetição, é importante evitar os chamados "laços infinitos"

```
>>> a = 10
>>> while a>8:
        print a,
        a = a+1
```

12: Indentação em Python é a maneira de agrupar comandos.

```
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 ...
Ctrl-C aborta comando IDLE.
```

3.1.2 Instrução: for

A instrução **for** interage com uma sequência, ou seja, requer um objeto lista ou qualquer outro de sequência.

for *i* **in** *sequência***:**
    comando
    .....
    comando

```
>>> a = ['João', 'Rafael', 'Douglas']
>>> a
['Jo\xe3o', 'Rafael', 'Douglas']
>>> for i in a:
        print i
João
Rafael
Douglas

>>> # Medindo algumas strings:
>>> a = [’gato’, ’janela’, ’vivendo’]
>>> for x in a:
        print x, len(x)

gato 4
janela 6
vivendo 7
```

A construção for em Python difere um pouco, do que se está acostumado, do C ou Pascal. Ao invés de se iterar sobre progressões aritméticas (como em Pascal), ou fornecer ao usuário a habilidade de definir tanto o passo da iteração quanto a condição de parada (como em C), o **for** de Python itera sobre os itens de uma sequência (uma lista ou uma string) na ordem em que aparecem.

Função range([inicio,]fim[,passo]): Se você precisar iterar sobre sequências numéricas, a função interna range() é a resposta. Ela gera listas contendo progressões aritméticas.

```
>>> range(9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

O ponto de parada fornecido nunca é gerado na lista; range(9) gera uma lista com 9 valores, exatamente os índices válidos para uma sequência de comprimento 9. É possível iniciar o intervalo em outro número, ou alterar a razão da progressão (inclusive com passo negativo):

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

3.1.3 Cláusulas *break, continue* e *else*

O break, suspende o laço mais interno de um for ou while. O continue, continua o programa no próximo comando situado fora do laço mais interno. Laços podem ter uma cláusula else, que é executada sempre que o laço se encerra por exaustão da lista (no caso do for) ou quando a condição se torna falsa (no caso do while), mas nunca quando o laço é encerrado por um break.

```
>>> for n in range(2, 10):
        for x in range(2, n):
                if n % x == 0:
                        print n, 'igual', x, '*', n/x
                        break
        else:
                # encontrou numero primo
                print n, 'eh um numero primo'

>>>
2 eh um numero primo
3 eh um numero primo
4 igual 2 * 2
5 eh um numero primo
6 igual 2 * 3
7 eh um numero primo
8 igual 2 * 4
9 igual 3 * 3
```

3.1.4 Instrução if

Em Python, não é permitido fazer atribuições em um *if*, então é necessário usar o operador "= =" quando fizer

comparações. A palavra *elif* é a abreviação de “else if”.

```
>>> if x < 0:
        print "Negativo"
    elif x = = 0:
        print "Zero"
    elif x <= 10:
        print "Entre 1 e 10"
    elif x <= 20:
       print "Entre 11 e 20"
    elif x <= 30:
        print "Entre 21 e 30"
    else:
        print "Acima de 30"

Exemplos:
a = input("Entre com um numero:")
if a < 0:
        print a," é negativo"
elif a= =0:
        print a," é zero"
else:
        print a," é positivo"
print "Obrigado!"

Execução 1:

Entre com um numero:0
0 é zero
Obrigado!

Execução 2:
```

```
Entre com um numero:2
2 é positivo
Obrigado!
```

3.1.5 Construção pass

A construção pass não faz nada. Ela pode ser usada quando a sintaxe exige um comando mas a semântica do programa não requer nenhuma ação. Por exemplo:

```
>>> while True:
        pass            # Busy-wait (espera ocupada)- para com
                        # a interrupção de teclado
```

3.1.6 Técnicas de Laço

método iteritems() - para obter chave/valor ao percorrer um dicionário com laço.

```
>>>dicionario={1:'um', 2:'dois', 3:'tres'}
>>>for i, j in dicionario.iteritens():
                print i, j

1  um
2  dois
```

função enumerate() - obtem o índice e o valor correspondente em uma lista.

```
>>>lista=['um', 'dois', 'tres']
>>>for i,j  in enumerate(lista):
```

```
                print i, j

0  um
1  dois
2  tres
```

função zip() - percorrer duas sequências simultaneamente com laço e agrupá-los.

```
>>> questao = [’nome’, ’convite’, ’sapato favorito’]
>>> resposta = [’jose’, ’especial’, ’marrom’]
>>> for q, a in zip(questao, resposta):
                print ’Qual seu %s? Eh %s.’ % (q, a)
Qual seu nome? Eh sergio.
Qual seu convite? Eh especial.
Qual seu sapato favorito? Eh marrom.
```

função reversed() - percorrer uma lista em ordem reversa.

```
>>>lista=[1,2,3]
>>>for i in reversed(lista):
        print i
3
2
1
```

função sorted() - retorna lista na forma ordenada.

```
>>>lista=['a','b','e','c']
>>>for i in sorted(lista):
        print i,
```

```
a b c e
```

3.2 Estrutura de Dados

3.2.1 Usando Listas como Pilhas

O último elemento a entrar é o primeiro a sair.

```
>>> pilha = [3, 4, 5]
>>> pilha.append(6)
>>> pilha.append(7)
>>> pilha
[3, 4, 5, 6, 7]
>>> pilha.pop()
7
>>> pilha
[3, 4, 5, 6]
>>> pilha.pop()
6
>>> pilha.pop()
5
>>> pilha
[3, 4]
```

3.2.2 Usando Listas como Filas

O primeiro elemento que entra é o primeiro que sai.

```
>>> fila = ["Erica", "Jo", "Miguel"]
>>> fila.append("Tereza") # entra Teresa
>>> fila.append("Gilson") # entra Gilson
>>> fila.pop(0)
’Erica’
```

```
>>> fila.pop(0)
'Jo'
>>> fila
['Miguel', 'Teresa', 'Gilson']
```

3.2.3 Funções sobre listas: filter(), map(), e reduce()

'filter(*função*, *sequência*)' - retorna uma sequência consistindo dos itens pertencentes a sequência para os quais *função*(*item*) é verdadeiro. Se a sequência for string ou tuple, o resultado será sempre do mesmo tipo; caso contrário, será sempre uma lista. Por exemplo, para computar números primos:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> filter(f, range(2, 25))      #válido só nessa faixa de dados
[5, 7, 11, 13, 17, 19, 23]
```

'map(*função*, *sequência*)' - aplica função(item) para cada item da sequência e retorna a lista de valores retornados a cada aplicação. Por exemplo, para computar quadrados:

```
>>> def QUADRADO(x): return x*x
...
>>> map(QUADRADO, range(1, 10))
[1, 4, 9, 16, 25, 36, 49, 64, 81]

Exemplo soma de 2 sequências:

>>> seq = range(8)
>>> def soma(x, y): return x+y
```

```
...
>>> map(soma(seq, seq))
[0, 2, 4, 6, 8, 10, 12, 14]
```

'reduce(*função*, *sequência*)' - esta função pega dois primeiros itens da sequência e passa para a função e depois pega o retorno da função e o próximo item e passa para a função, fazendo isso até acabar a sequência. Por exemplo, para computar a soma dos 10: primeiros números inteiros:

```
>>> def soma(x,y): return x+y

>>> reduce(soma, range(1, 11))
55
```

Um terceiro argumento pode ser passado para indicar o valor inicial. Neste caso, redução de sequências vazias retornará o valor inicial. Se a sequência não for vazia, a redução se iniciará a partir do valor inicial.

3.2.4 Abrangência de Listas(*list comprehensions)*

Cada abrangência de lista consiste numa expressão seguida da cláusula *for*, e então *zero* ou mais cláusulas *for* ou *if*. O resultado será uma lista proveniente da avaliação da expressão no contexto das cláusulas *for* e *if* subsequentes. Se a expressão gerar uma tupla, a mesma deve ser inserida entre parênteses.

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

3.2.5 O comando del

Remove item da lista a partir de um índice. Permite também fatiar a lista (slice). Também apaga variáveis.

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a              # apagando variáveis
```

3.2.6 Trabalhando com conjuntos - sets

Python também inclui um tipo de dados chamado conjunto (*set*). Um conjunto é uma coleção desordenada de dados, sem elementos duplicados. Usos comuns para isso incluem verificações da existência de objetos em outras sequências e eliminação de itens duplicados. Conjuntos também suportam operações matemáticas como união, interseção, diferença e diferença simétrica.

```
>>> cestaFrutas = [’abacate’, ’laranja’, ’laranja’, ’pera’, ’banana’]
>>> frutas = set(cestaFrutas)    # cria conjunto sem duplicação
>>> frutas
set([’abacate’, ’laranja’, ’pera’, ’banana’])
>>> ’banana’ in frutas                # teste rápido
True
>>> ’jaboticaba’ in frutas
False
>>> # Demonstração de operações de conjuntos
...
>>> a = set(’abracadabra’)
>>> a
set([’a’, ’r’, ’b’, ’c’, ’d’])      # set deixa apenas uma letra de cada sem repetição

>>> b = set(’alacazam’)
>>> b
set([’a’, ’l','c','z','m'])
>>> a - b                      # letras em a e não em b
set([’r’, ’d’, ’b’])

>>> a | b # letras em a  e em  b
set([’a’, ’c’, ’r’, ’d’, ’b’, ’m’, ’z’, ’l’])
```

```
>>> a & b # letras em ambos a e b
set(['a', 'c'])

>>> a ^ b #letras em a ou b e não em ambos
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

4. Funções, Recursão, Exceção, Arquivos e Módulos

4.1 Funções

Funções são blocos de código com um nome; recebem um conjunto de parâmetros (ou argumentos) e retornam um valor. Python possui, como seria esperado de uma linguagem de programação completa, suporte a funções. Existem diversas funções pré-definidas pelo interpretador. A sintaxe geral para definir uma função é:

```
def nome_funcao(arg_1, arg_2, ..., arg_n):
    #
    # bloco de código contendo corpo da função
    #
    return valor_de_retorno # retornar é opcional
```

Os argumentos são valores fornecidos na chamada da função, e que ficam disponíveis por meio de variáveis locais no corpo da função. Ao encontrar o comando return, a função termina imediatamente e o controle do programa volta ao ponto onde a função foi chamada. Se uma função chega a seu fim sem nenhum valor de retorno ter sido especificado, o valor de retorno é None.

O corpo da função deve começar na linha seguinte e deve ser indentado. Opcionalmente, a primeira linha do corpo da função pode ser uma string, cujo propósito é documentar a função. Se presente, essa string chama-se ***docstring.***

```
>>> def f(): #docstring
        ''' funcao f
```

```
        do sergio'"
        return
>>> print f()
None

>>>f                              #endereço da função na memória
<function f at 0X00e20370>

>>>f.__doc__                      #documentação da função
'funcao f\n\tdo sergio'

>>> def f():
      return "Oi"
>>> print f()
Oi

>>> def f(nome):
        return "Oi, "+nome+"!"
>>> print f("Joao")
Oi, Joao!
```

Exemplo de programa com função:

```
def imprime_cardapio (pratos):
    print "Cardapio para hoje\n"
    for p in pratos:
        imprime_prato(p)
    print "\nTotal de pratos: %d" % len(pratos)
```

```
def imprime_prato(p):
    print "%s ..... %10.2f" % (p["nome"], p["preco"])

# defino dicionários descrevendo os pratos
p1 = {"nome": "Arroz com brocolis", "preco": 9.90}
p2 = {"nome": "Soja com legumes",   "preco": 7.80}
p3 = {"nome": "Lentilhas", "preco": 4.80}
lista_pratos = [p1, p2, p3]

# e chamo uma função, passando os pratos como argumento
imprime_cardapio(lista_pratos)

#-----------------fim programa---------------------------------------
```

4.1.1 Escopo das variáveis

A execução da função gera uma nova tabela de símbolos utilizada para as variáveis locais da função, mais precisamente, toda atribuição a variável dentro da função armazena o valor na tabela de símbolos local. Referências a variáveis são buscadas primeiramente na tabela local, então na tabela de símbolos global e finalmente na tabela de símbolos interna (*built-in*). Portanto, deve-se declarar que a variável é global antes de atribuir um valor para

ela, se você deseja acessar a variável externa, se fizer o contrário da erro. O uso de uma variável externa é possível, sem definir ela globalmente, só não é possível fazer atribuição a ela, pois a linguagem gera uma variável local.

```
>>>v = 0
>>>def processa():
        global v
          v = 1

>>>v=0
>>>def p():
         x=1
         v=3
         print x,v

>>>p()
1 3

>>v   # veja usamos a variável v internamente e a v externa não foi alterada
0
```

4.1.2 Funções como parâmetro

Nomes de funções podem ser manipulados como variáveis e mesmo como argumentos de funções. Para saber se um nome se refere a uma função, use o predicado callable().

```
>>> def f(g):
```

```
        return g(5)

>>> def h(x):
        return x*x

>>> f(h)
25
>>> m = h
>>> callable(m)
True
>>> f(m)
25
```

4.1.3 Função com 2 argumentos

```
>>> def c(x,y):
      return x+y

>>>c(1,2)
3
>>>c('a','b') #mudança do tipo de variável
durante a execução
'ab'
```

4.1.4 Função com valores padrão ou default

```
>>> def aplica_multa(valor, taxa=0.1):
      return valor + valor * taxa
>>>aplica_multa(10)
11.0
```

```
>>>aplica_multa(10,0.5)
15.0
```

Dica: Não utilize como valor padrão listas, dicionários e outros valores mutáveis; os valores padrões são avaliados apenas uma vez e o resultado obtido não é o que intuitivamente se esperaria.

4.1.5 Função com argumentos opcionais

```
>>> def somatoria(*argumentos):
        soma = 0
        for i in argumentos:
                soma+=i         # É igual a soma=soma+i
        return soma

>>> somatoria(1,2,3,4,5)
15
```

A função pode receber quantos os argumentos o usuário desejar. Se o argumento não for compatível retorna um erro.

```
>>> somatoria(1,2,3,4,'teste')

Traceback (most recent call last):
File "<pyshell#112>", line 1, in <module>
somatoria(1,2,3,4,'teste')
File "<pyshell#110>", line 4, in somatoria
soma+=i
TypeError: unsupported operand type(s) for +=: 'int' and 'str'

>>> def testa_primo(n):
```

```
        teste=1
        for i in range(2,n):
                if n % i == 0:
                        teste=teste+1
        if teste != 1:
                print 'Número não primo'
        else:
                print 'Número primo'

>>> testa_primo(28)
Número não primo
>>> testa_primo(7)
Número primo
```

4.1.6 Função com conjunto de argumentos nomeados

Estes parâmetros especiais devem necessariamente ser os últimos definidos na lista de parâmetros da função.

```
def equipe(diretor, produtor, **atores):
      for personagem in atores.keys():
         print "%s: %s" % (personagem, atores[personagem])
      print "-" * 20  #repete o sinal de – 20 vezes
      print "Diretor:  %s" % diretor
      print "Produtor: %s" % produtor

equipe(diretor="Barreto",
      produtor="Paulo Bondade",
     Frank="Tom Belo", Edmundo="Patricia Maldonado",
     Linda="Juliana Fonseca")
```

4.1.7 Função lambda

Com a palavra-chave lambda, funções curtas e anônimas podem ser criadas. Aqui está uma função que devolve a soma de seus dois argumentos:
'**lambda a, b: a+b**'. Funções Lambda podem ser utilizadas em qualquer lugar que exigiria uma função tradicional. Sintaticamente, funções Lambda estão restritas a uma única expressão. Semanticamente, ela são apenas açúcar sintático para a definição de funções normais. Assim como definições de funções aninhadas, funções lambda não podem referenciar variáveis de um escopo mais externo.

```
>>> def make_incrementor(n):
        return lambda x: x + n

>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.2 Funções recursivas

Funções recursivas são funções que chamam a si mesma, de forma que para resolver um problema maior utiliza a recursão para chegar as unidades básicas do problema em questão e então calcular o resultado final.

Fibonacci recursivo:

```
#coding: utf-8
# fib_recursivo.py
def fibonacci(indice):
   if indice == 1: return 0
   elif indice == 2: return 1
     else: return fibonacci(indice - 1) + fibonacci(indice - 2) #
chamada recursiva

for i in range(1,11):
   print fibonacci(i),
```

Busca Binária Recursiva:

```
def testa(lista,valor):
   def busca_binaria(imin,imax):
      if imin==imax: return imin
      else:
         meio=(imax+imin)/2
      if valor>lista[meio]:
         return busca_binaria(meio+1,imax)
      else:
         return busca_binaria(imin,meio)
   i = busca_binaria(0,len(lista)-1)
   if lista[i]==valor:
      print valor,"encontrado na posicao",i
   else:
      print valor,"nao encontrado"

testa([1,2,5,6,9,12],5)
```

```
>>>
5 encontrado na posicao 2
```

Quando uma função é chamada, um pouco de memória é usado para guardar o ponto de retorno, os argumentos e variáveis locais. Assim, soluções iterativas são normalmente mais eficientes do que soluções recursivas equivalentes. Isto não quer dizer que soluções iterativas sempre sejam preferíveis a soluções recursivas. Se o problema é recursivo por natureza, uma solução recursiva é mais clara, mais fácil de programar e, frequentemente, mais eficiente.

4.3 Funções pré-definidas

Python possui uma série de funções pré-definidas, que já estão disponíveis quando executamos o interpretador, sem ter que recorrer a bibliotecas externas. Algumas funções importantes que ainda foram ou não apresentadas no texto seguem:

- `range(a,b)`: recebe dois inteiros, retorna uma lista de inteiros entre `a` e `b`, não incluindo `b`. Esta função é frequentemente utilizada para iterar laços `for`.

```
>>> print range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `len(a)`: retorna o comprimento da variável `a`. Para listas, tuplas e dicionários, retorna o número de elementos; para strings, o número de caracteres.

```
>>> print len([1,2,3])
```

```
3
```

- `round(a, n)`: recebe um float e um número; retorna o float arredondado com este número de casas decimais.

```
>>> print round(1.2345,2)
1.23
```

- `pow(a, n)`: recebe dois inteiros; retorna o resultado da exponenciação de `a` à ordem `n`. É equivalente à sintaxe `a ** n`.

```
>>>pow(2,8)
256
```

- `chr(a)`: recebe um inteiro (entre 0 e 255) como parâmetro, retornando o caractere correspondente da tabela ASCII.

```
>>> print chr(97)
a
```

- `unichr(a)`: como `chr()`, recebe um inteiro (aqui variando entre 0 e 65535), retornando o caracter Unicode correspondente.

```
>>> print unichr(97)
a
```

- `ord(a)`: recebe um único caractere como parâmetro, retornando o seu código ASCII.

```
>>> print ord("a")
```

```
97
```

- `min(a, b)`: retorna o menor entre `a` e `b`, sendo aplicável a valores de qualquer tipo.

```
>>> print min(5,6)
5
```

- `max(a, b)`: retorna o maior entre `a` e `b`.

```
>>> print max(5,6)
6
```

- `abs(n)`: retorna o valor absoluto de um número.

```
>>> print abs(-1+1j)
1.41421356237
```

- `hex(n)` e `oct(n)`: retornam uma string contendo a representação em hexadecimal e octal, respectivamente, de um inteiro.

```
>>> print hex(256)
0X100
>>> print oct(256)
0400
```

Há também funções de conversão explícita de tipo; as mais frequentemente utilizadas incluem:

- `float(n)`: converte um inteiro em um float.

```
>>> print float(1)
```

```
1.0
```

- `int(n)`: converte um float em inteiro.

```
>>> print int(5.5)
5
```

- `str(n)`: converte qualquer tipo em uma string. Tipos sequências são convertidos de forma literal, peculiarmente.

```
>>> print str([1,2,3]), str({"a": 1})
[1, 2, 3] {'a': 1}
```

- `list(l)` e `tuple(l)`: convertem uma sequência em uma lista ou tupla, respectivamente.

```
>>> print list("ábaco")
['á', 'b', 'a', 'c', 'o']
```

4.4 Módulos

A medida que seus programas crescem, pode ser desejável dividi-los em vários arquivos para facilitar a manutenção. Talvez você até queira reutilizar uma função sem copiar sua definição a cada novo programa. Para permitir isto, Python possui uma maneira de depositar definições em um arquivo e posteriormente reutilizá-las em um script ou seção interativa do interpretador. Esse arquivo é denominado módulo. Definições de um módulo podem ser importadas por outros

módulos ou no módulo principal. Um módulo é um arquivo contendo definições e comandos Python. O nome do arquivo recebe o sufixo '.py' . Dentro de um módulo, seu nome (uma string) está disponível na variável global __name__. O comando **reload(módulo)** recarrega módulo.

