# Python na Prática:
# Um curso objetivo de programação em Python

http://www.async.com.br/projects/pnp/

Christian Robottom Reis
Async Open Source
kiko@async.com.br

Abril de 2004

## Conteúdo

# 1 Introdução

Este tutorial foi criado para apoiar um curso básico da linguagem de programação Python. Em especial, ele tem duas características: primeiro, foi escrito originalmente em português, o que é raro para um livro técnico nesta área. Segundo, ele é indicado para quem já possui alguma experiência em programação, mas que deseja uma noção prática do por que e de como usar Python para programar, sem ter que navegar por horas entre livros e manuais.

Uma observação: este tutorial não é um guia aprofundado, e nem um manual de referência. A documentação disponível em `http://www.python.org/docs/` é excelente (e melhor do que muitos dos livros publicados sobre Python), e serve de referência para os aspectos parti-

culares da linguagem.

**Convenções e Formato** Este tutorial assume Python versão 2; a versão mais recente é a 2.3.3. O texto se baseia em um grande conjunto de exemplos, que aparecem indentados em uma seção distinta, em uma fonte de largura fixa. Os exemplos, de forma geral, descrevem precisamente a saída produzida ao digitar os comandos interativamente. Definições importantes são introduzidas em **negrito**. Nomes de variáveis e expressões de código vêm em uma fonte diferente, `com serifa`.

Quanto ao idioma, os termos em inglês mais comuns são utilizados, com uma explicação prévia do que significam. É importante que os termos em inglês sejam conhecidos, porque são utilizados correntemente na bibliografia e em discussões com outros programadores.

## 1.1 O que é Python?

Python é uma linguagem de programação[1]. Em outras palavras, e sendo simples ao extremo, usamos Python para escrever software. Esta linguagem tem alguns pontos que a tornam especial:

- É uma linguagem **interpretada**.
- Não há pré-declaração de variáveis, e os tipos das variáveis são determinados **dinamicamente**.
- O controle de bloco é feito apenas por **indentação**; não há delimitadores do tipo `BEGIN` e `END` ou { e }.
- Oferece **tipos de alto nível**: strings, listas, tuplas, dicionários, arquivos, classes.
- É **orientada a objetos**; aliás, em Python, tudo é um objeto.

Nas próximas seções estes aspectos são discutidos em detalhes.

### 1.1.1 Linguagem interpretada

Linguagens de programação são freqüentemente classificadas como compiladas ou interpretadas. Nas compiladas, o texto (ou **código-fonte**) do programa é lido por um programa chamado **compilador**, que cria um arquivo binário, executável diretamente pelo hardware da plataforma-alvo. Exemplos deste tipo de linguagem são C ou Fortran. A compilação e execução de um programa simples em C segue algo como:

```
% cc hello.c -o hello
% ./hello
Hello World
```

[1] Python ´e às vezes classifi cado como linguagem de *scripting*, um termo com o qual n˜ao concordo muito, j´a que Python ´e de uso geral, podendo ser usada para criar qualquer tipo de software. O termo *script* geralmente se refere a programas escritos em linguagens interpretadas que automatizam tarefas ou que 'conectam' programas distintos.

onde `cc` é o programa compilador, `hello.c` é o arquivo de código-fonte, e o arquivo criado, `hello`, é um executável binário.

Em contrapartida, programas escritos em linguagens interpretadas não são convertidos em um arquivo executável. Eles são executados utilizando um outro programa, o **interpretador**, que lê o código-fonte e o **interpreta** diretamente, durante a sua execução. Exemplos de linguagem interpretada incluem o BASIC tradicional, Perl e Python. Para executar um programa Python contido no arquivo `hello.py`, por exemplo, utiliza-se algo como:

```
% python hello.py
Hello World
```

Note que o programa que executamos diretamente é o interpretador `python`, fornecendo como parâmetro o arquivo com código-fonte `hello.py`. Não há o passo de geração de executável; o interpretador transforma o programa especificado à medida em que é executado.

### 1.1.2 Tipagem dinâmica

Um dos conceitos básicos em programação é a **variável**, que é uma associação entre um nome e um valor. Ou seja, abaixo, neste fragmento na linguagem C:

```
int a;
a = 1;
```

temos uma variável com o nome `a` sendo declarada, com tipo **inteiro** e contendo o valor 1. Em Python, não precisamos declarar variáveis, nem seus tipos:

```
>>> a = 1
```

seria a instrução equivalente; define uma variável com o valor 1, que é um valor inteiro.

Python possui o que é conhecido como **tipagem dinâmica**: o tipo ao qual a variável está associada pode variar durante a execução do programa. Não quer dizer que não exista tipo específico definido (a chamada **tipagem fraca**): embora em Python não o declaremos explicitamente, as variáveis sempre assumem um único tipo em um determinado momento.

```
>>> a = 1
>>> type(a)          # a função type() retorna o tipo
<type 'int'>        # associado a uma variável
>>> a = "1"
>>> type(a)
<type 'str'>
>>> a = 1.0
>>> type(a)
<type 'float'>
```

Tipagem dinâmica, além de reduzir a quantidade de planejamento prévio (e digitação!) para escrever um programa, é um mecanismo importante para garantir a simplicidade e flexibilidade das **funções** Python. Como os tipos dos argumentos não são explicitamente declarados, não há restrição sobre o que pode ser fornecido como parâmetro. No exemplo acima, são fornecidos argumentos de tipos diferentes à mesma função `type`, que retorna o tipo deste argumento.

### 1.1.3 Controle de bloco por indentação

Na maior parte das linguagens, há instruções ou símbolos específicos que delimitam blocos de código – os blocos que compõem o conteúdo de um laço ou expressão condicional, por exemplo. Em C:

```
if (a < 0) {
    /* bloco de código */
}
```

ou em Fortran:

```
    if (a .lt. 0) then
C       bloco de código
    endif
```

os blocos são delimitados explicitamente — em C por chaves, e em Fortran pelo par `then` e `endif`. Em Python, blocos de código são demarcados apenas por espaços formando uma indentação visual:

```
print "O valor de a é "
if a == 0:
    print "zero"
else:
    print a
```

Esta propriedade faz com que o código seja muito claro e legível — afinal, garante que a indentação esteja sempre correta — porém requer costume e um controle mais formal[2].

### 1.1.4 Tipos de alto nível

Além dos tipos básicos (inteiros, números de ponto flutuante, booleanos), alguns tipos pré-determinados em Python merecem atenção especial:

**Listas:** como um vetor em outras linguagens, a lista é um conjunto (ou **seqüência**) de valores acessados (**indexados**) por um índice numérico, inteiro, começando em zero. A lista em Python pode armazenar valores de qualquer tipo.

```
>>> a = ["A", "B", "C", 0, 1, 2]
>>> print a[0]
A
>>> print a[5]
2
```

[2]Por exemplo, ´e importante convencionar se a indentaç ˜ao ser´a feita por uma tabulaç ˜ao ou por um n´umero determinado de espaç os, j´a que todos editando um mesmo m´odulo Python devem usar o mesmo padr˜ao.

**Tuplas:** tuplas são também seqüências de elementos arbitrários; se comportam como listas com a exceção de que são **imutáveis**: uma vez criadas não podem ser alteradas.

**Strings:** a cadeia de caracteres, uma forma de dado muito comum; a string Python á uma seqüência imutável, alocada dinamicamente, sem restrição de tamanho.