O procedimento para criar um módulo é o mesmo para escrever um programa completo. Abra uma janela nova em IDLE(File>New Window), ou digite Crtl-N. Agora digite as funções e salve com a extensão .py, este programa deve estar no diretório que esteja no path. Exemplo:

Descobrindo diretório:

```
>>>import sys #
>>>sys.path
```

Mostra todos os diretórios do path: ['C:python25' , 'C:\\Python25\\Lib\\idlelib',.....]

Escreveremos o módulo funcoes.py e armazenaremos num arquivo de nome funcoes.py.

```
#funcoes.py

def perimetro(r):

    """esta funcao calcula perimetro de um
       circulo de raio r"""
    try:
       r=float(r)
       return 2*3.14*r

    except:
```

```
        print 'Erro argumento'

def area(r):
    r=float(r)
    """calcula area"""
    try:
        r=float(r)
        return 3.14*(r**2)
    except:
        print 'argumento e numero'
def nada():
    """faz nada so para mostrar
doc strings"""
    pass
```

No shell importaremos o módulo e veremos o funcionamento, quando o módulo é importado o Python gera o arquivo compilado Python (funcoes.pyc):

```
>>>import funcoes
>>>dir (funcoes)
mostra atributos e métodos

>>>print funcoes.__doc__
None  #não tem documentação

>>>print funcoes.nada.__doc__
faz nada so para mostrar
doc strings

>>>print funcoes.perimetro(4)
```

```
25.12
```

Na hora de execução devemos informar módulo.função(parâmetros).

Outra forma de usar funções é importá-las diretamente:

from módulo **import** funcao1, funcao2,.....

```
>>>form funcoes import area
>>>print area(4) #use só o nome da função
50.24
```

4.4.1 Módulos padrão

Python possui um biblioteca padrão de módulos, descrita em um documento em separado, a *Python Library Reference* (doravante "Library Reference"). Alguns módulos estão embutidos no interpretador; estes possibilitam acesso a operações que não são parte do núcleo da linguagem, seja por eficiência ou para permitir o acesso a chamadas de sistema. O conjunto presente destes módulos é configurável, por exemplo, o módulo amoeba só está disponível em sistemas que suportam as primitivas do Amoeba. Existe um módulo que requer especial atenção: sys, que é embutido em qualquer interpretador Python. As variáveis sys.ps1 e sys.ps2 definem as strings utilizadas como prompt primário e secundário:

```
>>> import sys
>>> sys.ps1
'>>> '
```

```
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Essas variáveis só estão definidas se o interpretador está em modo interativo.

A variável sys.path contém uma lista de strings que determina os caminhos de busca de módulos conhecidos pelo interpretador. Ela é inicializada para um caminho default determinado pela variável de ambiente PYTHONPATH ou por um valor default interno se a variável não estiver definida. Você pode modificá-la utilizando as operações típicas de lista, por exemplo:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

A função interna dir() é utilizada para se descobrir que nomes são definidos por um módulo. Ela retorna uma lista ordenada de strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__',
'__name__', '__stderr__',
```

```
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'callstats', 'copyright',
'displayhook', 'exc_clear', 'exc_info', 'exc_type',
'excepthook',
'exec_prefix', 'executable', 'exit', 'getdefaultencoding',
'getdlopenflags',
'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint',
'maxunicode',
'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache',
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval',
'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin',
'stdout',
'version', 'version_info', 'warnoptions']
```

Sem nenhum argumento, dir() lista os nomes atualmente definidos.

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib',
'fibo', 'sys']
```

Observe que ela lista nomes dos mais diversos tipos: variáveis, módulos, funções, etc.
dir() não lista nomes de funções ou variáveis internas. Se você desejar conhecê-los, eles estão definidos no módulo padrão __builtin__:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception',
'False',
'FloatingPointError', 'FutureWarning', 'IOError',
'ImportError',
'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'OverflowWarning',
'PendingDeprecationWarning', 'ReferenceError',
'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration',
'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError',
'UserWarning', 'ValueError', 'Warning', 'WindowsError',
'ZeroDivisionError', '_', '__debug__', '__doc__',
'__import__',
'__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod',
'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
```

’frozenset’, ’getattr’, ’globals’, ’hasattr’, ’hash’, ’help’, ’hex’, ’id’, ’input’, ’int’, ’intern’, ’isinstance’, ’issubclass’, ’iter’, ’len’, ’license’, ’list’, ’locals’, ’long’, ’map’, ’max’, ’min’, ’object’, ’oct’, ’open’, ’ord’, ’pow’, ’property’, ’quit’, ’range’, ’raw_input’, ’reduce’, ’reload’, ’repr’, ’reversed’, ’round’, ’set’, ’setattr’, ’slice’, ’sorted’, ’staticmethod’, ’str’, ’sum’, ’super’, ’tuple’, ’type’, ’unichr’, ’unicode’, ’vars’, ’xrange’, ’zip’]
[PYT]

4.4.2 Pacotes

Pacotes são uma maneira de estruturar espaços de nomes para módulos utilizando a sintaxe de “separação por ponto”. Como exemplo, o módulo A.B designa um submódulo chamado ‘B’ num pacote denominado ‘A’. O uso de pacotes permite que autores de grupos de módulos (como NumPy ou PIL) não se preocupem com a colisão entre os nomes de seus módulos e os nomes de módulos de outros autores. Suponha que você deseje projetar uma coleção de módulos (um “pacote”) para o gerenciamento uniforme de arquivos de som. Existem muitos formatos diferentes (normalmente identificados pela extensão do nome de arquivo, por exemplo. ‘.wav’, ‘.aiff’, ‘.au’), de forma que você pode precisar criar e manter uma crescente coleção de módulos de conversão entre formatos. Ainda podem existir muitas operações diferentes passíveis de aplicação sobre os arquivos de som (mixagem, eco, equalização, efeito estéreo artificial). Logo, possivelmente você também estará escrevendo uma interminável coleção de módulos para aplicar estas operações. Aqui está uma possível estrutura para o seu pacote (expressa em termos de um sistema hierárquico de arquivos):

```
Sound/                  Pacote de mais alto nível
      __init__.py       Inicializa pacote de som
      Formats/          Sub pacote para arquivos de
                        conversão de formato
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            ...
      Effects/          Sub pacote de efeito de sons
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
      Filters/          Sub pacote para filtros
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
          ...
```

Ao importar esse pacote o Python busca pelo subdiretório com mesmo nome nos diretórios listados em sys.path. Os arquivos '__init__.py' são necessários para que Python trate os diretórios como um conjunto de módulos. Isso foi feito para evitar diretórios com nomes comuns, como 'string', de inadvertidamente pode esconder módulos válidos que ocorram a posteriori no caminho de busca. No caso mais simples, '__init__.py' pode ser um arquivo vazio. Porém, ele pode conter código de inicialização para o pacote ou gerar a variável __all__, que será descrita depois. Usuários do pacote podem importar módulos individuais, por exemplo:

import Sound.Effects.echo

Assim se carrega um submódulo Sound.Effects.echo. Ele deve ser referenciado com seu nome completo, como em:
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)

Uma alternativa para a importação é:
from Sound.Effects import echo

Assim se carrega o módulo sem necessidade de prefixação na hora do uso. Logo, pode ser utilizado como se segue:
echo.echofilter(input, output, delay=0.7, atten=4)

Também é possível importar diretamente uma única variável ou função, como em:

from Sound.Effects.echo import echofilter

Novamente, há carga do submódulo echo, mas a função echofilter() está acessível diretamente sem prefixação:
echofilter(input, output, delay=0.7, atten=4)

Observe que ao utilizar from *package* import *item*, o item pode ser um sub pacote, submódulo, classe, função ou variável. O comando import primeiro testa se o item está definido no pacote, senão assume que é um módulo e tenta carregá-lo. Se falhar em encontrar o módulo uma exceção ImportError é levantada. Em oposição, na construção import *item.subitem.subsubitem*, cada item, com exceção do último, deve ser um pacote. O último pode ser também um pacote ou módulo, mas nunca uma entidade contida em um módulo.

4.4.3 Importando * de Um Pacote

Agora, o que acontece quando um usuário escreve: from Sound.Effects import * ? Idealmente, poderia se esperar que todos submódulos presentes no pacote fossem importados. Infelizmente, essa operação não funciona muito bem nas plataformas Mac ou Windows, onde não existe distinção entre maiúsculas ou minúsculas nos sistema de arquivos. Nestas plataformas não há como saber como importar o arquivo 'ECHO.PY', deveria ser com o nome echo, Echo ou ECHO. A restrição de nomes de arquivo em DOS com o formato 8+3 adiciona um outro problema na hora de se utilizar arquivos com nomes longos. A única solução é o autor do pacote fornecer um índice explícito do pacote. O comando de importação utiliza a seguinte convenção: se o arquivo '__init__.py' do pacote define a lista chamada __all__, então esta lista indica os nomes dos módulos a serem importados quando o comando from *package* import * é encontrado. Fica a cargo do autor do pacote manter esta lista atualizada, inclusive fica a seu critério excluir inteiramente o suporte a importação direta de todo o pacote através do from *package* import *. Por exemplo, o arquivo 'Sounds/Effects/__init__.py' poderia conter apenas:__all__ = ["echo", "surround", "reverse"]
Isso significaria que from Sound.Effects import * iria importar apenas os três sub-módulos especificados no pacote Sound. Se __all__ não estiver definido, o comando from Sound.Effects import * não importará todos os submódulos do pacote Sound.Effects no espaço de nomes corrente. Há apenas garantia que o pacote Sound.Effects foi importado (possivelmente executando qualquer código de inicialização em '__init__.py') juntamente com os nomes definidos no pacote. Isso inclui todo

nome definido em '__init__.py' bem como em qualquer submódulo importado a partir deste. Considere o código abaixo:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Neste exemplo, os módulos *echo* e *surround* são importados no espaço de nomes corrente, pois estão definidos no pacote Sound.Effects. O que também funciona quando __all__ estiver definida. Em geral, a prática de importar * de um dado módulo é desaconselhada, principalmente por prejudicar a legibilidade do código. Contudo, é recomendada em sessões interativas para evitar excesso de digitação. Lembre-se que não há nada de errado em utilizar from Package import specific_submodule! De fato, essa é a notação recomendada a menos que o módulo que efetua a importação precise utilizar submódulos homônimos em diferentes pacotes. (Tutorial Python)

4.4.4 Referências em Um Mesmo Pacote

Os submódulos frequentemente precisam referenciar uns aos outros. Por exemplo, o módulo surround talvez precise utilizar o módulo echo. De fato, tais referências são tão comuns que o comando import primeiro busca módulos dentro do pacote antes de utilizar o caminho de busca padrão. Portanto, o módulo surround pode usar simplesmente import echo ou from echo import echofilter. Se o módulo importado não for encontrado no pacote corrente (o pacote do qual o módulo corrente é submódulo), então o comando import procura por um módulo de mesmo nome no escopo global. Quando pacotes são estruturados em sub pacotes (como no exemplo Sound), não

existe atalho para referenciar submódulos de pacotes irmãos, então o nome completo do pacote deve ser utilizado. Por exemplo, se o módulo Sound.Filters.vocoder precisa utilizar o módulo echo no pacote Sound.Effects, é preciso importá-lo como from Sound.Effects import echo. (Tutorial Python).

4.4.5 Pacotes em múltiplos diretórios

Pacotes suportam mais um atributo especial, __path__. Este é inicializado como uma lista contendo o nome do diretório com o arquivo '__init__.py' do pacote, antes do código naquele arquivo ser executado. Esta variável pode ser modificada; isso afeta a busca futura de módulos e sub pacotes contidos no pacote. Apesar de não ser muito usado, pode ser usado para estender o conjunto de módulos usado num pacote. (Tutorial Python)

4.4.6 Módulos Importantes

Há um grande conjunto de módulos que se instalam juntamente com o interpretador Python; são descritos nesta seção alguns dos mais interessantes.

- `sys`: oferece várias operações referentes ao próprio interpretador. Inclui: `path`, uma lista dos diretórios de busca de módulos do python, `argv`, a lista de parâmetros passados na linha de comando e `exit()`, uma função que termina o programa.
- `time`: oferece funções para manipular valores de tempo. Inclui: `time()`, uma função que retorna o *timestamp* (data e hora atual); `sleep(n)`, que pausa a execução

por n segundos; e `strftime(n)`, que formata a data e hora atual em uma string de acordo com um formato fornecido.

- `os`: oferece funções relacionadas ao ambiente de execução do sistema. Inclui: `mkdir()`, que cria diretórios; `rename()`, que altera nomes e caminhos de arquivos; e `system`, que executa comandos do sistema.
- `os.path`: oferece funções de manipulação do caminho independente de plataforma. Inclui: `isdir(p)`, que testa se `d` é um diretório; `exists(p)`, que testa se `p` existe; `join(p,m)`, que retorna uma string com os dois caminhos `p` e `m` concatenados.
- `string`: oferece funções de manipulação de string (que também estão disponíveis como métodos da string). Inclui: `split(c, s, p)`, que divide a string `c` em até `p` partições separadas pelo símbolo `s`, retornando-as em uma lista; `lower(c)`, que retorna a string `c` convertida em minúsculas; e `strip(c)`, que retorna `c` removendo espaços e quebras de linha do seu início e fim.
- `math`: funções matemáticas gerais. Inclui funções como `cos(x)`, que retorna o cosseno de `x`; `hypot(x, y)`; que retorna a distância euclidiana entre `x` e `y`; e `exp(x)`; que retorna o exponencial de `x`.
- `random`: geração de números randômicos. Inclui: `random()`, que retorna um número randômico entre 0 e 1; `randrange(m,n)`, que retorna um randômico entre `m` e `n`; `choice(s)`, que retorna um elemento randômico de uma sequência `s`.

- `getopt`: processamento de argumentos de comando de linha; ou seja, os parâmetros que passamos para o interpretador na linha de execução. Inclui: `getopt()`, que retorna duas listas, uma com argumentos e outra com opções da linha de comando.
- `Tkinter`: um módulo que permite a criação de programas com interface gráfica, incluindo janelas, botões e campos texto.

`A documentação do Python inclui uma descrição detalhada (e muito boa) de cada um destes módulos e de seus membros.`

4.5 Funções de entrada e saída

Entrada e saída são operações de comunicação de um programa com o mundo externo. Essa comunicação se dá usualmente através de arquivos. Arquivos estão associados a dispositivos. Por exemplo, disco, impressora, teclado. Em Python, um arquivo pode ser lido/escrito através de um objeto da classe file.

Sempre que um comando print é executado, o resultado vai para um arquivo chamado sys.stdout. Sempre que lemos um dado através do comando input ou raw_input, na verdade estamos lendo de um arquivo chamado sys.stdin. Mensagens de erro ou de rastreamento de exceções são enviadas para um arquivo chamado sys.stderr.

```
>>> import sys
>>> sys.stdout.write("alo")
alo
>>> print "alo"
```

```
alo
>>> sys.stdin.readline()
sfadfas
'sfadfas\n'
>>> raw_input()
fasdfadsf
'fasdfadsf'
```

Os arquivos sys.stdin, sys.stdout e sys.stderr normalmente estão associados ao teclado e ao display do terminal que está sendo usado, mas podem ser associados a outros dispositivos.

Em Unix/Linux e Windows:

programa > arquivo
Executa programa redirecionando stdout para arquivo

programa < arquivo
Executa programa redirecionando stdin de arquivo

programa1 | programa2
Executa programa1 e programa2 sendo que a saída de programa1 é redirecionada para a entrada de programa2

raw_input() - função que retorna sempre uma string:

```
>>> x = raw_input('Informe a fase: ')
Informe a fase: vapor
>>> x
'vapor'
```

Retornando um inteiro ou float:

```
>>> r = float(raw_input('Raio: '))
Raio: 4.5
>>> type(r)
<type 'float'>
>>> a = int(raw_input('Tamanho do lado do quadrado: ')
Tamanho do lado do quadrado: 23
>>> type(a)
<type 'int'>
```

input() - função que retorna número:

```
>>>x=input('No.')
No. 12
>>>x
12
```

print - comando que imprime dados, imprimindo dados com print:

```
>>> b = 1
>>> while b < 4:
        if b==1:
                print '%i dólar vale %.2f reais' %(b,b*2.98)
                                #formatacao %.2f (2 casas dec.)
        else:
                print '%i dólares valem %.2f reais' %(b,b*2.98)
        b+=1
```

```
1 dólar vale 2.98 reais
2 dólares valem 5.96 reais
3 dólares valem 8.94 reais
>>> a = ['João', 'Rafael', 'Douglas']
>>> a
['Jo\xe3o', 'Rafael', 'Douglas']
>>> for i in a:
        print i

João
Rafael
Douglas
>>> for x in a:
        print '%s tem %i letras' %(x,len(x))

João tem 4 letras
Rafael tem 6 letras
Douglas tem 7 letras
>>> a=[1,2,3]
>>> for i in range(len(a)):
        print i

1
2
3

>>> a=[1,2,3]
>>> for i in range(len(a)):
        print i,

1 2 3
```

Outros exemplos e funções:

```
>>> s = ’Oi, Mundo.’
>>> str(s)
’Oi, Mundo.'

>>> repr(s)              # coloca “”
"’Oi, Mundo.’"

>>> str(0.1)              # numero → string
’0.1’

>>> repr(0.1)
’0.10000000000000001’  # string com limite máximo da
representação

>>>for i in range(10):
        print repr(i).rjust(1), repr(i*2).rjust(5)  #atua em cima da
string

0     0
1     1
2     4
…....

>>>for i in range(5):                          #executa e veja o
que acontece
        print str(i).ljust(1),str(i*2),ljust(2)
        print str(i).center(1),str(i*2).center(2)
```

```
….
….
>>>'12'.zfill(5)   #string com zero a esquerda
'00015'

>>> tabela = {'Sergio': 4127, 'Jose': 4098, 'Dino': 7678}
>>> for nome, fone in tabela.items():
        print '%-10s ==> %10d' % (nome, fone)


Dino       ==>       7678
Jose       ==>       4098
Sergio     ==>       4127
>>>
```

4.5.1 Operações com arquivos

Em Python, assim como em outras linguagens, é possível manipular arquivos. Desse modo, o usuário consegue abrir, ler, alterar, gravar e fechar um arquivo. Tudo o que você precisa para manipular, corretamente, um arquivo é conhecer os métodos responsáveis por cada coisa. A função *open(nome_arquivo,modo,buffer)* retorna um objeto arquivo, em outras palavras, abre o arquivo que você especificar.

```
>>> arq = open('c:\teste.txt')
```

Você, ainda, tem a opção de passar como parâmetro o modo como vai abrir o arquivo. Caso passe um “r”, seu arquivo será somente de leitura; se passar “a”, poderá adicionar

conteúdo ao final do arquivo; caso passe “w”, o arquivo será apagado e aberto para escrita, no entanto, se ele não existir será automaticamente criado, etc. A opção ’r+’ abre o arquivo tanto para leitura como para escrita. Caso o usuário não passe o segundo parâmetro, Python assume o “r”.No Windows e no Macintosh, ’b’ adicionado a string de modo indica que o arquivo será aberto no formato binário. Sendo assim, existem os modos compostos : ’rb’, ’wb’, e ’r+b’. O Windows faz distinção entre arquivos texto e binários: os caracteres terminadores de linha em arquivos texto são levemente alterados em leituras e escritas. Essa mudança “por trás do pano” é útil em arquivos texto ASCII, mas irá corromper um arquivo binário como no caso de arquivos ‘JPEG’ ou ‘EXE’. Aopção Buffer indica se vai usar uma memória de buffer para entrada e saída(0-sem buffer / 1-buffer (default) / 2 ou mais - número de bytes do buffer).

```
>>> t = open('C:\meuarquivo.txt','w',0)
>>> print t
<open file 'C:\meuarquivo.txt', mode 'w' at 0x00A85A40>
```

4.5.2 Métodos do objeto arquivo

t.read(size): Para ler o conteúdo de um arquivo chame t.read(*tamanho*), que lê um punhado de dados retornando-os como string. O argumento numérico *tamanho* é opcional. Quando *tamanho* for omitido ou negativo, todo o conteúdo do arquivo será lido e retornado.
Nota: t é o objeto gerado quando executamos o comando open.

```
>>> t=open('c:\meu.txt','r',0)
```

```
>>> t.read()
'Esse é todo o conteúdo do arquivo.\n'
>>>
>>> t.read()  #final do arquivo atingido
''
```

t.readline(): lê uma única linha do arquivo. O caractere de retorno de linha (\n) é deixado ao final da string, só sendo omitido na última linha do arquivo se ele já não estiver presente lá. Isso elimina a ambiguidade no valor de retorno. Se f.readline() retornar uma string vazia, então o arquivo acabou. Linhas em branco são representadas por '\n', uma string contendo unicamente o terminador de linha.

```
>>> t.readline()
'Essa é a primeira linha do arquivo.\n'
>>> t.readline()
'Segunda linha do arquivo\n'
>>> t.readline()
''
```

t.readlines(): retorna uma lista contendo todas as linhas do arquivo. Se for fornecido o parâmetro opcional *tamanho*, será lida a quantidade especificada de bytes e mais o suficiente para completar uma linha.

```
>>> for linha in t:        # le diretamente as linhas do arquivo
        print linha,

Esta é a primeira linha do arquivo.
Segunda linha do arquivo.
```

t.write(*string*): escreve o conteúdo da *string* para o arquivo, retornando None.

```
>>> t.write('Isso é um teste.\n')
```

Ao escrever algo que não seja uma string, é necessário convertê-lo antes:

```
>>> valores = ('a resposta', 42)
>>> s = str(valores)
>>> t.write(s)
```

t.writelines(sequência**):** Escreve a lista (ou qualquer sequência) de strings, uma após a outra no arquivo.

```
>>> x=open("teste","w+")
>>> lista="1","2"
>>> x.writelines(lista)
>>> x.close()
```

t.tell(): retorna um inteiro que indica a posição corrente de leitura ou escrita no arquivo, medida em bytes desde o início do arquivo. Para mudar a posição utilize 'f.seek(*deslocamento*, *posição*)'. A nova posição é computada pela soma do *deslocamento* a um ponto de referência, que por sua vez é definido pelo argumento *posição*. O argumento *posição* pode assumir o valor 0 para indicar o início do arquivo, 1 para indicar a posição corrente e 2 para indicar o fim do arquivo. Este parâmetro pode ser omitido, quando é assumido o valor default 0 (valor default – valor assumido quando não tem o parâmetro).

```
t = open('testearquivo', 'r+')          # o arquivo deve existir
t.write('0123456789abcdef')
t.seek(5) # Vai para o sexto byte
print  t.read(1)
t.seek(-3, 2)                # Vai para o terceiro byte antes do fim
print t.read(1)
print t.tell()
t.close()                                #fecha arquivo

5
d
14
```

t.close(): fecha arquivo.

4.5.3 Interação com sistema operacional

Operações de entrada e saída são na verdade realizadas pelo sistema operacional. O módulo "**os"** possui diversas variáveis e funções que ajudam um programa Python a se adequar ao sistema operacional, por exemplo:

import os - importa módulo "os".

os.getcwd() - retorna o diretório corrente.

os.chdir(*dir*) - muda o diretório corrente para *dir*.

os.sep - é uma string com o caractere que separa componentes de um caminho ('/' para *Unix*, '\\' para *Windows*).

os.path.exists(“*caminho*”) - informa se existe o caminho.

os.system(“string”) - executa uma string como se fosse uma linha de comando.

```
a=os.system
a('format c:')   # cuidado formata o disco
```

os.startfile('arquivo') - executa um arquivo se ele for do tipo .bat ou .exe.

```
>>>os.startfile('c.bat')
```

Exemplo de um programa instalador, que cria um diretório na linha 6, copia o programa para o diretório e abre um arquivo de texto com o bloco de notas.

```
#instalar.py
import os
comando=os.system
comando('copy a:\meumodulo.py c:\Python27')
NOME=raw_input('Seu nome: ') # Linha 4
NOME='Python-'+NOME
comando('md c:\"+NOME) # Linha 6
comando('copy a:\meuprograma.py c:\"+NOME)
abrir=os.startfile
abrir('a:/leiame.txt')
```

4.6 Função vars()

O dicionário obtido com a função vars() pode ser usado para ter acesso a todas as variáveis definidas num escopo.

```
>>> vars()
{'__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__', '__doc__': None}
>>> def f():
        x = 1
        print vars()

>>> vars()
{'f': <function f at 0xb6e7f56c>, '__builtins__':
<module '__builtin__' (built-in)>, '__name__':
'__main__', '__doc__': None}
>>> f()
{'x': 1}
```

4.7 Erros e exceções

São os dois tipos de erro mais comuns. Um erro de sintaxe ocorre quando você não respeita a sintaxe, digitando errado ou esquecendo de algo, como ":".

```
>>> if x < 10
SyntaxError: invalid syntax
```

Nesse caso, está faltando ":" no final da condição.

4.7.1 Como ler uma mensagem de erro

A dura realidade é que um programador profissional passa boa parte de sua vida caçando erros, e por isso é fundamental saber extrair o máximo de informações das mensagens resultantes. A essa altura você talvez já tenha

provocado um erro para ver o que acontece. Vamos fazer isso agora, e aprender a ler as mensagens resultantes. Pode parecer perda de tempo, mas é importantíssimo saber interpretar as mensagens de erro porque a melhor forma de aprender a programar é experimentando, e ao experimentar você certamente vai provocar muitos erros. Como exemplo, vamos digitar um programa com uma expressão aritmética sem sentido.

```
c=1
print c
a=7+2
print 7/0

Traceback (most recent call last):
  File "C:/Python27/teste.py", line 4, in <module>
    print 7/0
ZeroDivisionError: integer division or modulo by zero
>>>
```

A primeira linha da mensagem de erro informa o arquivo onde ocorreu o erro:
File "C:/Python27/teste.py", line 4, in <module>.

Quando você está usando o interpretador em modo interativo essa informação é inútil, mas ao trabalhar com programas extensos ela se torna indispensável. As próximas duas linhas indicam onde ocorreu o erro e qual é o erro:
print 7/0
ZeroDivisionError: integer division or modulo by zero

No Python Shell(IDLE), tente agora este comando e veja o que ocorre.

```
>>> 7+/2
SyntaxError: invalid syntax
>>>
```

Note que o símbolo / foi sinalizado, indicando que está neste ponto o erro. Foi a partir desse ponto que a expressão deixou de fazer sentido para o interpretador.

4.7.2 Tratamento de Erros

Quando exceções ocorrem, o programa para e o interpretador gera uma mensagem de erro. No entanto, é possível resolver esse problema. Para este fim, basta usar duas instruções: *try* e *except*. Enquanto a instrução *try* contém comandos que serão executados, a instrução *except* executa o tratamento nas possíveis exceções desses comandos, ou seja, na instrução *except* você diz o que é para ser feito caso um erro ocorra durante a execução dos comandos.

```
>>> n = int(raw_input("Escolha uma chave: "))
Escolha uma chave: 6
>>> dic = {1:'um',2:'dois',3:'tres'}
>>> try:
        print dic[n]
except:
        print "Chave não existe"

Chave não existe
```

Se não houver exceções nos comandos, a instrução *except* é ignorada. É possível numa única instrução *try* criar

diversas *except* com diferentes tipos de exceções, entre eles: ***ValueError*, *IndexError*, *TypeError*.**

Existem, ainda, duas outras instruções: *raise* e *finally*. No primeiro caso, força a ocorrência de um determinado tipo de exceção. Quando existir a instrução *finally*, seus comandos serão executados independentemente de existir ou não exceções, e serão executados antes das exceções. A instrução *try* nunca permite que o usuário use *except* e *finally*, simultaneamente. Além de existir diversos tipos pré-definidos de exceções no Python, o usuário tem a liberdade de criar seus próprios tipos de exceções através de classes.

```
>>> while True:
        try:
                x = int(raw_input("Entre com um número: "))
                break
        except ValueError:
                print "Opa! Esse número não é válido. Tente de
novo..."
```

A construção **try . . . except** possui uma cláusula **else** opcional, que quando presente, deve ser colocada depois de todas as outras cláusulas. É útil para um código que precisa ser executado se nenhuma exceção foi levantada.

```
for arg in sys.argv[1:]:
try:
        f = open(arg, 'r')
except IOError:
        print 'Nao consegui abrir:', arg
else:
```

```
        print arg, ' tem ', len(f.readlines()), 'linhas.'
f.close()
```

finally: é executada sempre, ocorrendo ou não uma exceção. Quando ocorre a exceção, é como se a exceção fosse sempre levantada após a execução do código na cláusula *finally*. Mesmo que haja um break ou return dentro do try, ainda assim o finally será executado. O código na cláusula *finally* é útil para liberar recursos externos (como arquivos e conexões de rede), não importando se o uso deles foi bem sucedido. A construção try pode ser seguida da cláusula finally ou de um conjunto de cláusulas except, mas nunca ambas (não ficaria muito claro qual deve ser executada primeiro).

```
>>> try:
        raise KeyboardInterrupt
    finally:
        print 'Ate logo, mundo!'

Ate logo, mundo!
Traceback (most recent call last):
  File "<pyshell#20>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

Diversos tipos de exceções vêm pré-definidas pelo interpretador Python; o guia de referência do Python contém uma lista completa e os casos onde são levantadas. Note também que esta introdução a exceções não cobre a sua sintaxe completa; consulte a seção Errors and Exceptions do tutorial Python para maiores detalhes.

5. Biblioteca Padrão

5.1 Interface com Sistema Operacional

O módulo ***os*** oferece interfaceamento com o sistema operacional e possui várias funções. Para carregá-lo digite: **import os**, veja os comandos **dir(os) e help(os).**

os.getcwd() - retorna o diretório corrente.

```
>>>os.getcwd()
'C:\\Python25'
```

os.chdir(<diretório>) - muda o diretório corrente.

```
>>>os.chdir('/')
>>>os.getcwd()
'C:\\'
```

Use import os ao invés de from os import *, que pode ocultar funções como os.open()
que funcionam diferente.

rename(<velho>,<novoarquivo>) - renomeia arquivo.

Veja documentação através do help(os).

5.2 Módulo shutil

Utilitário com funções para copiar arquivos e árvore de diretórios.

```
>>> import shutil     # importa módulo
>>> shutil.copyfile('data.db', 'archive.db')   # copia arquivo
>>> shutil.move('/build/executables', 'installdir') # move
arquivo ou diretório para outro local
```

5.3 Módulo glob

Retorna uma lista de arquivos de diretório, usando caracteres coringa.

```
>>>glob.glob (*.py) # retorna todos os arquivos de extensão
.py do diretório.

['teste.py']
```

5.4 Módulo sys

Provê acesso a alguns objetos usados ou mantidos pelo interpretador que interage diretamente com o interpretador.

```
>>>sys.path – lista os diretórios do path.

['C:/Python27', 'C:\\Python27\\Lib\\idlelib',
'C:\\WINDOWS\\system32\\python27.zip',
'C:\\Python27\\DLLs', 'C:\\Python27\\lib',
'C:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk',
'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
>>>
```

Scripts geralmente precisam processar argumentos passados na linha de comando. Esses argumentos são armazenados como uma lista no atributo *argv* do módulo sys.

Por exemplo, teríamos a seguinte saída executando **'python demo.py um dois tres'** na linha de comando:

```
>>> import sys
>>> print sys.argv
['demo.py', 'um', 'dois', 'tres']

sys.exit() - encerra um script.
```

5.5 Módulo math

Oferece acesso às funções da biblioteca C para matemática e ponto flutuante.

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

5.6 Módulo random

Fornece ferramentas para gerar seleções aleatórias.

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10) # exemplo sem
substituicao
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
```

```
>>> random.random() # randomico float
0.17970987693706186
>>> random.randrange(6) # escolhendo randomico de uma
faixa de 6
4
```

5.7 Módulo datetime

Fornece classes para manipulação de datas e horas nas mais variadas formas.

```
Calendários são facilmente formatados e construídos

>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the
%d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of
December.'

Suporte a operação aritméticas entre datas

>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

5.8 Módulo zlib

Compressão de dados**.**

```
>>>import zlib
>>> x='O rato roeu a roupa rasgada do rei de Roma'
>>>len(x)
42
>>>t=zlib.compress(x)
>>>len(t)
42
>>>a=zlib.decompress(t)
>>>print t
O rato roeu a roupa rasgada do rei de Roma
```

5.9 Módulo array

Fornece objeto **array()**, para trabalhar com listas.

Somando uma lista com 4 elementos numéricos todos com tamanho de 2 bytes em vez de 16 bytes da representação normal.

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

5.10 Módulo collection

Objeto **deque** para anexar e remover rapidamente elementos em listas.

```
>>> from collections import deque
>>> d=deque(["tar1","tar2","tar3"])
>>> d.append("tar4")
>>> print "Manipulando",d.popleft()
Manipulando tar1
>>>
```

5.11 Módulo decimal

Para trabalhar com números de ponto flutuante em operações financeiras.

```
>>> from decimal import *
>>> Decimal('0.77') * Decimal('1.06')
Decimal("0.8162")
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
>>> getcontext().prec=54
>>> Decimal(1)/Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857')
```

5.12 Módulo itime

Para medições de tempo.

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.152927469554127385
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.099465943134362078
```

5.13 Módulo doctest

Para executar testes automáticos com o docstring. Retorna erro se o resultado não for o esperado.

```
# gere este arquivo teste.py
def media(values):
        """Calcula a media aritmetica de uma lista
        >>> print media([20, 30, 70])
        50.0
        """
        return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()
```

No shell aparecerá mensagem de erro:

```
**********************************************************
File "__main__", line 3, in __main__.media
Failed example:
    print media([20, 30, 70])
Expected:
    50.0
Got:
    40.0
**********************************************************
1 items had failures:
   1 of   1 in __main__.media
***Test Failed*** 1 failures.
```

Substitua o valor 50 por 40 e a função não retorna nenhum resultado.

6. Orientação a Objeto

6.1 Introdução a Programação Orientada a Objetos(POO)

Este termo foi criado por Alan Curtis Kay, autor da linguagem de programação Smalltalk. Antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar estas ideias foi a linguagem Simula 67, criada por Ole Johan Dahl e Kristen Nygaard em 1967. Este paradigma de programação já é bastante antigo e hoje vem sendo aceito nas grandes empresas de desenvolvimento de Software. Exemplos de linguagens modernas que utilizam POO: Java, C#, C++, Object Pascal (Delphi), Ruby, Python, Lisp, ...

6.2 O que POO?

Com o objetivo de aproximar o mundo real do mundo virtual, a programação orientada a objeto tenta simular o mundo real dentro do computador. Para isso, nada mais natural do que utilizar objetos, afinal, nosso mundo é composto de objetos, certo? Com isto, o programador é responsável por moldar o mundo dos objetos, e explicar para estes objetos como eles devem interagir entre si. Os objetos "conversam" uns com os outros através do **envio de mensagens**, e o papel principal do programador é especificar quais serão as mensagens que cada objeto pode receber, e também qual a ação que aquele objeto deve realizar ao receber aquela mensagem específica. Uma **mensagem** é um pequeno texto que os objetos conseguem entender e, por questões técnicas, não pode conter espaços. Junto com algumas dessas mensagens ainda é possível passar algumas informações para o objeto (parâmetros), dessa forma,

dois objetos conseguem trocar informações entre si facilmente. Por exemplo num sistema de controle de clientes podemos afirmar que um dos objetos são os clientes.

6.3 Definições básicas de orientação a objeto

O objeto é um tipo abstrato que contém dados mais os procedimentos que manipulam esses dados, ou é uma unidade de software que encapsula algoritmos e os dados sobre o qual os algoritmos atuam.

Mensagens: É uma informação enviada ao objeto para que ele se comporte de uma determinada maneira. Um programa orientado a objetos em execução consiste em envios, interpretações e respostas às mensagens.

Métodos: São procedimentos ou funções residentes nos objetos que determinam como eles irão atuar ao receber as mensagens.

Variáveis de Instância: São variáveis que armazenam informações ou dados do próprio objeto. Podem também ser chamadas de propriedades do objeto.

Classe: Estrutura fundamental para definir novos objetos. Uma classe é definida em código-fonte, e possui nome, um conjunto de atributos e métodos.

Subclasse: Uma nova classe originada de uma classe maior (classe pai).
Instância: São os objetos de uma classe. Cada objeto utilizado em uma aplicação pertencente a uma classe, e é uma instância

dessa classe(diz-se que o objeto fabricado por uma classe é uma *instância* da classe).

Hereditariedade: É um mecanismo que permite o compartilhamento de métodos e dados entre classes, subclasses e objetos. A hereditariedade permite a criação de novas classes programando somente as diferenças entre a nova classe e a classe pai.

Encapsulamento ou abstração: É um mecanismo que permite o acesso aos dados do objeto somente através dos métodos desse objeto. Nenhuma outra parte do programa pode operar sobre os dados de nenhum objeto. A comunicação entre os objetos é feita apenas através de mensagens.

Polimorfismo: Uma mesma mensagem pode provocar respostas diferentes quando recebidas por objetos diferentes. Com o polimorfismo, pode-se enviar uma mensagem genérica e deixar a implementação a cargo do objeto receptor da mensagem.

Persistência: É a permanência de um objeto na memória. Quando um objeto é necessário, ele é criado na memória (métodos construtores). Quando ele não for mais necessário, é destruído da memória (métodos destrutores). Quando um objeto é destruído, seu espaço na memória é liberado automaticamente. Este processo recebe o nome de "coleta de lixo" (*garbage collector*).

6.4 POO em Python

Em Python todos os tipos de dados são objetos. Com o objetivo de entender classes, é preciso falar das regras de escopo

em Python. O que há de importante para saber é que não existe nenhuma relação entre nomes em espaços distintos. Por exemplo, dois módulos podem definir uma função de nome "maximize" sem confusão – usuários dos módulos devem prefixar a função com o nome do módulo para evitar colisão. É normal utilizar a palavra *atributo* para qualquer nome depois de um ponto. Na expressão z.real, por exemplo, real é um atributo do objeto z. Estritamente falando, referências para nomes em módulos são atributos: na expressão *modulo.atrib*, *modulo* é um objeto módulo e *atrib* é seu atributo. Neste caso, existe um mapeamento direto entre os os atributos de um módulo e os nomes globais definidos no módulo: eles compartilham o mesmo espaço de nomes. Atributos de módulos são passíveis de atribuição, você pode escrever 'modulo.resposta = 42', e remoção pelo comando del ('del modulo.resposta').

Espaços de nomes são criados em momentos diferentes e possuem diferentes longevidades. O espaço de nomes que contém os nomes pré-definidos é criado quando o interpretador inicializa e nunca é removido. O espaço de nomes global é criado quando uma definição de módulo é lida, e normalmente duram até a saída do interpretador. Os comandos executados pela invocação do interpretador, ou pela leitura de um script, ou interativamente são parte do módulo chamado __main__, e portanto possuem seu próprio espaço de nomes (os nomes pré-definidos possuem seu próprio espaço de nomes no módulo chamado __builtin__). O espaço de nomes local para uma função é criado quando a função é chamada, e removido quando a função retorna ou levanta uma exceção que não é tratada na própria função. Naturalmente, chamadas recursivas de uma função possuem seus próprios espaços de nomes. Um escopo é uma região textual de um programa Python onde um espaço de nomes é diretamente acessível. Onde "diretamente acessível"

significa que uma referência sem qualificador especial permite o acesso ao nome. Ainda que escopos sejam determinados estaticamente, eles são usados dinamicamente. A qualquer momento durante a execução, existem no mínimo três escopos diretamente acessíveis: o escopo interno (que é procurado primeiro) contendo nomes locais, o escopo intermediário (com os nomes globais do módulo) e o escopo externo (procurado por último) contendo os nomes pré-definidos. Se um nome é declarado como global, então todas as referências e atribuições de valores vão diretamente para o escopo intermediário que contém os nomes do módulo. Caso contrário, todas as variáveis encontradas fora do escopo interno são apenas para leitura (a tentativa de atribuir valores a essas variáveis irá simplesmente criar uma variável local, no escopo interno, não alterando nada na variável de nome idêntico fora dele). Normalmente, o escopo local referencia os nomes locais da função corrente. Fora de funções, o escopo local referencia os nomes do escopo global (espaço de nomes do módulo). Definições de classes adicionam um espaço de nomes ao escopo local.