**Dicionários:** dicionários são seqüências que podem utilizar índices de tipos variados, bastando que estes índices sejam imutáveis (números, tuplas e strings, por exemplo). Dicionários são conhecidos em outras linguagens como arrays associativos ou *hashes*.

```
>>> autor = {"nome" : "Christian", "idade": 28}
>>> print autor["nome"]
Christian
>>> print autor["idade"]
28
```

**Arquivo:** Python possui um tipo pré-definido para manipular arquivos; este tipo permite que seu conteúdo seja facilmente lido, alterado e escrito.

**Classes e Instâncias:** classes são estruturas especiais que servem para apoiar programação orientada a objetos; determinam um tipo customizado com dados e operações particulares. Instâncias são as expressões concretas destas classes. Orientação a objetos em Python é descrita em maiores detalhes na seção 4.

### 1.1.5 Orientação a objetos

Orientação a objetos (OO) é uma forma conceitual de estruturar um programa: ao invés de definirmos variáveis e criarmos funções que as manipulam, definimos **objetos** que possuem dados próprios e ações associadas. O programa orientado a objetos é resultado da 'colaboração' entre estes objetos.

Em Python, todos os dados podem ser considerados objetos: qualquer variável — mesmo as dos tipos básicos e pré-definidos — possui um valor e um conjunto de operações que pode ser realizado sobre este. Por exemplo, toda string em Python possui uma operação (ou **método**) chamada `upper`, que gera uma string nova com seu conteúdo em maiúsculas:

```
>>> a = "Hello"
>>> a.upper()
'HELLO'
```

Como a maior parte das linguagens que são consideradas 'orientadas a objeto', Python oferece um tipo especial para definir objetos customizados: a **classe**. Python suporta também funcionalidades comuns na orientação a objetos: herança, herança múltipla, polimorfismo, reflexão e introspecção.

## 1.2 Por que Python?

Dado que existe uma grande diversidade de linguagens diferentes, por que aprender Python é interessante ou mesmo importante? Na minha opinião, a linguagem combina um conjunto único de vantagens:

- Os conceitos fundamentais da linguagem são simples de entender.
- A sintaxe da linguagem é clara e fácil de aprender; o código produzido é normalmente curto e legível.
- Os tipos pré-definidos incluídos em Python são poderosos, e ainda assim simples de usar.
- A linguagem possui um interpretador de comandos interativo que permite aprender e testar rapidamente trechos de código.
- Python é expressivo, com abstrações de alto nível. Na grande maioria dos casos, um programa em Python será muito mais curto que seu correspondente escrito em outra linguagem. Isto também faz com o ciclo de desenvolvimento seja rápido e apresente potencial de defeitos reduzido – menos código, menos oportunidade para errar.
- Existe suporte para uma diversidade grande de bibliotecas externas. Ou seja, pode-se fazer em Python qualquer tipo de programa, mesmo que utilize gráficos, funções matemáticas complexas, ou uma determinada base de dados SQL.
- É possível escrever extensões a Python em C e C++ quando é necessário desempenho máximo, ou quando for desejável fazer interface com alguma ferramenta que possua biblioteca apenas nestas linguagens.
- Python permite que o programa execute inalterado em múltiplas plataformas; em outras palavras, a sua aplicação feita para Linux normalmente funcionará sem problemas em Windows e em outros sistemas onde existir um interpretador Python.
- Python é pouco punitivo: em geral, 'tudo pode' e há poucas restrições arbitrárias. Esta propriedade acaba por tornar prazeroso o aprendizado e uso da linguagem.
- Python é livre: além do interpretador ser distribuído como software livre (e portanto, gratuitamente), pode ser usado para criar qualquer tipo de software — proprietário ou livre. O projeto e implementação da linguagem é discutido aberta e diariamente em uma lista de correio eletrônico, e qualquer um é bem-vindo para propor alterações por meio de um processo simples e pouco burocrático.

Ao longo das próximas seções serão expostos aos poucos os pontos concretos que demonstram estas vantagens.

# 2 Python básico: invocação, tipos, operadores e estruturas

Esta primeira seção aborda os aspectos essenciais da linguagem. Por falta de um lugar melhor, será introduzido aqui o símbolo para comentários em Python: quando aparece o caracter sustenido ( $\#$)[3] no código-fonte, o restante da linha é ignorado.

[3] Tamb´em conhecido como 'n´umero', 'jogo da velha', 'cardinal' e 'lasagna'.

## 2.1 Executando o interpretador Python interativamente

Python está disponível tanto para Windows e Macintosh como para os diversos Unix (incluindo o Linux). Cada plataforma possui um pacote específico para instalação, que normalmente inclui um grande número de bibliotecas de código e um programa executável `python`[4]. Este é o interpretador Python que usaremos para executar nossos programas.

Para iniciar, vamos executar o interpretador **interativamente**, no **modo shell**. Em outras palavras, vamos entrar o código-fonte diretamente, sem criar um arquivo separado. Para este fim, execute o interpretador (digitando apenas '`python`'). Será impresso algo como o seguinte (dependendo de qual sistema estiver usando):

```
% python
Python 2.3.3 (#1, Dec 22 2003, 15:38:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Este símbolo `>>>` chamamos de *prompt*, e indica que o interpretador está aguardando ser digitado um comando. O shell Python oferece uma excelente forma de testar código, e vamos começar com algumas instruções simples.

```
>>> a = 1
>>> print a
1
>>> b = "Hello world"
>>> print b
Hello world
```

Neste exemplo, são introduzidos dois aspectos básicos: primeiro, a atribuição de variáveis; segundo, a instrução `print`, que exibe valores e o conteúdo de variáveis.

A forma interativa de executar o Python é conveniente; no entanto, não armazena o código digitado, servindo apenas para testes e procedimentos simples. Para programas mais complexos, o código-fonte é normalmente escrito e armazenado em um arquivo.

## 2.2 Criando um programa e executando-o

Em Python, um arquivo contendo instruções da linguagem é chamado de **módulo**. Nos casos mais simples pode-se usar um único módulo, executando-o diretamente; no entanto, é interessante saber que é possível sub-dividir o programa em arquivos separados e facilmente integrar as funções definidas neles. Para criar um arquivo contendo um programa, basta usar qualquer editor de texto. Para um primeiro exemplo, crie um arquivo `hello.py` contendo as seguintes linhas:

[4] Normalmente, ´e inclu´ıdo um ambiente de ediç˜ao e execuç˜ao simples, como o IDLE, tamb´em escrito em Python. Estes podem ser tamb´em utilizados.

```
a = "Hello"
b = "world"
print a,b
```

E a seguir, execute-o da seguinte forma:

```
% python hello.py
```

Ou seja, execute o interpretador e passe como parâmetro o nome do arquivo que contém o código a ser executado. O resultado impresso deve ser:

```
Hello world
```

Perceba bem que não há preâmbulo algum no código-fonte; escrevemos diretamente o código a ser executado[5].

Nas próximas seções do tutorial, estaremos utilizando bastante o modo shell; no entanto, para exercícios e exemplos mais extensos que algumas linhas, vale a pena usar módulos para permitir edição e revisão.

## 2.3 Tipos, variáveis e valores

Nomes de variáveis começam sempre com uma letra, não contém espaços, e assim como tudo em Python, são sensíveis a caixa (*case-sensitive*) — em outras palavras, minúsculas e maiúsculas fazem, sim, diferença. Como explicado anteriormente, a variável não precisa ser pré-declarada e seu tipo é determinado dinamicamente.

### 2.3.1 Tipos numéricos

Tipos numéricos representam valores numéricos. Em Python há alguns tipos numéricos pré-definidos: inteiros (**int**), números de ponto flutuante (**float**), booleanos (**bool**) e complexos (**complex**). Estes tipos suportam as operações matemáticas comuns como adição, subtração, multiplicação e divisão, e podem ser convertidos entre si.

> Uma observação: booleanos foram adicionados na versão Python 2.2; sua utilização é normalmente associada a expressões condicionais. Maiores detalhes sobre este tipo estão disponíveis na seção 2.4.4.

A seguir alguns exemplos de criação de variáveis numéricas:

```
>>> a = 1               # valor inteiro
>>> preco = 10.99       # valor ponto flutuante, ou float.
>>> t = True            # valor booleano
>>> i = 4+3j            # valor complexo
```

[5] Em Unix, ´e poss´ıvel criar programas em Python e execut´a-los usando apenas o nome do m´odulo (sem digitar o nome do interpretador `python`). Basta adicionar a linha 'm´agica' `#!/usr/bin/env python` ao in´ıcio do m´odulo e dar permiss˜ao de execuç˜ao ao arquivo. Note que isto n˜ao evita que seja necess´ario que o interpretador esteja instalado no sistema; apenas permite que se possa omitir digitar o seu nome.

Valores inteiros podem também ser fornecidos em base octal e hexadecimal:

```
>>> a = 071
>>> print a
57
>>> a = 0xFF
>>> print a
255
```

Para ser considerado um **float**, o número deve possuir um ponto e uma casa decimal, mesmo que seja zero. O fato de ser considerado um float é importante para a operação divisão, pois dependendo do tipo dos operandos, a divisão é inteira ou em ponto flutuante.

```
>>> 5 / 2     # divisão inteira, resultado inteiro
2
>>> 5 / 2.0   # divisão em ponto flutuante
2.5
>>> 5 * 2.13
10.649999999999999
```

> Atenção especial para este terceiro exemplo, que demonstra uma particularidade da representação e impressão de números de ponto flutuante. Números deste tipo, como o valor `2.13` do exemplo, possuem uma representação interna particular devido à natureza dos intrínseca dos computadores digitais; operações realizadas sobre eles têm precisão limitada, e por este motivo o resultado impresso diverge ligeiramente do resultado aritmeticamente correto. Na prática, o resultado obtido é praticamente equivalente a 10.65[6].

**Determinando o tipo de uma variável** Para descobrir o tipo atual de uma variável, pode-se usar a função `type()`:

```
>>> a = 1
>>> print type(a)
<type 'int'>
>>> a = "hum"
>>> print type(a)
<type 'string'>
```

Uma das vantagens de Python são os tipos complexos que vêm pré-definidos, introduzidos na seção 1.1.2. As seções seguintes cobrem estes tipos.

[6]Todas as linguagens de programaç ˜ao convencionais utilizam este mesmo mecanismo de representaç ˜ao, e na pr´atica n˜ao resultam grandes problemas com isso, mas Python deixa transparecer esta particularidade em algumas situaç ˜oes, como esta. O tutorial Python em `http://www.python.org/docs/current/tut/` inclui uma seç ˜ao sobre as nuances do uso e representaç ˜ao de n´umeros de ponto flıtuante.

### 2.3.2 Listas

A lista é uma **seqüência**: um conjunto linear (como um vetor em outras linguagens) de valores indexados por um número inteiro. Os índices são iniciados em **zero** e atribuídos seqüencialmente a partir deste. A lista pode conter quaisquer valores, incluindo valores de tipos mistos, e até outras listas. Para criar uma lista, usamos colchetes e vírgulas para enumerar os valores:

```
>>> numeros = [1, 2, 3]
>>> opcoes  = ["nao", "sim", "talvez"]
>>> modelos = [3.1, 3.11, 95, 98, 2000, "Millenium", "XP"]
>>> listas  = [numeros, opcoes]
```

Neste exemplo criamos quatro listas diferentes, com elementos de tipos diversos. A quarta lista tem como elementos outras listas: em outras palavras, é uma lista de listas:

```
>>> print listas
[[1, 2, 3], ['nao', 'sim', 'talvez']]
```

Para acessar um elemento específico de uma lista, usamos o nome da lista seguido do índice entre colchetes:

```
>>> print numeros[0]
1
>>> print opcoes[2]
talvez
>>> print modelos[4]
2000
```

Usando índices negativos, as posições são acessadas a partir do final da lista, -1 indicando o último elemento:

```
>>> print numeros[-1]
3
>>> print modelos[-2]
Millenium
```

Python oferece um mecanismo para criar 'fatias' de uma lista, ou **slices**. Um slice é uma lista gerada a partir de um fragmento de outra lista. O fragmento é especificado com dois índices, separados por dois pontos; o slice resultante contém os elementos cujas posições vão do primeiro índice ao segundo, não incluindo o último elemento. Se omitirmos um dos índices no slice, assume-se início ou fim da lista.

```
>>> print numeros[:2]
[1, 2]
>>> print opcoes[1:]
['sim', 'talvez']
>>> print modelos[1:5]
[3.1000000000000001, 95, 98, 2000]
```

Note que o terceiro exemplo demonstra a mesma particularidade referente à representação de números de ponto flutuante, descrita na seção 2.3.1: ao imprimir a lista, Python exibe *a representação interna* dos seus elementos, e portanto, floats são impressos com precisão completa.

**Métodos da Lista** Na seção 1.1.5 introduzimos um conceito central da linguagem: 'tudo é um objeto´; uma lista em Python, sendo um objeto, possui um conjunto de operações próprias que manipulam seu conteúdo. O nome **método** é usado para descrever operações de um objeto; métodos são acessados no código-fonte digitando-se um ponto após o nome da variável, e a seguir o nome do método.

Para executar (ou **chamar**) um método, usamos parênteses após seu nome, fornecendo argumentos conforme necessário. Abaixo são exemplificados dois exemplos de chamadas de métodos em listas:

```
>>> numeros.append(0)
>>> print numeros
[1, 2, 3, 0]
>>> numeros.sort()
>>> print numeros
[0, 1, 2, 3]
```

Observe com atenção este exemplo, que demonstra os conceitos fundamentais descritos nos parágrafos anteriores:

- O método `append(v)` recebe como argumento um valor, e adiciona este valor ao final da lista.
- O método `sort()` ordena a lista, modificando-a. Não recebe argumentos.

A lista possui uma série de outros métodos; consulte a referência Python para uma descrição completa (e veja também a função `dir()` na seção 4.4).

### 2.3.3 Tuplas

A tupla é uma seqüência, como a lista: armazena um conjunto de elementos acessíveis por um índice inteiro. A tupla é imutável; uma vez criada, não pode ser modificada. Para criar uma tupla use parênteses, e vírgulas para separar seus elementos:

```
>>> t = (1, 3, 5, 7)
>>> print t[2]
5
```

A tupla é utilizada em algumas situações importantes: como a lista de argumentos de uma função ou método, como chave em dicionários, e em outros casos onde fizer sentido ou for mais eficiente um conjunto fixo de valores.

### 2.3.4 Strings

A string, como citado anteriormente, é uma seqüência imutável com um propósito especial: armazenar cadeias de caracteres.