Um detalhe especial é que atribuições são sempre vinculadas ao escopo interno. Atribuições não copiam dados, simplesmente vinculam objetos a nomes. O mesmo é verdade para remoções. O comando 'del x' remove o vínculo de x do espaço de nomes referenciado pelo escopo local. De fato, todas operações que introduzem novos nomes usam o escopo local. Em particular, comandos import e definições de função vinculam o módulo ou a função ao escopo local (a palavra-reservada **global** pode ser usada para indicar que certas variáveis residem no escopo global ao invés do local).

Observação: Objetos possuem um atributo de leitura escondido chamado __dict__ que retorna o dicionário utilizado para implementar o espaço de nomes do módulo. O nome __dict__ é

um atributo, porém não é um nome global. **(Tutorial Python Guido van Rossum)**

6.5 Definindo Classes

A maneira mais simples de definir uma classe é dada abaixo.

```
class nome:
        var = valor
        ...
        var = valor
        def metodo (self, ... arg):
        ...
        def metodo (self, ... arg):
        ...
```

As variáveis e os métodos são escritos precedidos pelo nome da classe e por um ponto (.) Assim, uma **variável *v* definida numa classe *c* é escrita *c.v* .** Os métodos sempre têm self como primeiro argumento, self se refere a uma instância da classe. Para criar uma instância utiliza-se a expressão: **nome_da_classe()**. Os objetos de classe suportam dois tipos de operações: referências a atributos e instanciação. Atributos válidos são todos os nomes presentes no espaço de nomes da classe quando o objeto classe foi criado. Portanto, se a definição da classe era:

```
class MinhaClasse:
        "Um exemplo simples de classe"
        i = 12345
```

```
        def f(self):
                return 'Ola mundo'
```

Então **MinhaClasse.i** e **MinhaClasse.f** são referências a atributos válidos, retornando um inteiro e um objeto função, respectivamente. Atributos de classe podem receber atribuições, logo você pode mudar o valor de MinhaClasse.i por atribuição. __doc__ é também um atributo válido, que retorna a docstring pertencente a classe: "Um exemplo simples de classe".

Instanciação de Classe: x = MinhaClasse() # cria um objeto vazio
A operação de instanciação cria um objeto vazio. Muitas classes preferem criar um novo objeto em um estado inicial pré-determinado. Para tanto, existe um método especial __init__.

```
def __init__(self):
        self.data = []
```

O método __init__() pode ter argumentos para aumentar sua flexibilidade. Neste caso, os argumentos passados para a instanciação de classe serão delegados para o método __init__().

```
>>> class Complex:
                def __init__(self, realpart, imagpart):
                self.r = realpart
                self.i = imagpart

x=complex()   # veja o método __init__ em ação, obrigando a fornecer os argumentos                                    para
instanciar o objeto com um estado inicial pré-determinado.
```

```
Traceback (most recent call last):
   File "<pyshell#9>", line1, in<module>
   x=complex()
TypeError:__init__() takes exactly 3 arguments (1 given)

>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Acessando atributos de dados e métodos nas instâncias:

```
>>>x=MinhaClasse()
>>>x.i=1
>>>while x.i<10:
        x.i=x.i*2
        print x.i

2
4
8
16
>>>while x.i<10:        #alterou atributo de dados sem deixar rastro
        x.i=x.i*2
        print x.i
        del x.i
>>>x.i
16
>>> del x.i             #apaga x.i e retorna valor inicial
>>>x.i
```

12345

Um método é uma função que "pertence a" uma instância (em Python, o termo método não é aplicado exclusivamente a instâncias de classes definidas pelo usuário: outros tipos de objetos também podem ter métodos; por exemplo, objetos listas possuem os métodos: append, insert, remove,sort, etc). Nomes de métodos válidos de uma instância dependem de sua classe. Por definição, todos atributos de uma classe que são funções equivalem a um método presente nas instâncias. No nosso exemplo, x.f é uma referência de método válida já que MinhaClasse.f é uma função, enquanto x.i não é, já que MinhaClasse.i não é função. Entretanto x.f não é o mesmo que MinhaClasse.f, o primeiro é um método de objeto e o segundo é um objeto função. O método em um objeto é chamado da forma **x.f()** (objeto x), porque um método é uma função, neste caso o parâmetro deste objeto foi MinhaClasse.f(x), com isto podemos dizer que um método com n argumentos o **1o. Argumento sempre será a instância da classe**.

Como nomear atributos de dados e métodos: Algumas ideias incluem colocar nomes de métodos com inicial maiúscula, prefixar atributos de dados com uma string única (quem sabe "_"), ou simplesmente utilizar substantivos para atributos e verbos para métodos. Isto evita erros difíceis de encontrar. O **argumento self** é só uma convenção usada por desenvolvedores Python, mas pode-se usar qualquer nome.

Atribuir um objeto função a uma variável local da classe é válido.

```
# Função definida fora da classe
def f1(self, x, y):
        return min(x, x+y)

class C:
        f = f1
        def g(self):
                return 'hello world'
        h =g
```

f,g,h agora são atributos de C. Mas esta solução não é uma boa prática.

Chamada de Métodos por outros métodos utilizando o argumento self :

```
class Somas:
        def __init__(self):
                self.dado = []
        def soma(self, x):
                self.dado.append(x)
        def somadupla(self, x):
                self.soma(x)
                self.soma(x)
```

6.6 Herança

Um mecanismo fundamental em sistemas orientados a objetos modernos é herança: uma maneira de derivar classes novas a partir da definição de classes existentes, denominadas neste contexto classes base. As classes derivadas possuem acesso transparente aos atributos e métodos das classes base, e podem redefinir estes conforme conveniente.

Herança é uma forma simples de promover reuso através de uma generalização: desenvolve-se uma classe base com funcionalidade genérica, aplicável em diversas situações, e definem-se subclasses concretas, que atendam a situações específicas.

Classes Python suportam herança simples e herança múltipla:

```
        class nome-classe(base1, base2, ..., basen): # (mod.base1...base definida em módulo)
    var-1 = valor-1
    .
    .
    var-n = valor-n

    def nome-método-1(self, arg1, arg2, ..., argn):
       # bloco de código do método
    .
    .
    def nome-método-n(self, arg1, arg2, ..., argn):
       # bloco de código do método
```

Como pode ser observado acima, classes base são especificadas entre parênteses após o nome da classe sendo definida. Na sua forma mais simples.

```
>>> class B:
        n = 2
        def f(self,x): return B.n*x

>>> class C(B):     # chamada
        def f(self,x): return B.f(self,x)**2
```

```
        def g(self,x): return self.f(x)+1

>>> b = B()
>>> c = C()
>>> b.f(3)
6
>>> c.f(3)
36
>>> c.g(3)
37
>>> B.n = 5
>>> c.f(3)
225
```

Os atributos especiais __doc__ , __module__ e função help :

```
class Tapecaria:
   """Reforma e servico de
   qualidade em veiculos."""
   def revisao(self):
      """
      Este metodo responde com a avaliacao dos servicos
      """
print 'Um servico excepcional'
print Tapecaria.__doc__          #documentacao da classe
print Tapecaria().__doc__        #documentacao da instancia
print Tapecaria().revisao.__doc__  #documentacao da funcao
da instancia

resultado:
```

```
Um servico excepcional
Reforma e servico de
        qualidade em veiculos

                Este metodo responde com a avaliacao dos
servicos
```

comando: help(Tapecaria)

Help on class Tapecaria in module __main__

```
class Tapecaria
 |  Reforma e servcio de
 |  qualidade em veiculos.
 |
 |  Methods defined here:
 |
 |  revisao(self)
 |      Este metodo responde com a avaliacao dos servicos
```

Ferramenta de introspecção a ser comentada adiante. A primeira linha mostra o módulo a que pertence a classe e todos os métodos e as docstrings.

```
>>> Tapecaria.__module__        #mostra o módulo da classe
'__main__'
```

6.7 Atributos Privados

Em princípio, todos os atributos de um objeto podem sera acessados tanto dentro de métodos da classe como de fora.

Quando um determinado atributo deve ser acessado apenas para implementação da classe, ele não deveria ser acessível de fora. Em princípio tais atributos não fazem parte da interface "pública" da classe. Atributos assim são ditos *privados*. Em Python, atributos privados têm nomes iniciados por dois caracteres "sublinhado", isto é, __ .

```
>>> class C:
        def __init__(self,x): self.__x = x
        def incr(self): self.__x += 1
        def x(self): return self.__x
>>> a = C(5)
>>> a.x()
5
>>> a.incr()
>>> a.x()
6
>>> a.__x
Traceback (most recent call last):
File "<pyshell#13>", line 1, in -toplevela.__
x
AttributeError: C instance has no attribute '__x'
```

6.8 Outros Métodos

Já vimos um método desses: o construtor __init__ . Alguns outros são:

Adição: __add__
    Chamado usando '+'
Subtração: __sub__
    Chamado usando '-'

Representação: __repr__

Chamado quando objeto é impresso

Conversão para string: __str__

Chamado quando o objeto é argumento do construtor da classe str . Se não especificado, a função __repr__ é usada

Multiplicação:__mul__

Chamado usando '*'

```
>>> class vetor:
        def __init__(self,x,y):
                self.x, self.y = x,y
        def __add__(self,v):
                return vetor(self.x+v.x, self.y+v.y)
        def __sub__(self,v):
                return vetor(self.x-v.x, self.y-v.y)
        def __repr__(self):
                return "vetor("+str(self.x)+","+str(self.y)+")"
>>> a=vetor(1,2)
>>> a += vetor(3,5)
>>> a-vetor(2,2)
vetor(2,5)
>>> print a
vetor(4,7)


>>> class Op_Basicas:
        def __init__(self, entrada):
                self.valor = entrada
        def __add__(self, other): # método especial!
                return self.valor + other.valor
        def __mul__(self, other): # método espeial!
                return self.valor * other.valor
```

```
>>> a = Op_Basicas(56)
>>> b = Op_Basicas(-23)
>>> a + b
33
>>> a * b
-1288

>>> str1 = Op_Basicas('56')
>>> str2 = Op_basicas('-23')
>>> str1 + str2
'56-23'
>>> str1 * str2 # vai dar erro porque não definimos operação soma multipilicação de strings
ERRO
```

Note a sobrecarga de operadores, que antes só eram usados para inteiros, float...e passa agora a ser utilizado em instâncias.

```
>>>str2=5
>>>str1*str2
'5656565656' #operação de string por inteiro já é assumida pelo operador *.
```

Objetos Callable: são todos os objetos que são capazes de carregar um número indeterminado de argumentos. Funções, métodos e classes são sempre callable(chamáveis). Para sabermos se um objeto é callable usamos a operação boolena **callable(obj) .**

```
>>> callable(str1) # str1 é uma instância da classe Op_Basicas...
False
>>> callable(Op_Basicas)
True
>>> callable(d.quadrado)
True
>>> callable('abacaxi')
False
```

Obs: instãncias não são objetos callable.

```
>>> class Op_Basicas:
        def __init__(self, entrada):
                self.valor = entrada
        def __add__(self, other):
                return self.valor + other.valor
        def __mul__(self, other):
                return self.valor * other.valor
        def __call__(self, qualquer_coisa): # método especial!
                return qualquer_coisa

>>> a=Op_Basicas(56)
>>> callable(a)  # com o método __call__ a instância passa a ser callable
True
```

6.9 Funções Úteis

Há duas funções particularmente úteis

para estudar uma hierarquia de classes e instâncias:

- **isinstance(objeto, classe):** retorna verdadeiro se o objeto for uma instância da classe especificada, ou de alguma de suas subclasses.
- **issubclass(classe_a, classe_b):** retorna verdadeiro se a classe especificada como classe_a for uma subclasse da classe_b, ou se for a própria classe_b.

```
>>>isinstance(a,Op_Basicas)
True
```

6.10 Introspecção e reflexão

São propriedades de sistemas orientados a objetos que qualificam a existência de mecanismos para descobrir e alterar, em tempo de execução, informações estruturais sobre um programa e objetos existentes neste. Python possui tanto características introspectivas quanto reflexivas. Permite obter em tempo de execução informações a respeito do tipo dos objetos, incluindo informações sobre a hierarquia de classes. Preserva, também, **metadados** que descrevem a estrutura do programa sendo executado, e permitindo que se estude como ele está organizado sem a necessidade de ler o seu código-fonte.

Algumas funções e atributos são

particularmente importantes neste sentido, e são apresentadas nesta seção:

- `dir(obj)`: esta função pré-definida lista todos os nomes de variáveis definidos em um determinado objeto; foi apresentada anteriormente como uma forma de obter as variáveis definidas em um módulo, e aqui pode ser descrita em sua glória completa: descreve o conteúdo de qualquer objeto Python, incluindo classes e instâncias.
- `obj.__class__`: este atributo da instância armazena o seu objeto classe correspondente.
- `obj.__dict__`: este atributo de instâncias e classes oferece acesso ao seu estado local.
- `obj.__module__`: este atributo de instâncias e classes armazena uma string com o nome do módulo do qual foi importado.
- `classe.__bases__`: esta atributo da classe armazena em uma tupla as classes das quais herda.
- `classe.__name__`: este atributo da classe armazena uma string com o nome da classe.

7. Exemplos de Orientação a Objetos e Outros

7.1 Definição de uma classe

• uma classe com três métodos

```
class Contador(object):
        def __init__(self):
                self.dic = {}
        def incluir(self, item):
                qtd = self.dic.get(item, 0) + 1
                self.dic[item] = qtd
        def contar(self, item):
                return self.dic[item]
```

- __init__ é invocado automaticamente na instanciação
- Note que a referência ao objeto onde estamos operando é explícita (self).

```
>>>a=Contador()
>>>a
<__main__.Contador object at 0x00E1AEB0>
>>>a.incluir('a')
>>> print a.contar('a')
1
>>>a.incluir('b')
>>>print a.contar('b')
1
>>>a.incluir('b')
>>>print a.contar('b')
2
```

- Nas chamadas de métodos, o self é implícito pois a sintaxe é instancia.metodo().
- Todos os métodos de instância recebem como primeiro argumento uma referência explícita à própria instância.
- Por convenção, usa-se o nome self.
- É obrigatório fazer referência explícita a self para acessar atributos da instância.
- Usamos __init__ para inicializar os atributos da instância.

7.2 Instâncias abertas e classes "vazias"

- instâncias podem receber atributos dinamicamente

```
>>> class Animal(object):
        'um animal qualquer'
>>> baleia = Animal()
>>> baleia.nome = 'Moby Dick'
>>> print baleia.nome
```

7.3 Atributos de classe / de instância

- instâncias adquirem atributos da classe

```
>>> class Animal(object):
        nome = 'mailo' # atributo da classe

>>> cao = Animal()
>>> cao.nome # atributo adquirido da classe
'mailo'
>>> cao.nome = 'dino' # criado na instância
>>> cao.nome
```

```
'dino'
>>> Animal.nome # na classe, nada mudou
'mailo'
```

7.4 Métodos de classe / estáticos

- indicados por meio de decorators

```
class Exemplo(object):
        @classmethod
        def da_classe(cls, arg1):
                return (cls, arg1)
        @staticmethod
        def estatico(arg1):
                return arg1

>>> Exemplo.da_classe('fu')
(<class '__main__.Exemplo'>, 'fu')
>>> Exemplo.estatico('bar')
'bar'
```

7.5 Herança

```
class Contador(object):
        def __init__(self):
                self.dic = {}
        def incluir(self, item):
                qtd = self.dic.get(item, 0) + 1
                self.dic[item] = qtd
```

```
        def contar(self, item):
                return self.dic[item]
```

- **Herança**

```
class ContadorTolerante(Contador):
        def contar(self, item):
                return self.dic.get(item, 0)
```

- **A forma mais simples**

```
class ContadorTotalizador(Contador):
        def __init__(self):
                Contador.__init__(self)
                self.total = 0
        def incluir(self, item):
                Contador.incluir(self, item)
                self.total += 1
```

- **A forma mais correta**

```
class ContadorTotalizador(Contador): # super(...) acesso a classe pai

        def __init__(self):
                super(ContadorTotalizador,self).__init__()
                self.total = 0
        def incluir(self, item):
                super(ContadorTotalizador,self).incluir(item)
                self.total += 1
```

- **herança múltipla**

```
class ContadorTT(ContadorTotalizador, ContadorTolerante):
        pass
```

• MRO = ordem de resolução de métodos:

```
>>> ContadorTT.__mro__
(<class '__main__.ContadorTT'>,
<class '__main__.ContadorTotalizador'>,
<class '__main__.ContadorTolerante'>,
<class '__main__.Contador'>,
<type 'object'>)
```

Exemplo de uso

```
>>>class ContadorTT(ContadorTotalizador,
ContadorTolerante):
        pass

>>>ct=ContadorTT()
>>>for x in 'abobrinha':
        ct.incluir(x)

>>>ct.total
9
>>>ct.contar('a')
2
>>>ct.contar('x')
0
```

7.6 Encapsulamento

```
>>>class C():
        pass

>>>a=C()
>>>a.x=10   #pode alterar atributo
>>>print a.x
10
>>>a.x=11
>>>print a.x
11
```

• apenas para leitura, via decorator:

```
class C(object):
        def __init__(self, x):
                self.__x = x
        @property
        def x(self):
                return self.__x
```

• a notação __x protege este atributo contra acessos acidentais

```
>>>a=C(10)
>>>print a.x
10
>>>a.x=11
Error can´t set attribute
```

• para leitura e escrita:

```
class C(object):
        def __init__(self, x=0):
                self.__x = x
        def getx(self):
                return self.__x
        def setx(self, x):
                if x < 0: x = 0
                self.__x = x
        x = property(getx, setx)
```

```
>>>a=C(10)
>>>a.x=10
>>>a.y=11
>>>print a.x
11
>>>a.x=10
>>>print a.x
10
>>>b=C()
>>>print a.x
10
>>>print b.x
0
```

7.7 Passagem de parâmetros

```
def f(a, b=1, c=None):
        return a, b, c
>>> f()
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: f() takes at least 1 argument (0 given)
>>> f(9)
(9, 1, None)
>>> f(5,6,7)
(5, 6, 7)
>>> f(3, c=4)
(3, 1, 4)
```

```
def f(*args, **xargs):
        return args, xargs
```

```
>>> f()
((), {})
>>> f(1)
((1,), {})
>>> f(cor='azul')
((), {'cor': 'azul'})
>>> f(10,20,30,sabor='uva',cor='vinho')
((10, 20, 30), {'cor': 'vinho', 'sabor': 'uva'})
```

7.8 Polimorfismo

• polimorfismo com fatiamento e len

```
>>> l = [1, 2, 3]
>>> l[:2]
```

```
[1, 2]
>>> ’casa’[:2]
’ca’
>>> len(l), len(’casa’)
(3, 4)
```

• “tipagem pato” (duck typing) é polimorfismo dinâmico.