```
>>> a = "Mondo Bizarro"
>>> print a
Mondo Bizarro
```

Strings podem ser delimitadas tanto com aspas simples quanto duplas; se delimitamos com aspas duplas, podemos usar as aspas simples como parte literal da string, e vice-versa. Para inserir na string aspas literais do mesmo tipo que o delimitador escolhido, prefixe-as com uma contra-barra \. As atribuições abaixo são equivalentes:

```
>>> b = "All's quiet on the eastern front."
>>> c = 'All\'s quiet on the eastern front.'
>>> b == c
True
```

São usados caracteres especiais para denotar quebra de linha (`\n`), tabulação (`\t`) e outros.

```
>>> a = "Hoje\n\t é o primeiro dia."
>>> print a
Hoje
    é o primeiro dia.
```

Para criar uma string com múltiplas linhas, é útil o delimitador aspas triplas: as linhas podem ser quebradas diretamente, e a string pode ser finalizada com outras três aspas consecutivas:

```
a = """I wear my sunglasses at night
So I can so I can
Keep track of the visions in my eyes"""
```

Finalmente, como toda seqüência, a string pode ser indexada ou dividida em slices, usando o operador colchetes:

```
>>> a = "Anticonstitucionalissimamente"
>>> print a[0]
A
>>> print a[13]
i
>>> print a[:4]
Anti
>>> print a[-5:-1]
ment
```

A string possui um operador especial, a porcentagem (%), que será descrito na seção 2.4.6. Possui ainda um grande número de métodos, descritos em detalhes na seção *String Methods* do manual de referência Python.

### 2.3.5 Dicionários

Um dicionário representa uma coleção de elementos onde é possível utilizar um índice de qualquer tipo imutável, ao contrário da lista, onde índices são sempre inteiros seqüencialmente atribuídos. É costumeiro usar os termos chave e valor (key/value) para descrever os elementos de um dicionário - a chave é o índice, e o valor, a informação correspondente àquela chave.

Para declarar dicionários, utilizamos o símbolo chaves, separando o índice do valor por dois pontos e separando os pares índice-valor por vírgulas:

```
>>> refeicoes = {"café" : "café", "almoço" : "macarrão",
...              "jantar" : "sopa"}
>>> print refeicoes["almoço"]
macarrao

>>> precos_modelos = {98 : 89, 99 : 119, 2000 : 199}
>>> print precos_modelos[98]
89
```

Neste exemplo criamos dois dicionários com três elementos cada um. As chaves do dicionário `refeicoes` são as strings ”café”, ”almoço”e ”jantar”, e os valores respectivos, as strings ”café”, ”macarrão”e ”sopa”.

**Métodos do Dicionário** O dicionário também possui alguns métodos notáveis:

- `keys()` retorna uma lista (sim, exatamente, do tipo lista) com as chaves do dicionário;
- `values()` retorna uma lista com os valores do dicionário;
- `items()` retorna uma lista de tuplas com o conteúdo do dicionário, cada tupla contendo um par (chave, valor);

```
>>> precos_modelos.keys()
[99, 98, 2000]
>>> precos_modelos.values()
[119, 89, 199]
    # A ordem dos elementos retornados por keys() e
    # values() é arbitrária; não confie nela.
```

- `has_key(k)` verifica se a lista possui aquela chave:

```
>>> precos_modelos.has_key(98)
True
>>> precos_modelos.has_key(97)
False
```

- `update(d2)` atualiza o dicionário com base em um segundo dicionário fornecido como parâmetro; elementos do dicionário original que também existem no segundo são atualizados; elementos que existem no segundo mas que não existem no original são adicionados a este.

```
>>> precos_modelos.update({2000 : 600, 2001: 700})
>>> print precos_modelos
{99: 400, 98: 300, 2001: 700, 2000: 600}
```

No exemplo acima, observe que a chave 2000 foi atualizada, e 2001, acrescentada.

É possível usar o dicionário como uma estrutura de dados simples, com campos para cada informação a ser armazenada. Por exemplo, para armazenar informação sobre um produto hipotético, com código, descrição e preço:

```
>>> produto = {"código":771, "desc":"Copo", "preço":10.22}
>>> print produto["código"]
771
>>> print produto["desc"]
Copo
```

É um padrão comum criar listas de dicionários neste formato, cada ítem na lista correspondendo a um produto em particular.

## 2.4 Operadores

O próximo tópico essencial da linguagem são **operadores**, símbolos que operam sobre variáveis e valores.

### 2.4.1 Operadores aritméticos

A maior parte dos operadores aritméticos em Python funciona de maneira intuitiva e análoga aos operadores em outras linguagens. Demonstrando por exemplo:

```
>>> print a + 3       # adição
10
>>> print a - 2       # subtração
5
>>> print a / 2       # divisão inteira: argumentos inteiros
3                     # e resultado inteiro

>>> print a / 2.5     # divisão em ponto flutuante: pelo
2.8                   # menos um argumento deve ser float

>>> print a % 4       # resto da divisão inteira
3
>>> print a * 2       # multiplicação
14
>>> print a ** 2      # exponenciação
49
```

A exponenciação também pode ser feita por meio de uma função, `pow()`, como descrito na seção 3. A raiz quadrada e outras funções matemáticas estão implementadas no módulo `math`; veja a seção 5 para maiores detalhes.

Com exceção da exponenciação e da divisão inteira, estes operadores são bastante comuns em linguagens de programação. Os operadores aritméticos podem ser usados em floats também:

```
>>> a = 1.15
>>> print a / a - a * a + a
0.57349375
```

e os operadores de adição (`+`) e multiplicação (`*`), em strings:

```
>>> a = "exato"
>>> print a * 2
exatoexato
>>> print "quase " + a
quase exato
```

, listas:

```
>>> a = [-1, 0]
>>> b = [1, 2, 3]
>>> print b * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print a + b
[-1, 0, 1, 2, 3]
```

e tuplas:

```
>>> a = (1, 2)
>>> print a + (2, 1)
(1, 2, 2, 1)
```

Como exemplificado acima, o operador adição (e multiplicação) serve para concatenar listas, tuplas e strings. Não pode ser utilizado com dicionários (que podem ser atualizados usando a função `update()`, mas para os quais a operação de concatenação não faria sentido).

### 2.4.2 Operadores sobre cadeias de bits

Para inteiros, existem operadores que manipulam seus valores como cadeias de bits (operadores *bit-wise*) de acordo com a aritmética booleana:

```
>>> a = 0xa1
>>> b = 0x01
>>> print a, b  # para imprimir os valores decimais
161, 1
```

```
>>> a & b       # and
1
>>> a | b       # or
161
>>> a << 1      # shift para esquerda
322
>>> b >> 1      # shift para direita
0
>>> ~a          # inversão em complemento de 2
-162
```

Note que a representação dos valores inteiros, por padrão, é decimal: aqui, embora fornecemos os valores em base hexadecimal, os resultados aparecem na base 10.

### 2.4.3 Operadores de atribuição

O operador mais simples, e que já fui utilizado em diversos exemplos anteriores, é o operador de atribuição. Este operador é representado por um único símbolo de igualdade, =, definindo uma variável e automaticamente atribuindo a ela um valor. O exemplo abaixo define uma variável a, com valor inteiro 1.