```
>>> def dobro(x):
... return x * 2
...
>>> dobro(10)
20
>>> dobro([1, 2, 3])
[1, 2, 3, 1, 2, 3]
>>> dobro(’casa’)
’casacasa'
```

• Baralho Polifórmico: classe simples

```
class Carta(object):
        def __init__(self, valor, naipe):
                self.valor = valor
                self.naipe = naipe
        def __repr__(self):
                return ’<%s de %s>’ % (self.valor, self.naipe)
```

• métodos especiais: __len__, __getitem__
• com esses métodos o Baralho implementa o protocolo das sequências

```
class Baralho(object):
        naipes = 'copas ouros espadas paus'.split()
        valores = 'A 2 3 4 5 6 7 8 9 10 J Q K'.split()
        def __init__(self):
                self.cartas = [
                Carta(v, n) for n in self.naipes for v in
self.valores
                ]
        def __len__(self):
                return len(self.cartas)
        def __getitem__(self, pos):
                return self.cartas[pos]
        def __setitem__(self, pos, item):
                self.cartas[pos] = item


>>>a=Baralho()
>>>a.cartas
>>>len(a)
52

>>> from random import shuffle
>>> shuffle(a)
>>>a[-3:]
[<K de espadas>, <10 de copas>, <K de copas>]
```

8. Tkinter - Módulo de Interface Gráfica

8.1 Introdução

O Tkinter é um conjunto de elementos (widgets), na linguagem Python, para a construção de interface gráfica com o usuário(GUI). Esse módulo gráfico vem junto com o Python. Vários elementos estão disponíveis neste módulo como botão, caixa de texto, janela etc.., além disso temos vários manipuladores de eventos (event handlers)como abrir e fechar janela e eventos de loop[13], na maioria das vezes é necessário atribuir ao elemento um evento[14], mas existem casos em que assim que adicionamos o elemento ele já tem associado um evento e seu manipulador(janela: já vem com botão de abrir e fechar).

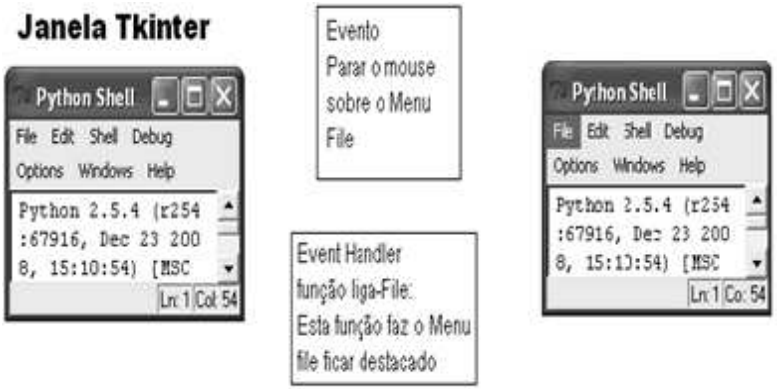


Figura 2: Eventos e Manipuladores

13 Evento que fica em espera aguardando até que um manipulador seja acionado, ou que uma entrada de dados seja feita

14 Rotina que executa alguma tarefa como: armazenar dados, fechar janela

Veja este caso acima: o mouse ficou parado em cima do menu File(Event) e o manipulador de evento(Event Handler) automaticamente modificou este botão destacando-o. Veja que para este caso não foi necessário fazer um ligação ou atribuição(binding), uma vez que este tipo de binding já é automático.

Como todo bom manual nosso primeiro programa é o "Alô Mundo!".

```
# -*- coding: cp1252 -*-     # caracteres acentuados
# File: alo1.py

from Tkinter import *      # importa o modulo Tkinter

raiz = Tk()                # cria uma janela de mais alto  nível
                           # chamada de raiz (root)

w = Label(raiz, text="Alô Mundo!")      # insere widgets na
                                        #  janela
w.pack()                                # método que faz o
                                        # texto caber na janela

raiz.mainloop()      # mostra a janela raiz  e executa o event

                     #loop, aguarda um evento ou fechamento da

                     # janela através de um evento destruidor
```

Execute este programa no prompt do sistema através do comando: python alomundo.py. Ao executar um programa em IDLE este poderá entrar em conflito com o mainloop da IDLE

uma vez que esta interface GUI é uma aplicação Tkinter.

Fig 3: Programa Alô Mundo na janela do Tkinter

8.2 A classe Tk

8.2.1 Item Básico

Para trabalharmos com a classe tk devemos primeiro disponibilizá-la através do comando import, depois importar a classe tk que cria a janela principal com os widgets minimizar, maximizar e fechar a janela, e depois visualizá-la através do event loop mainloop.

8.2.2 Um programa bem simples

```
>>>from Tkinter import *
>>>raiz=Tk()
>>>raiz.mainloop()
```

Fig 4: Tela do programa simples.

Este é um programa muito simples, inclusive os comandos podem ser digitados no IDLE.

8.2.3 Nosso primeiro objeto GUI

Sempre que trabalhamos com o Tkinter devemos importar todo conteúdo do módulo, quando fazemos isto importamos a classe Tk que tem todos o elementos(widgets). Nossa interface deve ser estruturada em forma de classes a partir da classe raiz, uma instância da classe Tk. Para programarmos com interfaces GUI Tkinter devemos citar três conceitos:

- A criação de um objeto GUI e a associação a raiz.
- O método pack.
- O conceito containers[15] versus widgets

É importante dividirmos os elementos ou componentes gráficos(widgets) em **containers e widgets.** Os elementos(widgets) serão os componentes gráficos visíveis(geralmente) da GUI e tem uma função voltada ao usuário, já os containers são componentes que agregam vários widgets, ou seja um recipiente onde colocamos os widgets. O Tkinter oferece vários tipos de containers. "Canvas" é um container para aplicações de desenho, e o container mais frequentemente utilizado é o "Frame"[16]. Frames são oferecidos pelo Tkinter como uma classe chamada "Frame".

Vejamos isto:

```
from Tkinter import *
root = Tk()
```

15 São componentes que agregam vários elementos(widgets).
16 É uma região retangular da tela.

```
container1 = Frame(root)
container1.pack()
root.mainloop()
```

No programa acima importamos o módulo Tk criamos um objeto raiz instância de Tk e dentro deste objeto colocamos um objeto container do tipo Frame, este tipo de estrutura gera uma relação de pai e filho ou mestre-escravo de tal forma que ao fecharmos a instância pai ou mestre automaticamente fecharemos a janela. O objeto container1 é uma instância de Frame e esta conectado logicamente ao objeto raiz, uma instância de Tk. Na próxima linha temos um método simplesmente designado, "pack" é um método que transforma em visuais as relações entre os componentes GUI e seus mestres. Se você não definir o componente pack ou outro gerenciador de geometria, nunca verá a GUI. Um gerenciador de geometria é essencialmente um API[17] – um meio de dizer ao Tkinter como você quer que containers e widgets se apresentem visualmente. Tkinter oferece três gerenciadores para esta finalidade: pack, grid e place. Pack e grid são os mais usados por serem mais simples.

* Substitua o comando container1.pack() por: container1.grid() e depois por container1.place() e veja o que acontece.

Comparando dois exemplos: No programa anterior, por não termos colocado nada dentro dela, a raiz mostrou a si mesma na

17 Application Programming Interface

tela com seu tamanho padrão, mas neste programa, nós preenchemos sua cavidade com o container1. Agora, a raiz se estica para acomodar o tamanho de container1, mas como não colocamos nenhum widget neste frame, nem especificamos um tamanho mínimo para ele, a cavidade da root se encolhe até o limite, podemos dizer que esta frame é elástica. Por isso não há nada para ser visto abaixo da barra de título desse programa.

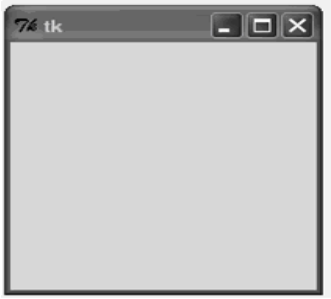


Fig. 5: Exemplo Anterior

Fig. 6: Exemplo

8.2.4 Containers e widgets

A estrutura de containers pode ser bastante sofisticada e pode conter um dentro do outro e dentro destes os widgets.

Figura 7: Containers com widgets(botões)

Veja a estrutura acima onde temos a janela raiz dentro dela os objetos containers frame 1 e frame 2 e dentro destas janelas os widgets. Temos agora que usar um gerenciador de geometria para posicionar widgets dentro das janelas.

Para exemplificar criaremos uma janela raiz com dois container e dentro deles os botões como acima.

```
# -*- coding: cp1252 -*- #para colocar caracteres acentuados
from Tkinter import *
root = Tk()
Container1 = Frame(root)
Container2= Frame(root)
Container1.pack()
Container2.pack()
botao = Button(Container1) #cria botão
botao["text"]= "ALO MUNDO1!" #texto do botão
botao["background"] = "gray" #cor do botão
botao.pack() #visualiza  e posiciona botão
```

```
botao = Button(Container2) # botão 2
botao["text"]= "ALO MUNDO2!"
botao["background"] = "gray"
botao.pack()
botao = Button(Container2) # botão 3
botao["text"]= "ALO MUNDO2!"
botao["background"] = "gray"
botao.pack()
root.mainloop()
```

Experimente mudar a cor do botão de “gray” para “blue”.

8.2.5 Usando classes para estruturar o programa

Ao escrever grandes programas é uma boa ideia dividir os códigos em classes. Esta estruturação em classes é muito importante e vai ajudá-lo a controlar melhor seu desenvolvimento. Um programa estruturado em classes é provavelmente – especialmente se seu programa for muito grande – muito mais fácil de ser entendido. Uma consideração muito importante é que estruturar sua aplicação como uma classe ajudará você a evitar o uso de variáveis globais. Eventualmente, conforme seu programa for crescendo, você provavelmente irá querer que alguns de seus event handler consigam compartilhar informações entre si. Uma maneira é usar variáveis globais, mas não é uma técnica muito recomendada. Um caminho muito melhor é usar instâncias (isto é objetos gerados pela chamada da classe), e para isso você precisa estruturar sua aplicação como classes.
Vamos estruturar em classe parte do exemplo anterior, primeiro criamos uma classe chamada Aplic e depois transcreveremos alguns dos códigos do programas anteriores para dentro do seu

método construtor (__init__). Nesta versão então teremos uma classe chamada Aplic e nela definiremos a forma da interface GUI. Todo esse código é inserido dentro do método construtor dessa classe(class Aplic). Quando o programa é executado, uma instância da classe é criada através do comando **aplic=Aplic(raiz)**, veja que a instância tem a letra inicial do nome minúsculo e a classe maiúscula (convenção) e perceba que passamos para esta classe como argumento a "raiz".

```
# -*- coding: cp1252 -*-
from Tkinter import *
class Aplic:
        def __init__(self, pai):
                self.Container1 = Frame(pai)
                self.Container1.pack()
                self.botao1= Button(self.Container1)
                self.botao1["text"]= "Aplicação"
                self.botao1["background"] = "red"
                self.botao1.pack()
raiz = Tk()
aplic = Aplic(raiz)
raiz.mainloop()
```

Vamos inserir agora mais um elemento(widget) nesta classe, para ficar assim a aplicação:

Fig. 8: Dois elementos(widget) na classe Aplic.

```
# -*- coding: cp1252 -*-
from Tkinter import *
class Aplic:
        def __init__(self, pai):
                self.Container1 = Frame(pai)
                self.Container1.pack()
                self.label1=Label(self.Container1)
                self.label1["text"]="Aperte o botão"
                self.label1.pack()
                self.botao1= Button(self.Container1)
                self.botao1["text"]= "Aplicação"
                self.botao1["background"] = "red"
                self.botao1.pack()
raiz = Tk()
aplic = Aplic(raiz)
raiz.mainloop()
```

Os elementos(widgets) tem outras configurações como:

- height – altura em número de linhas de texto
- width – largura em número de número de caracteres de texto

- font – configura a fonte de texto.
- fg – cor de frente(foreground)
- bg – cor de fundo(background)

Outro exemplo com botão e suas configurações:

```
# -*- coding: cp1252 -*-
from Tkinter import *
class Aplic:
        def __init__(self, pai):
                self.Container1 = Frame(pai)
                self.Container1.pack()
                self.label1=Label(self.Container1)
                self.label1["text"]="Aperte o botão"
                self.label1["fg"] = "cyan"
                self.label1.pack()
                self.botao1= Button(self.Container1)
                self.botao1["text"]= "Aplicação"
                self.botao1["bg"] = "yellow"
                self.botao1["height"]=4
                self.botao1["font"]=('wide latin','14','italic','bold')
                self.botao1.pack()
raiz = Tk()
aplic = Aplic(raiz)
raiz.mainloop()
```

Figura 9: Botão com algumas configurações modificadas.

8.2.5.1 Elementos do Tkinter

Button	Botão simples- usado para executar um comando ou outra operação.
Canvas	Gráficos estruturados. Pode ser usado para plotar e desenhar gráficos, criar editores gráficos e para implementar widgets customizados.
Checkbutton	Representa uma variável que pode ter dois valores distintos. Clicando no botão o valor é trocado.
Entry	Um campo para entrada de texto.
Frame	Um widget tipo container. A Frame pode ter borda e fundo e é usado para agrupar widgets quando criamos uma aplicação ou diálogo.
Label	Mostra um texto ou imagem.

Listbox	Mostra uma lista de alternativas. A listbox pode ser configurada para comportar-se como radiobutton ou checklist.
Menu	Um painel de menu. Usado para implementar menus pull-down e pop-up.
MenuButton	Um botão de menu. Usado para implementar menus pull-down.
Message	Mostra um texto. Similar ao label, mas podemos automaticamente ajustar texto ao tamanho e proporção
Radiobutton	Representa um valor de variável que pode ter vários valores. Clicando o botão, o valor da variável é posicionado para aquele valor e os outros valores são apagados.
Scale	Permite posicionar um valor numérico arrastando um ponteiro.
Scrollbar	Barra de rolamento padrão usada com widgets canvas, entry, listbox e text.
Text	Formata um texto. Permite a você mostrar e editar textos com vários estilos e atributos. Também suporta incorporar imagens e janelas.
Toplevel	Janela de mais alto nível.

Todos os widgtes tem a mesma hierarquia.

Um widgtes típico possui cerca de 150 métodos. A classe Toplevel fornece o gerenciador de interfaces. Além disso temos

os gestores de geometria que permite criar lay-outs, são eles:grid [18] , pack[19] e place[20].

8.2.6 Posicionando os botões

Já verificamos que o método pack é quem cuida da parte visual do widget e ele possue um argumento chamado side que pode assumir os valores: left, right, top, bottom.

```
# -*- coding: cp1252 -*-
from Tkinter import *

class Aplic:

    def __init__(self, pai):

        frame = Frame()
        frame.pack()

        self.botaoE= Button(frame, text="Sai", fg="red",
command=frame.quit)
        self.botaoE.pack(side=LEFT)

        self.botaoD = Button(frame, text="Oi!",
command=self.digaOi)
        self.botaoD.pack(side=RIGHT)

    def digaOi(self):
```

18 Lay-out em uma grade bi-dimensional.

19 Lay-out em blocos retangulares.

20 Permite definir o lay-out do local que colocaremos o widget.

```
        print "Oi Moçada!"

root = Tk()

app = Aplic(root)
root.mainloop()
```

Figura 10: Posicionamento dos Botões

Observe em botaoE e botaoD do programa anterior os comandos que foram associados aos botões.

Posicionamento superior e inferior.

```
# -*- coding: cp1252 -*-
from Tkinter import *

class Aplic:

    def __init__(self, pai):

        frame = Frame()
        frame.pack()
```

```
        self.botaoA= Button(frame, text="Acima", fg="red",
command=self.digaOi[21])
        self.botaoA.pack(side=TOP)

        self.botaoB = Button(frame, text="Abaixo!",
command=self.digaOi)
        self.botaoB.pack(side=BOTTOM)
        self.botaoE= Button(frame, text="Sai", fg="red",
command=frame.quit)
        self.botaoE.pack(side=LEFT)

        self.botaoD = Button(frame, text="Oi!",
command=self.digaOi)
        self.botaoD.pack(side=RIGHT)

    def digaOi(self):
        print "Oi"

root = Tk()

app = Aplic(root)
root.mainloop()
```

21 Command Binding – ligação do evento a uma rotina de comando.

Figura 11: Vários botões posicionados.

Veja nesta figura que após colocarmos um botão no container, ele é preenchido pelo botão e deixa a área restante para os próximos widgets, quando não colocamos o parâmetro side o botão é colocado no topo do container deixando a área logo abaixo para você colocar os outros widgets. Não devemos dar vários posicionamentos aos widgets numa frame uma vez que isto pode causar efeitos indesejados quando alterarmos o tamanho dela. Para melhor posicionar os widgets podemos colocar vários containers.

8.2.7 Manipulando os eventos (event handler)

Agora iremos detalhar como você pode associar um **event handler** a um **event** como clicar um botão. Os events handlers são ações executadas em resposta a um event, para que isso ocorra precisamos associar este evento ao event handler através de método chamado event binding[22] cuja sintaxe é widget.bind(envent, event handler). Outra forma de ligar um manipulador de eventos a um widget é chamado "command

22 Ligação do evento a uma rotina de resposta.

binding"[23], já citada anteriormente. Por ora, vejamos melhor o event binding, e tendo entendido este, será moleza explicar o command binding. Antes de começarmos, vamos esclarecer uma coisa: a palavra "botão" pode ser usada para designar duas coisas inteiramente diferentes:
- um widget Button – um elemento visual que é mostrado no monitor do computador.
- um botão no seu mouse – aquele que você pressiona com o dedo.

```
from Tkinter import *
class Janela:
 def __init__(self,principal):
    # define o container principal
    self.frame=Frame(principal)
    self.frame.pack()
    # coloca no container mensagem de texto
    self.texto=Label(self.frame, text='Clique para ficar amarelo')
    self.texto['width']=26
    self.texto['height']=3
    self.texto.pack()
    # coloca no container um botao com  verde inicialmente
    self.botaoverde=Button(self.frame,text='Clique')
    self.botaoverde['background']='green'
    self.botaoverde.bind("<Button-1>",self.muda_cor)
    self.botaoverde.pack()
 # funcao que muda a cor do botao
 def muda_cor(self, event):
 # Muda a cor do botao!
  if self.botaoverde['bg']=='green':
    self.botaoverde['bg']='yellow'
```

23 Ligação do evento a um comando(rotina de tratamento do evento).

```
    self.texto['text']='Clique para ficar verde'
  else:
    self.botaoverde['bg']='green'
    self.texto['text']='Clique para ficar amarelo'

raiz=Tk()
Janela(raiz)
raiz.mainloop()
```

Fig 12: Tela do programa na condição verde e amarela.

No programa acima inserimos um widget do tipo **Label** que serve para colocarmos rótulos ou melhor textos na janela. Verifique que quando criamos widget ele está associado a um janela principal ou container principal, que é o local onde ele será encaixado.

Ainda no programa acima temos a associação do evento apertar o botão 1 do mouse com a execução da função mudar de cor, veja nesta linha (self.botaoverde.bind("<Button-1>",self.muda_cor)) que temos a string "<Button-1>" que é passada como argumento que representa apertar o botão 1 do mouse, por exemplo se fosse apertar o botão enter a string seria "<Return>".

8.2.7.1 Eventos do Mouse

Vimos no exemplo acima que ao acionarmos o botão 1 do mouse o botão muda de cor, podemos também verificar no exemplo acima que na linha com o rótulo "<Button-1>" na passagem de parâmetro temos um objeto evento sendo passado como argumento, em Tkinter sempre que um evento ocorre um objeto é passado contendo todas as informações necessárias do evento. Um **objeto evento** é um tipo especial de objeto contendo todas as informações necessárias a manipulação do evento. Quando quisermos passar uma sequência de teclas que deve ser acionadas, isto é possível através de uma tupla veja exemplo:

```
# -*- coding: cp1252 -*-
from Tkinter import *
class Janela:
 def __init__(self,toplevel):
    self.frame=Frame(toplevel)
    self.frame.pack()
    self.texto=Label(self.frame, text='Clique para ficar amarelo')
    self.texto['width']=26
    self.texto['height']=3
    self.texto.pack()
    self.botaoverde=Button(self.frame,text='Clique')
    self.botaoverde['background']='green'
    self.botaoverde.bind(("<Button-1>","<Button-
3>"),self.muda_cor)
    self.botaoverde.pack()
 def muda_cor(self, event):
 # Muda a cor do botao!
    if self.botaoverde['bg']=='green':
```

```
      self.botaoverde['bg']='yellow'
      self.texto['text']='Clique para ficar verde'
    else:
      self.botaoverde['bg']='green'
      self.texto['text']='Clique para ficar amarelo'
raiz=Tk()
Janela(raiz)
raiz.mainloop()
# atenção acione botão 1 e depois o botão 2
```

Experimente ainda as strings “<Motion>” (movimentação do mouse), “<Leave>”(sai do botão) e “<ButtonRelase-X>” (botão do meio e X pode ser 1, 2 ou 3).

8.2.7.2 Foco e eventos do teclado

O foco é um mecanismo que serve para indicar que elemento gráfico (widget) está respondendo aos eventos. Um método importante quando falamos em foco é o método widg1.focus_force() que coloca o foco no widget **widg1.** Na tabela abaixo vemos as strings associadas aos eventos quando desenvolvemos um programa.

Tabela de Eventos

Evento	**String**
Apertar tecla Enter	“<Return>”
Apertar tecla Delete	“<Delete>”
Apertar tecla Backspace	“<BackSpace>”
Apertar tecla Escape	“<Escape>”

Apertar Seta para Esquerda	“<Left>”
Apertar Seta para Direita	“<Right>”
Apertar Seta para Baixo	“<Down>”
Apertar Seta para Cima	“<Up>”
Tirar o foco do widget	“<FocusOut>”
Colocar o foco no widget	“<FocusIn>”
Apertando qualquer tecla alfanumérica	“<KeyPress-Alfa>” ou “<Alfa>”
Soltando tecla Alfa	“<KeyRelease-Alfa>”
Apertando duas vezes botão 1 mouse	“<DoubleButton-1>”
Apertando 3 vezes tecla P	“<Triple-KeyPress-P>”
Apertando qualquer botão mouse	“<Any-Button>”
Apertando qualquer tecla	“<Any-KeyPressA>”

Exemplo de programa com eventos:

```
# -*- coding: cp1252 -*-   # usar caracteres acentuados
from Tkinter import *
class Janela:
    def __init__(self,principal):
            self.frame=Frame(principal)
            self.frame.pack()
            self.frame2=Frame(principal)
            self.frame2.pack()
            #coloca em container texto FOCO e APERTE...
```

```
        self.titulo=Label(self.frame,text='FOCO \n Aperte TAB
para Foco',
                          font=('Verdana','13','bold'))
        self.titulo.pack()
        self.msg=Label(self.frame,width=40,height=6,
                          text = 'Teste Foco!')
        self.msg.focus_force()
        self.msg.pack()
      # Definindo o botão 1
        self.b01=Button(self.frame2,text='Botão 1')
        self.b01['bg']='blue'
      # definindo eventos --------
        # se for acionado mouse ou tecla enter
        self.b01.bind("<Return>",self.aperta01)
        self.b01.bind("<Any-Button>",self.botao01)
        # se tab for acionado foca e aperta botão
        self.b01.bind("<FocusIn>",self.fin01)
        self.b01.bind("<FocusOut>",self.fout01)
        self.b01['relief']=RAISED
        self.b01.pack(side=LEFT)
      # -------------------------
      # Definindo o botão 2
        self.b02=Button(self.frame2,text='Botão 2')
        self.b02['bg']='yellow'
        self.b02.bind("<Return>",self.aperta02)
        self.b02.bind("<Any-Button>",self.botao02)
        self.b02.bind("<FocusIn>",self.fin02)
        self.b02.bind("<FocusOut>",self.fout02)
        self.b02['relief']=RAISED
        self.b02.pack(side=LEFT)

  def aperta01(self,event):
```

```
        self.msg['text']='ENTER sobre o Botão 1'

    def aperta02(self,event):
        self.msg['text']='ENTER sobre o Botão 2'

    def botao01(self,event):
        self.msg['text']='Clique sobre o Botão 1'

    def botao02(self,event):
        self.msg['text']='Clique sobre o Botão 2'
     def fin01(self,event):
        self.b01['relief']=SUNKEN
    def fin02(self,event):
        self.b02['relief']=SUNKEN
    def fout01(self,event):
        self.b01['relief']=RAISED #desaperta botão 1
    def fout02(self,event):
        self.b02['relief']=RAISED
raiz=Tk()
Janela(raiz)
raiz.mainloop()
```

Figura 13: Tela do programa

8.2.8 Usando CheckButton

Este tipo de elemento(widget) é usado quando queremos fazer escolha entre valores distintos.
`Para usar um Checkbutton`, cria-se uma variável Tkinter:

```
var = IntVar()
c = Checkbutton(master, text="Expand", variable=var)
```

Como padrão esta variável é posicionada em 1 quando o botão é selecionado e 0 quando não. Você pode mudar estes valores usando as opções onvalue e offvalue. Esta variável não precisa ser um número inteiro.

```
 var = StringVar()
  c = Checkbutton(
      master, text="Color image", variable=var,
```

```
    onvalue="Ligado", offvalue="Desligado"
    )
```

Exemplo:

```
# -*- coding: cp1252 -*-
from Tkinter import *
class botaoescolha:
   def __init__(self, master):
      self.frame=Frame(master,width=500,height=100)
      master.title("Check Button")
      self.frame.pack()
      self.var = IntVar()
      self.msg=Label(self.frame,text='Botão Check')
      self.msg.pack()
      c = Checkbutton(master, text="Escolha",
                  variable=self.var,height=8 , width=20)
      c.bind("<1>",self.cb)
      c.pack()

   def cb(self,event):
      print self.var
      if self.var==1:
         print "desligado"
         self.var=0
      else:
         print "ligado"
         self.var=1

raiz=Tk()
```

```
botaoescolha(raiz)
raiz.mainloop()
```

Fig. 14: Tela do programa

8.2.9 Criando Menus

Para criar um menu você deve instanciar a classe Menu e usar o método add para adicionar entradas.

- **add_command(label=string, command=callback)** - adiciona uma entrada ao menu.
- **add_separator()** - adiciona um separador de linha. Usado para agrupar entradas.
- **add_cascade(label=string, menu=menu instance)** - adiciona um submenu.

```
from Tkinter import *
def mostra():
  filewin = Toplevel(root)
  button = Button(filewin, text="Mostra Botao")
```

```
   button.pack()

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="Novo", command=mostra)
filemenu.add_command(label="Abrir", command=mostra)
filemenu.add_command(label="Salvar", command=mostra)
filemenu.add_command(label="Fechar", command=mostra)

filemenu.add_separator()

filemenu.add_command(label="Sai", command=root.quit)
menubar.add_cascade(label="Arquivo", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Desfaz", command=mostra)

editmenu.add_separator()

editmenu.add_command(label="Apags", command=mostra)
editmenu.add_command(label="Seleciona Todos",
command=mostra)

menubar.add_cascade(label="Editar", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Ajuda", command=mostra)
helpmenu.add_command(label="Sobre...", command=mostra)
menubar.add_cascade(label="Ajuda", menu=helpmenu)

root.config(menu=menubar)
root.mainloop()
```

Fig. 15: Tela do Programa

8.2.10 Barra de ferramentas

Vários programas colocam abaixo da barra de menu uma barra de ferramentas, normalmente esta barra contém vários botões como abrir arquivo, imprimir, etc...

Exemplo de barra de ferramentas e uma frame com vários botões.

```
#arquivo Tkintertoolbar.py
from Tkinter import *

root = Tk()

def chamada():
        print "Aviso"

# create a menu
menu = Menu(root)
```

```
filemenu = Menu(menu)
menu.add_cascade(label="Arquivo", menu=filemenu)
filemenu.add_command(label="Novo", command=chamada)
filemenu.add_command(label="Abrir...", command=chamada)
filemenu.add_separator()
filemenu.add_command(label="Sair", command=chamada)
helpmenu = Menu(menu)
menu.add_cascade(label="Ajuda", menu=helpmenu)
helpmenu.add_command(label="Sobre...",
command=chamada)



# criar barra de ferramentas
toolbar = Frame(root)
b = Button(toolbar, text="novo", width=6,
command=chamada)
b.pack(side=LEFT, padx=2, pady=2)
b = Button(toolbar, text="abre", width=6, command=chamada)
b.pack(side=LEFT, padx=2, pady=2)
toolbar.pack(side=TOP, fill=X)
root.config(menu=menu)
root.mainloop()
```

Figura 16: Tela do Programa

8.2.11 Alarme com Tkinter

Vejamos como gerar um sinal sonoro.

```
from Tkinter import *

class Alarm(Frame):
    def repete(self):
        self.bell()
        self.after(self.msecs, self.repete)
    def __init__(self):
        Frame.__init__(self)
        self.msecs = 100
        self.pack()
        parar = Button(self, text='Parar sons!',
command=self.quit)
        parar.pack()
        parar.config(bg='navy', fg='white', bd=8)
        self.parar = parar
        self.repete()

if __name__ == '__main__':
   Alarm().mainloop()
```

Fig. 17: Tela do Programa

8.2.12 Criar uma janela de entrada de dados

Este exemplo cria um elemento janela de alto nível e adiciona alguns alguns elementos, um rótulo e uma caixa de texto.

```
from Tkinter import *

root = Tk()
root.title('Entrada')
Label(root, text="Dados:").pack(side=LEFT, padx=5, pady=10)
e = StringVar()
Entry(root, width=40, textvariable=e).pack(side=LEFT)
e.set("'Digite aqui!'")
root.mainloop()
```

Figura 18: Tela do Programa

O widget **Entry** é uma forma de entrada de dados. Os dados informados pelo usuário neste campo estão na forma de strings, como a função raw_input. Através do método **get**, fornecido pelo **Entry** podemos utilizar os dados entrados em nosso código. Este widget possui as mesmas configurações que botões e labels exceto a altura **height** que neste caso é de uma linha. Para entrada de senhas podemos utilizar a opção show='*' que mostra asteriscos em vez dos dados teclados.

Outro exemplo:

```
from Tkinter import *
class entradadedados:
    def __init__(self,Pai):
        self.pai=Frame(Pai)
        self.pai.pack()
        Label(self.pai,text="Nome",fg="black").grid(row=0)
        Label(self.pai,text="Endereco").grid(row=1)
        self.e1=Entry(self.pai)
        self.e2=Entry(self.pai)
        self.e1.grid(row=0, column=1)
        self.e2.grid(row=1,column=1)

root=Tk()
Raiz=entradadedados(root)
Raiz.pai.mainloop()
```

Fig. 19: Tela do Programa

Mais um exemplo e neste usamos uma uma opção chamada de **pad** para dar um espaço extra entre bordas e widgets dentro da frame.

```
self.b02=Button(self.frame2,text='Botão 2')
self.b02['padx'],self.b02['pady'] = 10, 5
```

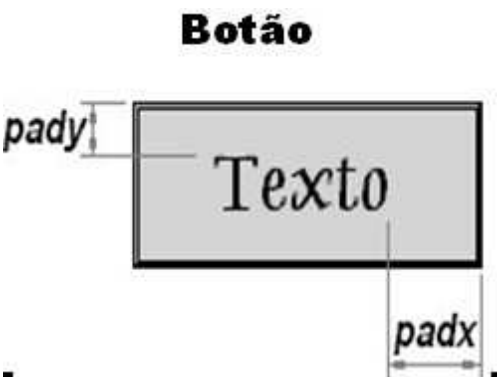


Figura 20: pady e padx em um elemento do tipo botão, aumentando as bordas.

```
from Tkinter import *
class Senhas:
    def __init__(self,principal):
        self.frame1=Frame(principal)
        self.frame1.pack()
        Label(self.frame1,text='SENHAS', fg='darkblue',
        font=('Verdana','14','bold'), height=3).grid(row=0)
        fonte1=('Verdana','10','bold')
        Label(self.frame1,text='Nome: ',
        font=fonte1,width=8).grid(row=1,column=0)
        self.nome=Entry(self.frame1,width=10,
        font=fonte1)
        self.nome.focus_force()
        self.nome.grid(row=1,column=1)
        Label(self.frame1,text='Senha: ',
        font=fonte1,width=8).grid(row=2,column=0)
        self.senha=Entry(self.frame1,width=10,show='*',
```

```
        font=fonte1)
        self.senha.grid(row=2,column=1)
        self.confere=Button(self.frame1, font=fonte1,
text='Envia',
        bg='pink', command=self.conferir,padx=10,pady=10)
        self.confere.grid(row=3,column=0)
        self.msg=Label(self.frame1,font=fonte1,
        height=3,text='AGUARDANDO...')
        self.msg.grid(row=4,column=0)
    def conferir(self):
        NOME=self.nome.get()
        SENHA=self.senha.get()
        if NOME == SENHA:
            self.msg['text']='ACESSO PERMITIDO'
            self.msg['fg']='darkgreen'
        else:
            self.msg['text']='ACESSO NEGADO'
            self.msg['fg']='red'
            self.nome.focus_force()
instancia=Tk()
Senhas(instancia)
instancia.mainloop()
```

Fig. 21: Tela do Programa

8.2.13 Usando o configure

```
# -*- coding: cp1252 -*-
from Tkinter import *
principal = Frame()
principal.pack()
rotulo = Label (principal, text="Rótulo Exemplo",
foreground="blue")
rotulo.pack ()
rotulo.configure(relief="ridge", font="Arial 24 bold",
border=5, background="yellow")
principal.mainloop()
```

Com configure:

Figura 22: Tela do Programa

Sem configure:

Figura 23: Sem o comando configure

Outro exemplo do configure:

```
from Tkinter import *
principal = Frame() ; principal.pack()
a = Label (principal, text="1") ; a.pack (side="left")
b = Label (principal, text="2") ; b.pack (side="bottom")
c = Label (principal, text="3") ; c.pack (side="right")
d = Label (principal, text="4") ; d.pack (side="top")
e = Label (principal, text="5") ; e.pack (side="top")
for widget in (a,b,c,d,e):
        widget.configure(relief="groove", border=10,
        font="Times 24 bold")
principal.mainloop()
```

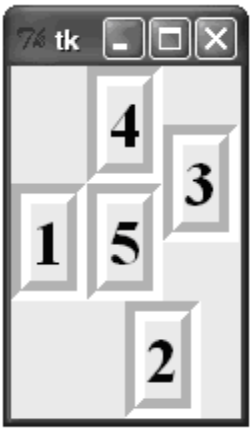


Figura 24: Tela do Programa

8.2.14 Usando a opção fill

fill ('none', 'x', 'y' ou 'both')

- Indica como o desenho do elemento irá preencher o espaço alocado
- 'x' e 'y' indica que irá preencher a largura e altura
- 'both' indica preenchimento de todo o espaço
- 'none' indica que apenas o espaço necessário será ocupado (default)

```
from Tkinter import *
principal = Frame() ; principal.pack()
a = Label (principal, text="1") ; a.pack (side="left")
b = Label (principal, text="2") ; b.pack
(side="bottom",fill="x")
c = Label (principal, text="3") ; c.pack (side="right",fill="y")
```

```
d = Label (principal, text="4") ; d.pack (side="top")
e = Label (principal, text="5") ; e.pack (side="top")
for widget in (a,b,c,d,e):
        widget.configure(relief="groove", border=10,
        font="Times 24 bold")
principal.mainloop()
```

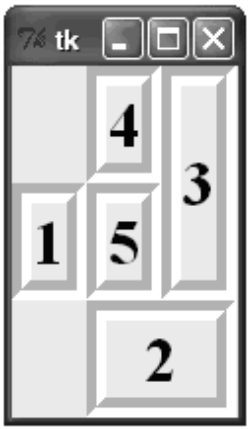


Figura 25: Usando a opção fill

8.2.15 Canvas

O widget canvas fornece uma facilidade de estrutura gráfica para o Tkinter. Ele é altamente versátil e é usado para desenhar e plotar gráficos, criar editor gráfico e criar widgets padronizados, até animações são possíveis. Por default, novos itens são desenhados acima dos itens que já estão no Canvas. O Tkinter fornece um conjunto de métodos permitindo a você

manipular os itens de várias formas e adicionar eventos e retorno deles individualmente.

Você deve usar o widget Canvas para mostrar e editar gráficos e outros desenhos e ainda implementar vários tipos de widgets padronizados. Por exemplo: você pode usar um Canvas como uma barra de conclusão atualizando um objeto retângulo.

O Canvas suporta os seguintes itens:

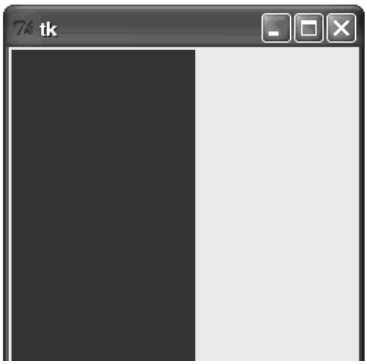
- arcos – arco de círculo
- bitmap – imagem binária
- imagem – imagem colorida
- linha – linha poligonal
- oval – círculos e elipses
- polígonos
- retângulo
- texto
- janela – um widget tk

Define-se um Canvas da mesma forma que um outro widget qualquer, além de possuir as opções de configuração de botões e labels, ele possui as opções **bd** e **cursor**.

A opção **bd** especifica a espessura da borda do widget em pixels e a opção **cursor** define de que forma o cursor aparece quando está posicionado no widget. Existem muitas opções de cursor válidas para qualquer widget. Vamos ao nosso primeiro exemplo.

```
from Tkinter import *
class Kanvas:
   def __init__(self,raiz):
      self.canvas1 = Canvas(raiz, width=100, height=200,
      cursor='star', bd=15,
      bg='red')
```

```
        self.canvas1.pack(side=LEFT)
        self.canvas2 = Canvas(raiz, width=100, height=200,
        cursor='star', bd=5,
        bg='yellow')
        self.canvas2.pack(side=LEFT)
instancia=Tk()
Kanvas(instancia)
instancia.mainloop()
```



Veja que no programa anterior criamos os canvas diretamente na janela top-level(principal).

Sistema de Coordenadas Canvas

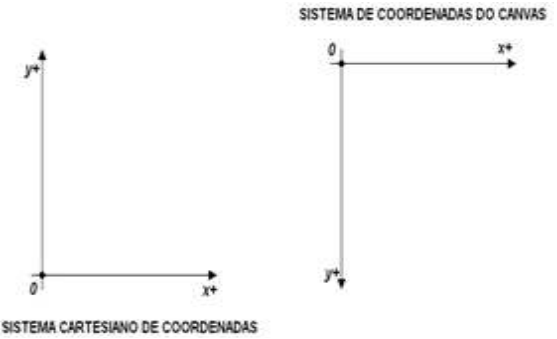


Fig. 26: Sistema de coordenadas Canvas

Veja que a coordenada 0,0 está situada a esquerda e no alto e o eixo y está invertido.

Criando linha em Canvas: a sintaxe do comando para desenhar uma linha num canvas é:**self.nome_do_canvas.create_line(x1,y1,....,xn,yn)** – a linha começa em x1,y1 e segue a x2,y2 e depois x3, y3 até xn,yn. Podemos definir a opção **width=número** que serve para definir espessura da linha.