```
>>> a = 1
```

Existem formas variantes deste operador que o combinam com os operadores aritméticos e bit-wise introduzidos nas seções anteriores. Estas formas foram introduzidos na versão Python 2.0, e existem primariamente para oferecer uma maneira conveniente de re-atribuir um valor transformado a uma variável.

A sintaxe para o operador combinado utiliza o símbolo do operador aritmético/bit-wise relevante, seguido da igualdade. Este operador combinado efetua a operação sobre o valor da variável, já atribuindo o resultado a esta mesma. Exemplificando:

```
>>> a = 1
>>> a += 1
>>> print a
2
>>> a /= 2
>>> print a
1
>>> a *= 10
>>> print a
10
```

### 2.4.4 Operadores condicionais

Tradicionalmente, programação envolve testar valores (e tomar decisões com base no resultado do teste). Um teste é essencialmente uma expressão condicional que tem um resultado verdadeiro ou falso. Esta seção descreve os operadores condicionais mais comuns em Python.

A partir da versão 2.2 de Python, existe um tipo numérico que representa o resultado de um teste condicional: o tipo booleano. É um tipo bastante simples, possuindo apenas dois valores possíveis: `True` e `False`. A partir da versão 2.3 de Python, qualquer comparação retorna um valor booleano; versões anteriores retornavam 1 ou 0, respectivamente. Na prática, o resultado é equivalente, ficando apenas mais explícito e legível o resultado.

**Igualdade**

```
>>> print 2 == 4            # igualdade
False
>>> print 2 != 4            # diferente de
True
>>> print "a" == "a"
True
>>> print "a" != "b"
True
```

**Comparação**

```
>>> print 1 < 2             # menor que
True
>>> print 3 > 5             # maior que
False
>>> print 3 >= 4            # maior ou igual que
False
```

Comparações podem ser realizadas também com outros tipos de variáveis; para strings, listas ou dicionários, as comparações 'maior que' e 'menor que' se referem ao número e ao valor dos elementos, mas por não serem terrivelmente claras não são muito freqüentemente usadas[7].

```
>>> [2,2,3] < [2,3,2]
True
>>> "abacate" < "pera"
True
```

**Dica:** quando comparando o número de elementos de uma seqüência ou dicionário, é normalmente utilizada a função `len()`, que retorna o 'comprimento' da seqüência.

**Presença em Seqüências** Para seqüências e dicionários, existe o operador `in`, que verifica se o elemento está contido no conjunto; no caso do dicionário, o operador testa a presença do valor na seqüência de chaves.

```
>>> "x" in "cha"
False
>>> 1 in [1,2,3]
True
```

[7] A exceç ˜ao ´obvia ´e na ordenaç ˜ao de seq¨uˆencias contendo elementos destes tipos.

### 2.4.5 Operadores lógicos

Os operadores lógicos `not`, `and` e `or` permitem modificar e agrupar o resultado de testes condicionais:

```
>>> nome = "pedro"
>>> idade = 24
>>> nome == "pedro" and idade == 25
False
>>> nome == "pedro" and idade < 25
True
>>> len(nome) < 10 or not nome == "pedro"
False
```

Estes operadores são utilizados com freqüência nas estruturas de controle descritas na seção 2.5.

**Combinação de operadores** Python oferece uma forma implícita de combinar operações condicionais, sem o uso de operadores lógicos. Por exemplo, para verificar se um valor está entre dois extremos, pode-se usar a seguinte sintaxe:

```
if 0 < a < 10:
    print "Entre zero e dez"
```

Podem também ser comparados diversos valores simultaneamente:

```
if a == b == c:
    print "São idênticos"
```

e mesmo combinados operadores comparativos com operadores de igualdade:

```
if a == b <= c:
    print "São idênticos"
```

Esta expressão verifica se `a` é igual a `b` e além disso, se `b` é menor ou igual a `c`.

### 2.4.6 Substituição em strings: o operador `%`

Uma operação muito útil para processamento de texto é a substituição de símbolos em strings. É particularmente adequada para gerarmos strings formatadas contendo algum valor variável, como o clássico formulário: ”Nome: _____ Idade: _____ anos”.

1. Escreve-se a string normalmente, usando um símbolo especial no lugar da lacuna:
   - `%d`: para substituir inteiros
   - `%f`: para substituir floats

- `%s`: para substituir outra string

```
>>> a = "Total de itens: %d"
>>> b = "Custo: %f"
```

2. Para efetuar a substituição, aplica-se um operador `%` sobre a string contendo o símbolo de formatação, seguido do valor ou variável a substituir:

   ```
   >>> print a % 10
   Total de itens: 10
   ```

   Como pode ser visto, o símbolo é substituído pelo valor fornecido. Podem ser utilizados tanto valores explícitos quanto variáveis para a substituição:

   ```
   >>> custo = 5.50
   >>> print b % custo
   Custo: 5.500000
   ```

   Caso sejam múltiplos valores a substituir, use uma tupla:

   ```
   >>> print "Cliente: %s, Valor %f" % ("hungry.com", 40.30)
   Fornecedor: hungry.com, Custo 40.300000
   ```

Este operador permite ainda utilizar um número junto ao símbolo porcentagem para reservar um tamanho total à string:

```
>>> a = "Quantidade: %4d"
>>> print a % 3
>>> print a % 53
>>> print a % 120
Quantidade:    3
Quantidade:   53
Quantidade:  120
```

É possível controlar a formatação de tipos numéricos de maneira especial através de modificadores nos símbolos no formato **`m.n`**. Como acima, **`m`** indica o total de caracteres reservados. Para floats, **`n`** indica o número de casas decimais; para inteiros, indica o tamanho total do número, preenchido com zeros à esquerda. Ambos os modificadores podem ser omitidos.

```
>>> e = 2.7313
>>> p = 3.1415
>>> sete = 7
>>> print "Euler: %.7f" % e        # 7 casas decimais
Euler: 2.7313000
>>> print "Pi: %10.3f" % p         # 10 espaços, 3 casas
Pi:      3.142
>>> print "Sete: %10.3d" % sete    # 10 espaços, 3 dígitos
Sete:        007                  # (é um inteiro)
```

## 2.5 Estruturas de controle

Toda linguagem de programação possui instruções que controlam o fluxo de execução; em Python, há um conjunto pequeno e poderoso de instruções, descritas nas seções a seguir.

### 2.5.1 Condicional: a instrução `if`

A instrução condicional básica de Python é o `if`. A sintaxe é descrita a seguir (lembrando que a indentação é que delimita o bloco):

```
if condição:
    # bloco de código
elif condição:
    # outro bloco
else:
    # bloco final
```

As condições acima são comparações feitas utilizando os operadores condicionais descritos na seção 2.4.4, possivelmente combinados por meio dos operadores lógicos descritos na seção 2.4.5.

```
a = 2
b = 12
if a < 5 and b * a > 0:
    print "ok"
```

A instrução elif permite que se inclua uma exceção condicional — algo como "... **senão** se isso ...". O else é uma exceção absoluta[8].

```
if nome == "pedro":
    idade = 21
elif nome == "josé":
    idade = 83
else:
    idade = 0
```

Provendo `nome="álvaro"` para o bloco acima, será atribuído o valor zero a `idade`.

### 2.5.2 Laço iterativo: `for`

Há apenas dois tipos de laços em Python: `for` e `while`. O primeiro tipo, mais freqüentemente utilizado, percorre uma seqüência em ordem, a cada ciclo substituindo a variável especificada por um dos elementos. Por exemplo:

[8] N˜ao h´a nenhuma estrutura do tipo `switch/case`, mas ´e poss´ı vel simul´a-la usando uma express˜ao `if` com um `elif` para cada caso diferente.