```
from Tkinter import *
class Kanvas:
   def __init__(self,raiz):
      self.canvas1 = Canvas(raiz, width=400, height=400,
      cursor='fleur', bd=10,
      bg='red')
```

```
        self.canvas1.create_line(200,200,230,230,100,150,fill='bla
ck',width=4)
        self.canvas1.pack()
instancia=Tk()
Kanvas(instancia)
instancia.mainloop()
```

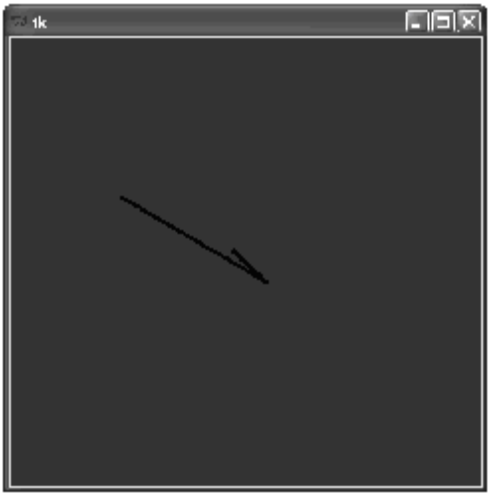


Figura 27: Desenhando uma linha

Texto e Círculo no Canvas:

```
# -*- coding: cp1252 -*-
#File: canvas4.py
from Tkinter import *
class Kanvas:
   def __init__(self,raiz):
      self.canvas1 = Canvas(raiz, width=400, height=400,
      cursor='fleur', bd=10,
      bg='red')
      xy = 20, 20, 300, 180
      self.canvas1.create_arc(xy, start=0, extent=270, fill="red")
      self.canvas1.create_arc(xy, start=270, extent=60,
fill="blue")
      self.canvas1.create_arc(xy, start=330, extent=30,
fill="green")
      self.canvas1.create_text(200,300,text="GrÃ¡fico",font=('
Arial','26','bold'))
      self.canvas1.pack()
      self.frame=Frame(instancia)
      self.label=Label(text="Desenhando linhas")
      self.frame.pack()
      self.label.pack()
instancia=Tk()
Kanvas(instancia)
instancia.mainloop()
```

Figura 28: Desenhando texto e círculos no Tkinter

Desenhando retângulo e polígono:

```
# -*- coding: cp1252 -*-
from Tkinter import *
class Kanvas:
   def __init__(self,raiz):
      self.canvas1 = Canvas(raiz, width=400, height=400,
      cursor='fleur', bd=10,
      bg='red')
      xy = 20, 20, 300, 180
      self.canvas1.create_arc(xy, start=0, extent=270, fill="red")
      self.canvas1.create_arc(xy, start=270, extent=60,
fill="blue")
      self.canvas1.create_arc(xy, start=330, extent=30,
fill="green")
      self.canvas1.create_text(200,350,text="GrÃ¡fico",font=('
Arial','26','bold'))
      self.canvas1.create_polygon(200,200,200,300,300,300,20
0,200, fill='yellow')
      self.canvas1.create_rectangle(200,150,250,200,fill='blue')
      self.canvas1.pack()
      self.frame=Frame(instancia)
      self.label=Label(text="Desenhando linhas")
      self.frame.pack()
      self.label.pack()
instancia=Tk()
Kanvas(instancia)
instancia.mainloop()
```

Figura 29: Desenhando no Tkinter

Mais comando do Canvas do Tkinter:

```
from Tkinter import *
class Palheta:
    def __init__(self,raiz):
        raiz.title("Palheta")
        self.canvas=Canvas(raiz, width=200, height=200)
        self.canvas.pack()
        self.frame=Frame(raiz)
        self.frame.pack()
        self.canvas.create_oval(15, 15, 185, 185,
        fill='white', tag='bola')
        Label(self.frame,text='Vermelho: ').pack(side=LEFT)
        self.vermelho=Entry(self.frame, width=4)
        self.vermelho.focus_force()
        self.vermelho.pack(side=LEFT)
        Label(self.frame,text='Verde: ').pack(side=LEFT)
        self.verde=Entry(self.frame, width=4)
        self.verde.pack(side=LEFT)
        Label(self.frame,text='Azul: ').pack(side=LEFT)
        self.azul=Entry(self.frame, width=4)
        self.azul.pack(side=LEFT)
        Button(self.frame, text='Mostrar',
        command=self.misturar).pack(side=LEFT)
        self.rgb=Label(self.frame, text='', width=8,
        font=('Verdana','10','bold'))
        self.rgb.pack()
    def misturar(self):
        cor="#%02x%02x%02x" %(int(self.vermelho.get()),
        int(self.verde.get()),
        int(self.azul.get()))
        self.canvas.delete('bola')
```

```
        self.canvas.create_oval(15, 15, 185, 185,
        fill=cor, tag='bola')
        self.rgb['text'] = cor
        self.vermelho.focus_force()
inst = Tk()
Palheta(inst)
inst.mainloop()
```

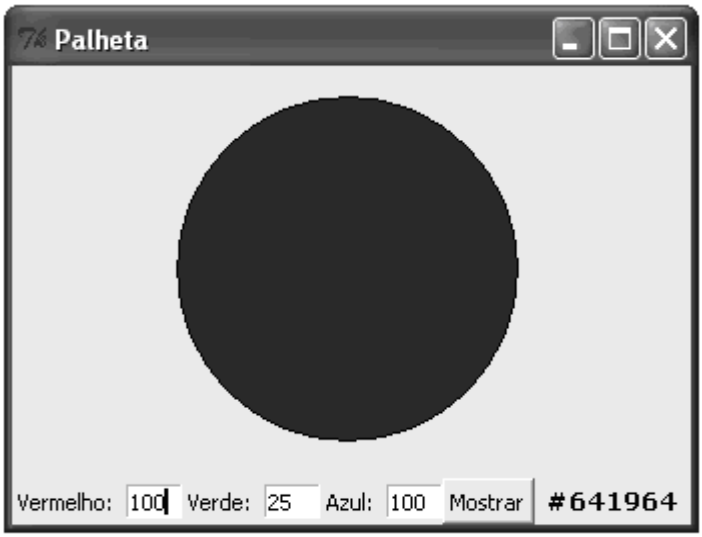


Figura 30: Palheta de cores.

9. CGI e Servidor HTTP

9.1 O CGI (Common Gateway Interface)

É uma padronização para troca de informações com servidores de aplicações externas, tais como o HTTP ou servidores Web. Essas especificações são mantidas pela NCSA[24]. Um documento HTML é estático, ou seja é um arquivo tipo texto imutável, por outro lado um programa CGI gera uma informação dinâmica, já que ele é executado em tempo real. Por exemplo, digamos que você queria permitir que as pessoas acessem seu banco de dados através da WEB. Basicamente você precisa criar um programa CGI, que o servidor Web irá executar, para transmitir as informações para o motor do banco de dados, receber os resultados e exibi-los de volta para o cliente. Este é um exemplo de um gateway[25], e este é o lugar onde CGI, versão 1.1, tem as suas origens. O exemplo de banco de dados é uma ideia simples, mas na maioria das vezes é bastante difícil de implementar. Realmente não há nenhum limite quanto ao que você pode se conectar pela Web. A única coisa que você precisa lembrar é que qualquer que seja o seu programa CGI seu tempo de processamento deve ser o menor possível, caso contrário, o usuário vai apenas ficar olhando para seu navegador a espera que alguma coisa aconteça. Como os programas CGI são executáveis, existem algumas precauções de segurança que precisam ser implementadas. O que mais afeta o usuário típico da WEB é o fato de que programas CGI necessitam residir em um diretório especial, de modo que o servidor Web saiba que é para executar o programa e não apenas apresentá-lo ao navegador. Este diretório esta

24 NCSA - National Center for Supercomputing Applications

25 Gateway -é uma ponte entre duas redes, permitindo a conexão entre elas.

geralmente sob controle do webmaster[26] que proíbe determinados usuários de criar programas CGI. O programa CGI fica alojado no diretório chamado de cgi-bin um diretório especial que residem todos programas CGI. Um programa CGI pode ser escrito em: C, Fortran, Perl, Visual Basic. Python, etc.. Só depende do que você tem disponível em seu sistema. Se você usar uma linguagem de programação como C ou Fortran, você sabe que você tem que compilar o programa para que ele seja executado, no entanto, usar se um dos scripts em vez linguagem, tais como Pytthon, Perl, Tcl, ou um shell Unix, o script em si só também tem de residir no cgi-bin. Muitas pessoas preferem escrever scripts CGI, em vez de programas, uma vez que eles são mais fáceis de depurar, modificar e manter do que um típico programa compilado.

9.2 O servidor Apache

É o mais bem sucedido servidor HTTP livre, criado em 1995 por Rob McColl, funcionário do NCSA. Este servidor atualmente está sob os cuidados da Apache Software Foundation, que tem uma dezena de projetos para WEB, e ele é compatível com o protocolo HTTP versão 1.1. O Apache Server, como é conhecido, é um software livre, o que significa que qualquer um pode estudar ou alterar seu código-fonte, além de poder utilizá-lo gratuitamente. Graças ao trabalho muitas vezes voluntário de vários desenvolvedores, o Apache continua sendo o servidor Web mais usado no mundo. Além de estar disponível para o Linux (e para outros sistemas operacionais baseados no Unix), o Apache também conta com versões para o Windows, para o Novell Netware e para o OS/2, o que o torna uma ótima opção para rodar em computadores obsoletos (desde que este

26 Webmaster – responsável pela manutenção e administração de um site.

atenda aos requisitos mínimos de hardware). O servidor Apache é capaz de executar código em C, C#, C++, Java, Perl, PHP Python, Ruby, Scala, SVG Scalable Vector Graphics, Tcl e até em ASP e pode atuar como servidor FTP, HTTP, entre outros. Sua utilização mais conhecida é a que combina o Apache com a linguagem PHP e o banco de dados MySQL.

9.3 HTTP e HTML

A web é construída a partir de duas tecnologias fundamentais: a linguagem HTML e o protocolo HTTP. O HTML[27] é a codificação usada para criar as páginas da Web, usaremos esta linguagem em nosso exemplos. O segundo pilar da Web é o HTTP[28], protocolo de transporte de hipertexto. Esse é o conjunto de comandos e regras que define como deve ser a comunicação entre um browser[29] (como o Mozilla) e um servidor HTTP como o Apache ou o IIS[30]. A expressão "servidor HTTP" pode significar duas coisas: o software que serve páginas via HTTP, ou o computador onde esse software está instalado. No mundo Unix, softwares servidores são chamados de "daemons", e a sigla HTTPd descreve um "HTTP daemon" genérico. Esta relação entre o browser e o servidor é chamada cliente-servidor, desta forma quando você digita uma URL[31] como: www.capotasprudentina.com.br/index.html, o seu browser localiza e conecta-se ao servidor www.capotasprudentina.com.br e envia-lhe o comando GET/index.html, o servidor lê o arquivo index.html e transmite

27 HTML - Hypertext Markup Language
28 HTTP - Hypertext Transport Protocol
29 Browser - navegador
30 IIS – Internet Information Services – servidor web da Microsoft
31 URL – Uniform Resource Locator ou *Localizador-Padrão de Recursos*

o conteúdo para o cliente e encerra a conexão, desta forma temos os seguintes passos na comunicação: conexão, solicitação, resposta e encerramento da conexão. A página index.html é o que chamamos de página estática.

9.4 Páginas Dinâmicas

Este tipo de solução apresentada acima atende algumas necessidades, mas com o advento do comércio eletrônico, chats e outras transações, tornou-se necessário novas tecnologias. A mais antiga tecnologia de páginas dinâmicas é o CGI um protocolo básico para interação entre um HTTPd e um programa gerador de páginas dinâmicas[32], é com ele que trabalharemos. Outras tecnologias de páginas dinâmicas são: Zope(escrito em Python), ASP(Microsoft), ColdFusion(Allaire), PHP etc..

9.5 Servidor Apache

Para testarmos os nossos scripts CGI, sugiro que instalemos o Servidor Apache.

Instalando o Apache:

- Baixe o Apache para Windows em:
  http://www.apache.org/dist/httpd/binaries/win32
  Utilize a última versão msi(neste caso 2.2.11) que é um pacote compactado do Windows Installer para versões recentes do windows.
  (http://www.apache.org/dist/httpd/binaries/win32/apache

32 - O nome páginas dinâmicas vem do mecanismo de montagem das páginas na hora da solicitação da mesma.

_2.2.11-win32-x86-openssl-0.9.8i.msi)

- **Importante**: instale o software usando uma conta com direitos administrativos no Windows.
  Ao baixá-lo, inicie a instalação, como você faria com qualquer programa para Windows com um bom instalador. Durante a instalação ele pedirá o domínio da rede (Network Domain), o nome do servidor (Server Name) e o e-mail do administrador do sistema. Como a instalação é para um servidor local, para desenvolvimento, preencha com "localdomain" no campo "Network Domain", "localhost" no campo "Server Name" e seu e-mail no último campo.

- Finalizada a instalação, o apache deverá estar funcionando. No Windows 2000/XP ele é instalado como um serviço que pode ser inicializado/desativado/reiniciado usando o console de

serviços, o "services.msc". Por padrão, ele se configura para ser iniciado sempre junto com o Windows (veja na imagem de tela acima a opção na instalação para mudar isso, além de mudar a porta para 8080). Ao fazer alguma alteração num arquivo de configuração do Apache ou ao instalar um módulo, por exemplo, deve-se reiniciar o servidor (não o computador, mas sim o serviço do Apache). Isso pode ser feito com o "Apache Service Monitor", que ficará ativo na área de notificação (bandeja do sistema, próximo ao relógio):

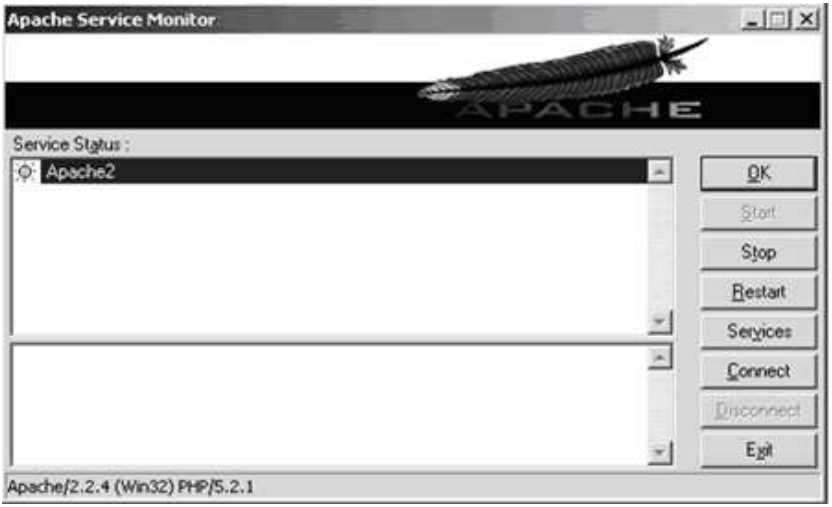
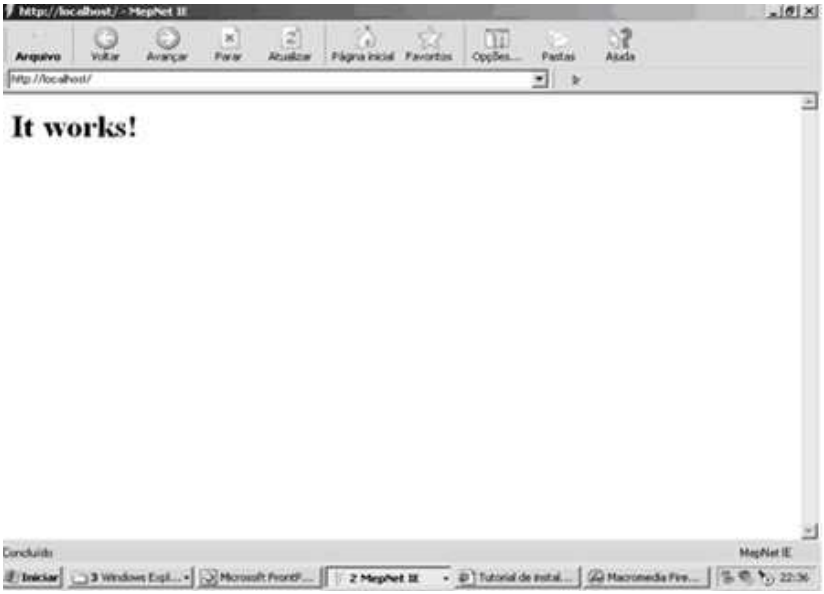


Fig. 31: Com esta interface pode-se iniciar, reiniciar ou parar o servidor Apache no Windows.

- Com ele instalado, abra qualquer navegador e digite o nome do seu computador, ou, de preferência, o nome "localhost", que sempre se refere ao computador local.

Se preferir, acesse http://127.0.0.1, o IP da máquina local. Se tudo deu certo até aqui, você deverá ver uma mensagem "It works" ("Ele trabalha", "Ele funciona").



- As páginas ficam por padrão na pasta "htdocs", dentro da pasta onde o Apache foi instalado. No meu caso, seria "C:\Apache2.2\htdocs". Basta substituir o conteúdo do arquivo "index.html" pelo do seu site, e colocar suas páginas e subpastas aí. Você poderá acessar as subpastas da pasta "htdocs" digitando http://localhost/pasta, em qualquer navegador no micro local, o nome do arquivo tem que se index.html. Se ao digitar localhost o apache não mostrar a página acima desinstale o Apache e instale novamente.

Instalado o Apache, ele lê basicamente HTML puro. O acesso ao arquivo index.html, no navegador, pode ser feito de duas formas:"http://localhost/index.html" ou "C:\meu site\index.html". No primeiro caso, o navegador solicita a página ao computador "localhost" (que é o seu computador!), recebe-a do servidor, armazena-a no cache, para depois exibi-la. Ele trata como se fosse um site da Internet; como o servidor do site está na rede local (ou mesmo no seu próprio PC, um computador só), trata-se de uma Intranet. No segundo caso, você está acessando o arquivo diretamente no seu HD. O navegador pode até armazenar algum dado no cache, mas não há nenhuma comunicação com nenhum servidor. Ele lê o arquivo como qualquer programa leria qualquer arquivo.

- A configuração do Apache está em alguns arquivos de texto puro, no estilo dos arquivos ".ini", e fica na pasta "conf", dentro da pasta do Apache. Ao editá-los e salvá-los, reinicie o servidor do apache para que as alterações entrem em vigor (usando o ícone do Apache na área de notificação, ou o services.msc, ou ainda o comando net stop apache2, e net start apache2, num prompt de comando). Os arquivos de configuração usados na versão Windows do Apache são o "httpd.conf" e o "mime.types".
- O "httpd.conf" é o principal, abordando diversas características do Apache. O "mime.types" define os tipos mime, tipos de arquivos e suas extensões, para que o Apache saiba como tratá-los. Por exemplo, ele deve saber que deve enviar arquivos HTML diretamente, sem processá-los, mas deve processar os arquivos PHP, antes

de entregar o resultado ao browser. É importante que você faça backup destes arquivos, para restaurá-los, se você editar algo e der errado. Em ambos as linhas iniciadas com o caractere # são comentários, e são ignoradas. Para desativar um item ou colocar explicações, basta iniciar a linha com #.

- É possível aparecer um erro na inicialização do Apache, se você tiver um outro servidor web ativo na máquina. A provável causa pode ser o uso da porta 80, se você mantém o IIS ativo, já que ele normalmente é iniciado antes e ocupa a porta 80 (praticamente qualquer servidor web se configura para usar a porta 80, que é dada como padrão). Para contornar isso e manter os dois servidores ativos, você deve trocar a porta de pelo menos um deles. No arquivo "httpd.conf" do Apache, localize o item "Listen 80", e troque o valor 80 por outro número que não esteja sendo usado por nenhuma outra porta. Por exemplo, 8080, 8081, etc. Localize mais para frente "ServerName localhost:80", e troque o 80 daí também.
- Salve o arquivo e reinicie o servidor. Agora, você deve acessar os sites digitando ":8080" após o "domínio". Como é local, você deverá digitar: "http://localhost:8080", trocando é claro, 8080 pela porta escolhida por você. Se entrar sem definir a porta, o navegador muito provavelmente irá usar a 80, o que resultará que o site aberto será o do outro servidor (como o IIS, por exemplo). Sempre use o número da porta separado por dois pontos do nome do domínio. O certo é "http://localhost:8080/teste.htm", e não "http://localhost/teste.htm:8080". Preocupe-se com isso apenas se você possuir mais de um servidor web na

mesma máquina. Veja este tutorial na internet.**[PIC]**

10. Programas CGI

10.1 Introdução

Nosso primeiro programa é o Olá.

```
#!\python25\python


print 'Content-type:text/html'
print
print '<HTML><BODY>'
print "<H1>OLA'</H1>"
print '</BODY></HTML>'
```

Comentando o programa:

- Em primeiro lugar devemos salvar o script em um diretório chamado: C:\Apache2.2\cgi-bin. O nome do arquivo será: ola.py
- A primeira linha é um comentário especial marcado pelos caracteres '#!'. Os sinais '#!' devem estar encostados na margem esquerda da página, e o restante deve conter o caminho até o programa que executará o script.(\python25\python ou /python25/python representação UNIX).
- Na linha 3 encontramos o comando print 'Context-type:text:html' produz a parte obrigatória do cabeçalho da página, exigida pelo CGI. Este cabeçalho define o documento a ser transmitido de acordo com o esquema

de classificação chamado MIME[33] (O MIME define uma coleção de cabeçalhos do e-mail para especificar atributos adicionais de uma mensagem incluindo o tipo de caractere do texto e dos arquivos de transferência. O MIME é extensivo, as suas definições incluem métodos para registrar novos conteúdos e valores. O 'text/html' é um tipo MIME padrão de documentos HTML. Embora originalmente definido por MIME e-mail, o cabeçalho *conteúdo-tipo* e o tipo MIME de registro é reutilizado em outros protocolos de Internet, como HTTP. O tipo de registro MIME é administrado por IANA[34].). Um arquivo tipo ascii é definido com o MIME type 'text/plain', um arquivo do tipo jpeg com 'image/jpeg'.

- Na linha 4 temos um print vazio, que gera uma linha em branco, que marca o fim do cabeçalho.
- E por fim temos um código HTML.
- Para executar o programa digite: localhost/cgi-bin/ola.py ou 127.0.1.0/cgi-bin/ola.py
- Mensagens como: Not Found – erro na digitação do caminho ou "Internal Server Error" erro quando o script CGI não gera um cabeçalho mínimo, são comuns e fique atento.

Comentário final: Se ao rodar o script a partir do prompt ou do IDLE você está vendo um traceback do interpretador Python, o problema está mesmo dentro do seu programa. Quando ocorre um erro de sintaxe (SyntaxError) o interpretador apenas leu, mas não chegou a executar nenhuma linha do seu script. Assim, o

33 MIME – Multipurpose Internet Mail Extension: especificação para o formato de anexos de e-mail que não são textos.

34 IANA - Internet Assigned Number Authority

famoso cabeçalho "Content-type: ..." e a linha em branco não são enviados para o servidor, e o traceback que o ajudaria a detectar o problema não chega ao seu browser, mas vai para um arquivo onde o Apache registra mensagens de erro. Este arquivo chama-se error.log e fica em /apache/logs/. Você pode inspecioná-lo com qualquer editor de texto. Outras vezes, o erro pode acontecer durante a execução e após o envio do cabeçalho. Neste caso, o traceback é perdido para sempre. É por isso que programadores de CGI experientes procuram testar exaustivamente seus scripts a partir da linha de comando antes de tentar acioná-lo pelo browser.

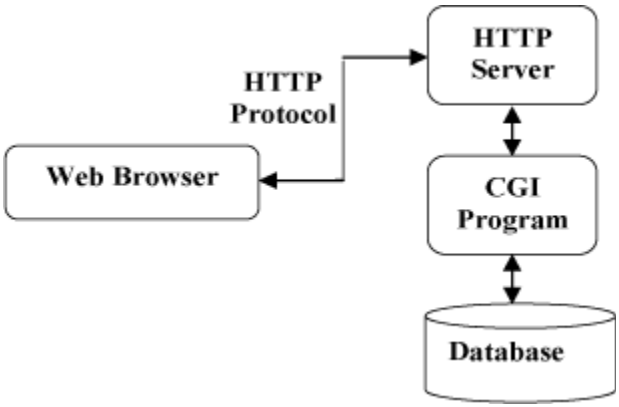


Fig. 32: O navegador acessando um site que possui uma base de dados(ex: site de banco)

De sempre um nome significativo ao programa em Python.