```
>>> jan_ken_pon = ["pedra", "papel", "cenoura"]
>>> for item in jan_ken_pon:
...   print item
...
pedra
papel
cenoura
```

A cada iteração, `item` recebe o valor de um elemento da seqüência. Para efetuar uma laço com um número fixo de iterações, costuma-se usar o `for` em conjunto com a função `range`, que gera seqüências de números:

```
>>> for i in range(1,4):
...   print "%da volta" % i
...
1a volta
2a volta
3a volta
```

**Iteração em dicionários** Para iterar em dicionários, podemos usar as funções `keys()` ou `values()` para gerar uma lista:

```
>>> dict = {"batata": 500, "abóbora": 1200,
...         "cebola": 800}
>>> for e in dict.keys():
...   print "Item: %8s Peso: %8s" % (e, dict[e])
Item: cebola  Peso:   800
Item: batata  Peso:   500
Item: abóbora Peso:  1200
```

Note que porque o dicionário em si não possui um conceito de ordem, a lista que resulta do método `keys()` possui ordenação arbitrária. Por este motivo, no exemplo acima, o laço não segue a declaração do dicionário.

**Controle adicional em laços** Para ambos os tipos de laço, existem duas instruções de controle adicional, `continue` e `break`. A primeira reinicia uma nova iteração imediatamente, interrompendo a iteração atual; a segunda faz o laço terminar imediatamente.

A forma geral do laço `for` é:

```
for variável in seqüência:
    # bloco de código
else:
    # bloco executado na ausência de um break
```

Note a presença da cláusula `else`. Esta cláusula é executada quando a saída do laço *não for determinada* por uma instrução `break`. Um exemplo clarifica este mecanismo:

```
valores = [2, 4, 5, 2, -1]
for i in valores:
    if i < 0:
        print "Negativo encontrado: %d" % i
        break
else:
    print "Nenhum negativo encontrado"
```

### 2.5.3 Laço condicional: `while`

O segundo tipo de laço, `while`, é utilizado quando necessitamos fazer um teste a cada iteração do laço.

```
while condição:
    # bloco de código
else:
    # bloco executado na ausência de um break
```

Como o laço for, o while possui uma cláusula `else`. Um exemplo do uso de while segue:

```
>>> m = 3 * 19
>>> n = 5 * 13
>>> contador = 0
>>> while m < n:
...   m = n / 0.5
...   n = m / 0.5
...   contador = contador + 1
...
>>> print "Iteramos %d vezes." % contador
Iteramos 510 vezes.
```

Não há uma instrução especial para efetuar um laço com teste ao final da iteração (como o laço `do ... while()` em `C`), mas pode-se usar um while infinito — usando uma condição verdadeira, fixa — em combinação com um teste e `break` internos:

```
>>> l = ["a", "b", "c"]
>>> i = len(l) - 1
>>> while True:
...   print l[i]
...   i = i - 1
...   if i < 0:
...     break
...
c
b
a
```

**Veracidade e falsidade de condições** As estruturas `if` e `while` utilizam condições lógicas para controle, avaliando-as de maneira booleana. Como qualquer valor ou expressão pode ser usado como condição, é importante entender qual o mecanismo de avaliação da linguagem.

Em Python, falso é denotado:

- pelo booleano False,
- pelo valor 0 (zero),
- pela lista, dicionário, tupla ou string vazios, de tamanho zero,
- pelo valor especial None, que significa nulo.

Qualquer outro valor simples é considerado verdadeiro. Instâncias podem definir regras mais complexas para avaliação; para detalhes consulte a seção *Basic customization* do manual de referência Python.

### 2.5.4 Exceções

Com os dois tipos de laços descritos na seção anterior, todas as necessidades 'normais' de controle de um programa podem ser implementadas. No entanto, quando algo inesperado ocorre, ou uma condição de erro conhecido é atingida, Python oferece uma forma adicional de controlar o fluxo de execução: a exceção.

A exceção é um recurso de linguagens de programação modernas que serve para informar que uma condição incomum ocorreu. Embora existam outras aplicações, em geral comunicam-se através de exceções erros ou problemas que ocorrem durante a execução de um programa.

Exceções são internamente geradas pelo interpretador Python quando situações excepcionais ocorrem. No exemplo abaixo,

```
>>> a = [1, 2, 3]
>>> print a[5]
```

o código interno do interpretador sabe que não podemos acessar uma lista através um índice não-existente, e gera uma exceção:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Vamos analisar a mensagem exibida. A primeira linha anuncia um **traceback**, que é a forma como é exibida a pilha de execução que gerou a exceção. A segunda indica a linha de código na qual ocorreu o problema, e o arquivo. Como estamos usando o modo interativo neste exemplo, o arquivo aparece como `<stdin>`, que é a entrada padrão. A terceira linha indica o tipo de exceção levantada — neste caso, `IndexError` — e informa uma mensagem que descreve mais especificamente o problema.

**Tratando exceções** A exceção normalmente imprime um traceback e interrompe a execução do programa. Uma ação comum é testar e controlar a exceção; para este fim, usamos uma cláusula `try`/`except`:

```
>>> a = [1, 2, 3]
>>> try:
...   print a[5]
... except IndexError:
...   print "O vetor nao possui tantas posições!"
O vetor nao possui tantas posições!
```

A instrução `try` captura exceções que ocorrerem no seu bloco de código; a linha `except` determina quais tipos de exceção serão capturados. A sintaxe da cláusula except segue os formatos:

```
except tipo_da_excecao [, variavel_da_excecao]:
    # bloco de código
```

ou

```
except (tipo_excecao_1, tipo_excecao_2, ...)
       [, variavel_da_excecao]:
    # bloco de código
```

O primeiro elemento da cláusula `except` é um tipo da exceção, ou uma tupla de tipos caso múltiplas exceções devam ser tratadas da mesma forma. O segundo elemento é opcional; permite que seja capturada uma instância que armazena informações da exceção. Um uso trivial desta instância é imprimir a mensagem de erro:

```
>>> a = "foo"
>>> print a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot add type "int" to string
```

Podemos capturar e tratar de forma especial o erro acima, imprimindo a mensagem fornecida e continuando a execução normalmente.

```
>>> try:
...   print a + 1
... except TypeError, e:
...   print "Um erro ocorreu: %s" % e
...
Um erro ocorreu: cannot add type "int" to string
>>>
```

Diversos tipos de exceções vêm pré-definidas pelo interpretador Python; o guia de referência 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.

## 2.6 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, descritas na seção 3; a seção atual detalha as formas de definir funções em código Python.

### 2.6.1 Sintaxe básica

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
```

O código da função, uma vez avaliado pelo interpretador, define um nome local. Os argumentos são valores fornecidos quando chamada a função, e que ficam disponíveis por meio de variáveis locais no corpo da função.

No exemplo abaixo, temos uma função que recebe uma lista de dicionários como parâmetro, e uma função que recebe um dicionário. A primeira itera pela lista recebida, e passa o elemento atual como argumento para a segunda.

```
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"])
```

Ao ser interpretado, o código acima define dois nomes locais: `imprime_cardapio` e `imprime_prato`. Para que a função seja de fato executada, usamos este nome seguido dos argumentos passados entre parênteses:

```
# 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)
```

o que resulta na seguinte saída quando executado:

```
Cardapio para hoje

    Arroz com brocolis ........... 9.90
      Soja com legumes ........... 7.80
             Lentilhas ........... 4.80

Total de pratos: 3
```

**Retornando um valor** No exemplo acima, as funções não 'retornam' valor algum, apenas exibindo informação. Para retornar um valor, basta usar a expressão `return` dentro da função. A primeira pergunta conceitual aqui seria 'retornar para quem?'; a resposta direta é 'para quem invocou a função'. Demonstrando por exemplos, a função abaixo:

```
def bar(t):
    return "The rain in Spain falls mainly in the %s." % t
```

define um valor de retorno, que é uma string; ao chamar esta função com um argumento:

```
a = bar("plains")
```

o código da função é avaliado e um valor, retornado, sendo armazenado na variável `a` (observe o uso do operador de atribuição seguido da chamada da função). Para visualizar o valor retornado, usamos a conveniente instrução `print`:

```
>>> print a
The rain in Spain falls mainly in the plains.
```

**Dica:** para retornar conjuntos de valores, basta retornar uma seqüência ou outro valor de tipo composto.

### 2.6.2 Truques com argumentos

**Argumentos nomeados** Argumentos podem ser fornecidos à função especificando-se o nome do parâmetro ao qual correspondem. O código abaixo é equivalente à chamada no exemplo anterior:

```
a = bar(t="plains")
```

mas fica explicitamente declarado que o parâmetro corresponde ao argumento t. Normalmente são nomeados argumentos para tornar a leitura do código mais fácil; no entanto, veremos a seguir como são essenciais para utilizar uma das formas de listas de argumentos variáveis.

**Valores padrão** Uma das funcionalidades mais interessantes nas funções Python é a possibilidade de definir valores padrão:

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

Neste exemplo, o valor padrão para a variável `taxa` é 0.1; se não for definido este argumento, o valor padrão será utilizado.

```
>>> print aplica_multa(10)
11.0
>>> print aplica_multa(10, 0.5)
15.0
```

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

**Conjuntos de argumentos opcionais** Uma forma de definir conjuntos de argumentos opcionais utiliza parâmetros 'curinga' que assumem valores compostos. Pode ser fornecido um parâmetro cujo nome é prefixado por um símbolo asterisco, que assumirá um conjunto ordenado de argumentos:

```
def desculpa(alvo, *motivos):
    d = "Desculpas %s, mas estou doente" % alvo
    for motivo in motivos:
        d = d + " e %s" % motivo
    return d + "."

>>> desculpa("senhor", "meu gato fugiu",
...          "minha tia veio visitar")

"Desculpas senhor, mas estou doente e meu gato fugiu e minha
tia veio visitar."
```

ou um parâmetro que assume um conjunto de argumentos nomeados, prefixado de dois asteriscos:

```
def equipe(diretor, produtor, **atores):
    for personagem in atores.keys():
        print "%s: %s" % (personagem, atores[personagem])
    print "-" * 20
    print "Diretor:  %s" % diretor
    print "Produtor: %s" % produtor

>>> equipe(diretor="Paul Anderson",
...        produtor="Paul Anderson",
```

```
...         Frank="Tom Cruise", Edmund="Pat Healy",
...         Linda="Julianne Moore")

Frank: Tom Cruise
Edmund: Pat Healy
Linda: Julianne Moore
--------------------
Diretor: Paul Anderson
Produtor: Paul Anderson
```

ou ainda as duas formas combinadas. Estes parâmetros especiais devem necessariamente ser os últimos definidos na lista de parâmetros da função. Note que estes argumentos especiais não agregam parâmetros que correspondem a argumentos explicitamente definidos, como `alvo`, `diretor` e `produtor` nos exemplos acima; estes são processados de maneira normal.

Conjuntos de argumentos opcionais são uma excelente forma de conferir flexibilidade a uma função mantendo compatibilidade com código pré-existente, sem prejudicar a sua legibilidade.

**Escopo de variáveis** Embora o tema escopo seja complexo, e não esteja restrito apenas a funções, cabe aqui uma observação sobre o escopo das variáveis definidas no corpo de funções.

Tanto as variáveis definidas no corpo de uma função, como a variável `i` no exemplo abaixo, quanto os argumentos da função são acessíveis apenas no **escopo local**; em outras palavras, apenas no bloco de código da própria função.

```
v = 0
w = 1
def verifica(a, b):
    i = 0
    # Neste bloco de código, v, w, a, b e i são acessíveis.

def cancela(x, y):
    i = 0
    # Neste bloco de código, v, w, x, y e i são acessíveis,
    # mas note que *este* i não tem nenhuma relação com
    # o i da função verifica() acima.
```

Variáveis definidas no corpo principal (ou seja, definidas em um bloco não-indentado, como as variáveis v e w acima) de um módulo podem ser *lidas* em qualquer função contida naquele arquivo; no entanto, não podem ser alteradas[9].

Caso se deseje definir ou atribuir um novo valor a uma variável global, existe uma instrução especial, `global`. Esta instrução indica que a variável cujo nome for especificado é para ser definida no escopo global, e não local.

```
v = 0
```

[9] O motivo pelo qual existe esta distinç ˜ao est´a intimamente associado à forma como Python atribui vari´aveis. Ao realizar uma operaç ˜ao de atribuiç ˜ao, a vari´avel ´e sempre defi nida no escopo *local*; acesso à vari´avel, no entanto, efetua uma busca nos escopos, do mais local para o mais global.

```
def processa(t):
    global v
    v = 1
```

No código acima, a função altera o valor da variável global v.

Há duas funções úteis para estudar os escopos que se aplicam em um determinado contexto:

- `locals()`, que retorna um dicionário descrevendo o escopo local ao contexto atual; os itens do dicionário são compostos dos nomes das variáveis definidas neste escopo e os valores aos quais correspondem.
- `global()`, que retorna um dicionário semelhante ao da função `locals()`, mas que descreve o escopo global.

O escopo de variáveis funciona de maneira semelhante quando tratando com métodos e definições de classes, que serão discutidos na seção 4.

## 2.7 Módulos e o comando `import`

Como dito anteriormente, cada arquivo contendo código Python é denominado um **módulo**. Na grande maioria das ocasiões utilizamos um ou mais módulos Python em combinação: o interpretador interativo é adequado para realizar experimentos curtos, mas não para escrever código de produção.

Um módulo Python consiste de código-fonte contido em um arquivo denominado com a extensão `.py`; como tal, pode conter variáveis, funções e classes; para fins de nomenclatura, qualquer um destes elementos contidos em um módulo é considerado um **atributo do módulo**.

Python, através do módulo, oferece excelentes mecanismos para modularizar código-fonte. Esta modularização pode ter diversas motivações: o programa pode ser grande demais, ter sub-partes reusáveis que devem ser separadas, ou ainda necessitar de módulos escritos por terceiros. Esta seção introduz este conceito através do comando `import`.