```
#!C:\\Python27\\python.exe
#-*- coding: ISO-8859-1 -*-

print "Content-type: text/html; ISO-8859-1"
print
```

```
print '''
    <html>
       <head>
          <title> Programando em Python para WEB :: Tutorial 1</title>
       <head>
       <body>
          Olá
       </body>
    <html>
'''
```

A 2a. Linha é usada para o uso de caracteres acentuados, veja também no browser o efeito da linha do título <title>. Retire a linha 2 e veja o erro, entre no diretório de logs do apache e veja o erro dentro do arquivo de texto error. Para executar o programa entre no navegador e digite: http://localhost/cgi-bin/ola2.py

Retirando a linha 2.

Internal Server Error

The server encountered an internal error or misconfiguration and was unable to complete your request.

Please contact the server administrator, admin@s-abecf5841b944.ap and inform them of the time the error occurred, and anything you might have done that may have caused the error.

More information about this error may be available in the server error log.

10.2 Mais Programas

10.2.1 Dados do ambiente

```
#!/python27/python

import os

print "Content-type: text/html";
print
print "<font size=+1>Environment</font>"
for param in os.environ.keys():
  print "<font size=+1>%20s</font>: %s" % (param,os.environ[param])
```

10.2.2 Data e Hora do Servidor atualizadas

```
#!/python27/python
# hora.py - CGI que exibe a hora local do servidor
from time import time, localtime

print 'Content-type: text/html'
print 'Refresh: 0.6'
print

a, ms, d, h, m, s = localtime(time())[0:6]
print '<HTML><BODY>'
print '<H1>Data: %02d/%02d/%02d  Hora: %02d:%02d:%02d</H1>' % (d,ms,a,h, m, s)
print '<P>* de acordo com o relogio interno deste
servidor</P>'
print '</BODY></HTML>'
```

Figura 33: Tela de execução do programa

Comentando o programa:

- Na linha 3 são importados do módulo time as funções time e localtime.
- Na linha 6 o sistema pede um refresh da página a cada 0,6s de tal forma que a hora é atualizada, este tempo de 0,6s é para que o sistema mostre sempre os segundos corretamente. Mude para 1.0s e veja o que acontece.
- Na linha temos a impressão da data e hora.

Veja neste programa que o ideal seria que o servidor envia-se a informação a cada segundo, mas com o protocolo HTTP cliente-servidor não há como o servidor enviar a informação uma vez que isto só é feito por iniciativa do cliente, isto é, quando o cliente requisita a informação. Esta é uma limitação importante na hora de implementarmos nossos programas CGI. Para solucionar este problemas os navegadores modernos tem um cabeçalho especial chamado Refresh, cuja

presença instrui o browser a solicitar a página após algum tempo.

10.3 Métodos GET e POST

Em determinadas situações nós precisamos passar informações de nosso browser para o servidor e por fim para o programa CGI, nestes casos a maioria dos browsers usam dois métodos que são os métodos: get e post.

Passando informação através do método get: o método get adiciona a informação na string de requisição da página, e pode ter no máximo 1024 caracteres, por este método não devemos passar senhas e informações importantes.
Passagem de parâmetro:

```
http://localhost/cgi-bin/get1.py?first_name=s&last_name=a

Chamou o script get1.py para tratar a informações passadas
first_name e last_name.
```

Para testarmos este método digitaremos dois arquivos:

1o. Formulário em HTML: armazenaremos ele no arquivo INDEX.HTML e no diretório c:\Apache2.2\htdocs, automaticamente quando digitarmos localhost o formulário aparecerá.

```
<form action="/cgi-bin/get1.py" method="get">
Nome: <input type="text" name="nome"><br />
Sobrenome: <input type="text" name="sobrenome" />

<input type="submit" value="Enviar" />
```

```
</form>
```

A primeira linha passa os parâmetros para nosso programa CGI. A penúltima linha gera um botão que quando acionado envia os dados.

2o. Programa CGI Python: armazenaremos o nosso arquivo GET1.PY no diretório c:\Apache2.2\cgi-bin, ele é quem responderá ao método POST.

```
#!/python27/python

# Importa modulos para trabalhar com CGI
import cgi, cgitb

# Cria a instancia de armazenamento de campos
form = cgi.FieldStorage()

# Get data from fields
nome = form.getvalue('nome')
sobrenome  = form.getvalue('sobrenome')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Ola' - Segundo Programa CGI </title>"
print "</head>"
print "<body>"
print "<h2>Ola' %s %s</h2>" % (nome, sobrenome)
print "</body>"
print "</html>"
```

O programa recebe os dados através do método getvalue.

Teste o programa e verifique ele sendo executado automaticamente assim que o botão Enviar for clicado.

Método POST: é um método mais seguro de passagem de informação para programas CGI. Esta informação em vez de ser passada em uma string após o sinal de ? , ela é enviada como uma mensagem separada, um script CGI.

```
<form action="/cgi-bin/get1.py" method="post">
Nome     : <input type="text" name="nome"><br />
Sobrenome: <input type="text" name="sobrenome" />

<input type="submit" value="Enviar" />
</form>
```

Veja só o método foi substituído.
Teste o programa fazendo apenas essa modificação e veja que não aparece na URL os dados. **Atenção**: deve-se dar um restart no Apache, para ele atualizar o cache.

10.4 Usando CHECKBOX

Usados quando temos a necessidade de selecionar uma ou mais opções. Abaixo temos um exemplo, com o formulário e o programa CGI, que devem ser executados da mesma forma que o exemplo anterior.

Formulário HTML:

```
<form action="/cgi-bin/checkbox.py" method="POST"
target="_blank">
<input type="checkbox" name="Azul" value="on" /> Azul
<input type="checkbox" name="Vermelho" value="on" /> Vermelho
<input type="submit" value="Selecione a Cor" />
</form>
```

Programa CGI:

```
#!\python27\python

# Importa modulos de manipulacao do CGI
import cgi, cgitb

# Cria instancia de armazenamento dos campos
form = cgi.FieldStorage()

# Pega dado dos campos
if form.getvalue('Azul'):
   azul_flag = "Ligado"
else:
   azul_flag = "Desligado"

if form.getvalue('Vermelho'):
   vermelho_flag = "Ligado"
else:
   vermelho_flag = "Desligado"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
```

```
print "<title>Programa Checkbox</title>"
print "</head>"
print "<body>"
print "<h2> CheckBox AZul esta' : %s</h2>" % azul_flag
print "<h2> CheckBox Vermelho esta' : %s</h2>" % vermelho_flag
print "</body>"
print "</html>"
```

10.5 Usando Botão Radio

Utilizamos quando devemos selecionar apenas uma opção entre várias disponível.

Formulário HTML:

```
<form action="/cgi-bin/radio.py" method="post" target="_blank">
<input type="radio" name="escolha" value="azul" /> Azul
<input type="radio" name="escolha" value="vermelho" /> Vermelho
<input type="submit" value="Selecione Cor" />
</form>
```

Programa CGI:

```
#!\python27\python

# Importando modulos para tratar CGI
import cgi, cgitb

# Cria instancia de armazenamento dos campos
form = cgi.FieldStorage()
```

```
# Pega dado dos campos
if form.getvalue('escolha'):
   escolha = form.getvalue('escolha')
else:
   escolha = "Nao escolheu"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Programa Radio CGI</title>"
print "</head>"
print "<body>"
print "<h2> A cor selecionada e' %s</h2>" % escolha
print "</body>"
print "</html>"
```

Siga as recomendações anteriores para executar o programa.

10.6 Usando texto

Muitas vezes precisamos passar várias linhas de dados para o servidor, o exemplo abaixo mostra como isto é feito.

Formulário: diretório **htdocs** / nome do arquivo **index.html**

```
<form action="/cgi-bin/texto.py" method="post" target="_blank">
<textarea name="textcontent" cols="40" rows="4">
Digite o texto aqui...
</textarea>
<input type="submit" value="Enviar" />
</form>
```

Programa CGI: para executá-lo não se esqueça de armazenar seu programa no diretório do apache **cgi-bin**.

```
#!\python27\python


# Importando modulos para CGI
import cgi, cgitb

# Criando instancia de armazenamento dos campos
form = cgi.FieldStorage()

# Obtendo os dados do campo
if form.getvalue('textcontent'):
   text_content = form.getvalue('textcontent')
else:
   text_content = "Nao digitado"
print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>";
print "<title>Texto CGI </title>"
print "</head>"
print "<body>"
print "<h2> O Conteúdo do texto é: %s</h2>" % text_content
print "</body>"
```

Não esqueça: O formulário de entrada está em **/htdocs** e tem o nome **index.html** e o programa que atende a solicitação está em **/cgi-bin.**

10.7 Usando Drop Downbox

Normalmente usado para selecionar um valor de uma lista com vários valores.

Formulário:

```
<form action="/cgi-bin/dropdown.py" method="post" target="_blank">
<select name="dropdown">
<option value="Matemática" selected>Matemática</option>
<option value="Física" >Física</option>
</select>
<input type="submit" value="Enviar"/>
</form>
```

Programa CGI:

```
#!\python27\python
# -*- coding: cp1252 -*-

# Importando modulos para CGI
import cgi, cgitb

# Cria instancia de armazenamento de campos
form = cgi.FieldStorage()

# Obtem dados
if form.getvalue('dropdown'):
   escolha = form.getvalue('dropdown')
else:
   escolha = "Não definida"
```

```
print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Programa Dropdown Box </title>"
print "</head>"
print "<body>"
print "<h2> O selecionado e' %s</h2>" % escolha
print "</body>"
print "</html>"
```

10.8 Usando o módulo cgitab

Este módulo é um debugger e tornar-se-á útil quando iniciarmos um grande script com muitas linhas, já que este código permite-nos visualizar os erros contidos no script.

Teste este código e veja o que acontece, depois retire os caracteres # das linhas 3 e 4 e veja o erro aparecer no browser. Para executar este programa armazene ele no diretório cgi-bin e digite no browser: http://localhost/cgi-bin/testadorcgi.py

```
#!/python27/python
import cgi
#import cgitb
#cgitb.enable()
print "Content-type: text/html"
print
form = cgi.FieldStorage()
laptops = form.getvalue('laptops','0')
desktops = form.getvalue('desktops','0')
```

```
print """
<html>
<body>
<form action='second.py'>
Quantos laptops voce tem? 
<input type='radio' checked name='laptops' value='0'
/>0 
<input type='radio' name='laptops' value='1' />1 
<input type='radio' name='laptops' value='2' />2
<p>
Quantos desktop voce tem? 
<input type='radio' checked name='desktops' value='0'
/>0 
<input type='radio' name='desktops' value='1' />1 
<input type='radio' name='desktops' value='2' />2
<p>
<input type='submit' />
<p>
Voce tem %d computadores.
</form>
</body>
</html>""" % (int(laptops)+(desktops))
```

A informação abaixo aparece no browser e você consegue visualizar o erro.

```
Traceback (most recent call last):
      File "C:/Arquivos de programas/Apache Software
Foundation/Apache2.2/cgi-bin/testadorcg1", line 29, in
<module>
   </html>""" % (int(laptops)+(desktops))
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Agora corrija o erro e veja o que acontece.

```
#!/python27/python
import cgi
import cgitb
cgitb.enable()
print "Content-type: text/html"
print
form = cgi.FieldStorage()
laptops = form.getvalue('laptops','0')
desktops = form.getvalue('desktops','0')
print """
<html>
<body>
<form action='second.py'>
Quantos laptops voce tem? 
<input type='radio' checked name='laptops' value='0'
/>0 
<input type='radio' name='laptops' value='1' />1 
<input type='radio' name='laptops' value='2' />2
<p>
Quantos desktop voce tem? 
<input type='radio' checked name='desktops' value='0'
/>0 
<input type='radio' name='desktops' value='1' />1 
<input type='radio' name='desktops' value='2' />2
<p>
<input type='submit' />
```

```
<p>
Voce tem %d computadores.
</form>
</body>
</html>""" % (int(laptops)+int(desktops))
```

10.9 Usando arquivos de texto para armazenar informações

Execute no navegador: localhost/cgi-bin/contador.py

```
#!\python27\python
# -*- coding:cp1252 -*-
######### CGI header ##############################
import cgi
print "Content-Type: text/html\n"

try:counter = open("counter.txt","r")
except:counter = open("counter.txt","a")

line = counter.readline()
counter.close()

if line == "":
   number = 1
else:
   number = int(line) + 1

counter = open("counter.txt","w")
counter.write(str(number))
counter.close()

########## HTML content ############################
```

```
print """
<html> <head> <title>Some Title </title> </head>
<body> <h4>Seu Conteúdo ...</h4>
"""
print "Visitante número: ", number

########## HTML footer ##############################
print """
</body></html>
"""
```

10.10 Usando Cookies em programação CGI

Quando trabalhamos com o HTTP, um protocolo de estado, e precisamos manter a informação através de muitas páginas visitadas pelo cliente como num site de compras, a melhor e mais eficiente forma de manter estas informações são através de cookies. Os cookies funcionam da seguinte forma: ao acessarmos uma página no servidor ele envia uma mensagem para o browser, chamada cookie, ela pode ser armazenada ou não, depende da configuração do browser, mas uma vez armazenada ela fica no disco rígido do visitante no formato de registro tipo texto. Agora para recuperar estas informações basta acessar estes registros e pronto temos as informações necessárias.

Os cookies são formados por 5 campos de texto:

Expira: data em que expira o cookie. Em branco ele expira quando o visitante sair do browser.

Domínio: nome de domínio de seu site.

Path: Caminho para o diretório ou página da web que armazenou a informação. Em branco se você quer recuperar o cookie de qualquer diretório ou página.

Seguro: este campo contém a palavra “secure” então o cookie somente pode ser recuperado em um servidor seguro. Em branco não tem restrição.
Nome e Valor: são armazenados como um par de nome e valor.

Setando Cookies: para enviar cookies para o browser. Os cookies serão enviados junto com o cabeçalho HTTP antes do Content -Type. Se você quiser identificar o usuário e a sua senha basta enviar o seguinte texto:

```
#!\python27\python

print "Set-Cookie:UserID=XYZ;"
print "Set-Cookie:Password=XYZ123;"
print "Set-Cookie:Expires=Tuesday, 31-Dec-2011 23:12:40 GMT";"
print "Set-Cookie:Domain=www.capotasprudentina.com.br;"
print "Set-Cookie:Path=/perl;"
print "Content-type:text/html\r\n\r\n"
.....O resto do conteúdo HTML.....
```

Os atributos Expires, Domain e Path são opcionais.

Nosso exemplo:

```
#!\python27\python.exe
print "set-cookie:UserID=sergio;"
print "set-cookie:Password=professor;"
print "Content-type:text/html;"
print ""
print'"
<body>
```

```
   Setando cookies.
<body>
</html>
'''
```

Recuperando Cookies: para recuperar os cookies armazenados é muito simples pois eles estão armazenados no ambiente CGI na variável HTTP_COOKIE e tem o seguinte formato: key1=value1;key2=value2;.....

Programa para recuperar cookies.

```
#!\python27\python.exe

# Import modules for CGI handling
from os import environ
import sys
import string
import cgi, cgitb
cgitb.enable()
aki=''
if environ.has_key('HTTP_COOKIE'):
   cookie=environ.get('HTTP_COOKIE')
   aki=cookie

print '''
Peguei os cookies.
'''
print aki
```

Armazene os programas em cgi-bin, execute o programa para setar os cookies e depois o programa que recupera os cookies, isto deve ser feito através do browser.

10.11 Fazendo download de arquivos

Neste exemplo mostramos como fazer para clicar num link e aparecer uma caixa de diálogo para fazer o download do arquivo.

```
#!\python27\python

# HTTP Header
print "Content-Type:application/octet-stream;
name=\"FileName\"\r\n";
print "Content-Disposition: attachment;
filename=\"FileName\"\r\n\n";

# Abre arquivo a ser enviado
f = open("teste.txt", "rb")

str = f.read();
print str

# Fecha arquivo
f.close()
```

10.12 Outros Programas

```
#!/python27/python
# calendin2.py - calendario dinamico - prototipo 2

print 'Content-type: text/html\n'
try:
        from time import time, localtime
        from calendar import monthcalendar
```

```
from string import join

ano, mes, hoje = localtime(time())[:3]

print '<HTML><TITLE>Calendario Dinamico</TITLE>'
print '<BODY>'
print '<CENTER>'
print '<H1>Calendario de %02d/%04d</H1>' % (mes, ano)
print '<TABLE>'
print '<TR>'
for dia_sem in ['seg','ter','qua','qui','sex','sab','dom']:
        if dia_sem in ['sab','dom']: bgcolor = 'green'
        else: bgcolor = 'blue'
        print '<TH WIDTH="45" BGCOLOR="%s">' % bgcolor
        print '<H3>%s</H3></TH>' % dia_sem
print '</TR>'
for semana in monthcalendar(ano, mes):
        print '<TR>'
        num_dia_sem = 0
        for dia in semana:
                if dia == hoje:
                        bgcolor = 'pink'
                elif num_dia_sem >= 5:
                        bgcolor = 'lightgreen'

                else:
                        bgcolor = 'lightblue'
                print '<TD ALIGN="RIGHT" BGCOLOR="%s">' % bgcolor
                if dia != 0:
                        print '<H2>%2d</H2>' % dia
                        print '</TD>'
        num_dia_sem = num_dia_sem + 1
        print '</TR>'
```

```
        print '</TABLE></CENTER>'

except:
        import sys
        from traceback import print_exc
        sys.stderr = sys.stdout
        print '<HR><H3>Erro no CGI:</H3><PRE>'
        print_exc()
        print '</PRE>'

        print '</BODY></HTML>'
```

Tela do browser:

Referências Bibliográficas

[PIC] – PICÃO, Marcos Elias - **Instalando o Apache+PHP+MYSQL no Windows**, 2007. Disponível em: http://www.guiadohardware.net/tutoriais/apache-php-mysql-windows/. Acessado: 22 novembro 2011.

[RL] – RAMALHO, Luciano, **Aprenda a Programar.** 2004.

[PCP] - Python CGI Programming. Disponível: http://www.tutorialspoint.com/python/python_cgi_programming.htm. Acessado: em 21 novembro 2011.

[FS] – FERG, Steven, **Pensando em Tkinter.** Tradução Josué Labaki , Unesp - Ilha Solteira – 2004.

[LF] – LUNDH, Fredik, **An Introduction to Tkinter.**1999.

[LJC] – LABAKI, Josué, **Introdução a Python Módulo C – Tkinter.** Unesp – Ilha Solteira -2004.

[LJI] – LABAKI, Josué, **Introdução a Python Módulo Básico.** Unesp – Ilha Solteira – 2004.

[PYT] – ROSSUM, Guido van, **Tutorial Python Versão 2.4.2.** 2005. Disponível em: http://www.python.org.br/wiki/DocumentacaoPython. Acessado em: 21 novembro 2011.

[PM] – PILGRIM, Mark, **Dive Into Python**, 2004.

[RLO] – RAMALHO, Luciano, **OO em Python Release 1**,

2008.

[PP] - Python na Prática: Um curso objetivo de Programação Python. Disponível em: http://www.async.com.br/projects/ Acessado em: 21 novembro 2011.