**Importando módulos e atributos de módulos** A instrução básica para manipular módulos é `import`. O módulo deve estar no caminho de procura de módulos do interpretador[10]. No exemplo a seguir, um módulo com o nome `os.py` é importado. Observe que a extensão `.py` *não* é incluída no comando `import` — apenas o nome principal:

```
>>> import os
>>> print os.getcwd()
/home/kiko
```

O módulo `os` define algumas funções internamente. No exemplo acima, invocamos a função `getcwd()` contida neste, *prefixando a função com o nome do módulo*. É importante esta

[10]O caminho de procura ´e uma lista de diret´orios onde o interpretador Python busca um m´odulo quando uma instruç ˜ao `import` ´e processada; normalmente esta lista inclui o diret´orio atual e os diret´orios de biblioteca padr˜ao. Pode ser manipulado pela vari´avel de ambiente `PYTHONPATH` ou pelo m´odulo `sys.path`.

observação: ao usar a forma `import módulo`, acessamos os atributos de um módulo usando esta sintaxe, idêntica à utilizada para acessar métodos da lista conforme descrito na seção 2.3.2.

Existe uma segunda forma do comando `import` que funciona de forma diferente. Ao invés de importar o módulo inteiro, o que nos obriga a usar as funções prefixadas pelo nome do módulo, este formato importa um atributo do módulo, deixando-o acessível localmente:

```
>>> from os import getcwd
>>> print getcwd()
/home/kiko
```

**Funções úteis** Há algumas funções pré-definidas no interpretador bastante úteis quando lidando com módulos e os atributos contidos em módulos:

- `dir(nome_módulo)` retorna uma lista dos nomes dos atributos contidos em um módulo, o que permite que você descubra interativamente quais símbolos e funções o compõem. Experimente, por exemplo, executar `print dir(os)` a partir do interpretador Python.
- `reload(nome_módulo)` recarrega o módulo importado a partir do seu arquivo; desta maneira, alterações podem ser efetuadas no arquivo do módulo e já utilizadas em uma sessão do interpretador, sem que seja necessário reiniciá-lo.

O mecanismo de importação possui uma série de nuances especiais, que estão associados ao tópico escopo, introduzido na seção anterior, e aos *namespaces*, que resumidamente determinam o conjunto de atributos acessíveis em um determinado contexto. Uma descrição mais detalhada destes tópicos é oferecida na seção *Python Scopes and Name Spaces* do tutorial Python.

## 2.8 Strings de documentação

Um recurso especialmente útil em Python é o suporte nativo à documentação de código por meio de strings localizadas estrategicamente, chamadas **docstrings**. Módulos, classes, funções e até propriedades podem ser descritas por meio de docstrings; o exemplo a seguir demonstra um módulo hipotético, `financ.py`, documentado adequadamente:

```
"""Módulo que contém funções financeiras."""

def calcula_juros(valor, taxa=0.1):
    """Calcula juros sobre um valor.

    Aplica uma taxa de juros fornecida sobre um valor
    e retorna o resultado. Se omitida a taxa, o valor
    0.1 será utilizado"""
    # ...

class Pagamento:
    """Classe que representa um pagamento a ser efetuado.
```

```
    Inclui informações de crédito e débito. Permite efetuar
    operações como devolução, cancelamento, transferência e
    pagamento em si. Possui o seguinte ciclo de vida:
    ...
    """
```

As docstrings do módulo acima podem ser visualizadas em tempo de execução, e mesmo a partir do modo interativo do interpretador por meio da função `help()` e do atributo `__doc__`:

```
>>> import financ
>>> help(calcula_juros)
Calcula juros sobre um valor.

    Aplica uma taxa de juros fornecida sobre um valor
    e retorna o resultado. Se omitida a taxa, o valor
    0.1 será utilizado

>>> print financ.__doc__
Módulo que contém funções financeiras.
```

Observe que a função `help` recebe como argumento *a própria função* `calcula_juros`; pode parecer pouco usual mas é um padrão comum em Python.

Este recurso é extremamente útil para o aprendizado da linguagem quando explorando objetos pré-definidos; utilize-o sempre que estiver necessitando de uma referência rápida:

```
>>> len([1,2,3])
3
>>> help(len)
Help on built-in function len:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Docstrings podem também ser processadas utilizando ferramentas externas, como o **epydoc**[11], gerando referências em formatos navegáveis como HTML.

# 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 não foram apresentadas no texto seguem:

[11] Dispon´ı vel em `http://epydoc.sourceforge.net/`; um exemplo da documentaç ˜ao gerada pode ser consultado em `http://www.async.com.br/projects/kiwi/api/`.

- `range(a,b)`: recebe dois inteiros, retorna uma lista de inteiros entre `a` e `b`, não incluindo `b`. Esta função é freqüentemente 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; e assim por diante.

```
>>> 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.
- `pow(a, n)`: recebe dois inteiros; retorna o resultado da exponenciação de `a` à ordem `n`. É equivalente à sintaxe `a ** n`.
- `chr(a)`: recebe um inteiro (entre 0 e 255) como parâmetro, retornando o caracter 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.
- `ord(a)`: recebe um único caracter 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.
- `max(a, b)`: retorna o maior entre `a` e `b`.
- `abs(n)`: retorna o valor absoluto de um número.
- `hex(n)` e `oct(n)`: retornam uma string contendo a representação em hexadecimal e octal, respectivamente, de um inteiro.

Há também funções de conversão explícita de tipo; as mais freqüentemente 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 seqüencias 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 seqüencia em uma lista ou tupla, respectivamente.

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

Além destas funções, existe uma grande biblioteca disponível nos módulos já fornecidos com o Python. Alguns destes módulos são discutidos na seção 5; como sempre, o manual Python é a referência definitiva no assunto.

As seções seguintes discutem algumas funções pré-definidas com comportamento especialmente relevante.

## 3.1 Manipulação de arquivos: a função `open()`

A função `open` é uma das mais poderosas do Python; serve para obter uma referência a um objeto do tipo arquivo. Assumindo que temos um arquivo chamado `arquivo.txt`, contendo um trecho de um livro famoso, podemos codificar o seguinte exemplo:

```
>>> a = open("arquivo.txt")
>>> print a
<open file 'arquivo.txt', mode 'r' at 0x820b8c8>
```

Uma vez obtida a referência ao objeto arquivo, podemos usar métodos específicos deste, como o método `read()`, que retorna o conteúdo do arquivo:

```
>>> texto = a.read()
>>> print texto
'...Would you tell me, please,
 which way I ought to go from here?'
'That depends a good deal on where you want to get to,'
 said the Cat.
'I don't much care where--' said Alice.
'Then it doesn't matter which way you go,' said the Cat.
```

**Sintaxe** O formato geral da função `open` é:

```
open(nome_do_arquivo, modo)
```

Ambos os parâmetros são strings. O modo determina a forma como o arquivo será aberto e é composto de uma ou mais letras; `'r'` (ou nada) abre para leitura, `'w'` abre para escrita, `'a'` abre para escrita, com dados escritos acrescentados ao final do arquivo. Se um símbolo `'+'` for agregado ao modo, o arquivo pode ser lido e escrito simultaneamente.

**Métodos do objeto arquivo** O objeto arquivo possui um conjunto de métodos úteis; os mais importantes são descritos abaixo. Note que o arquivo possui um conceito de posição atual: em um dado momento, operações serão realizadas com base em uma certa posição. Alguns métodos utilizam ou alteram esta posição; outros são operações globais, independentes da posição dela.

- `read()`: como visto acima, retorna uma string única com todo o conteúdo do arquivo.
- `readline()`: retorna a próxima linha do arquivo, e incrementa a posição atual.
- `readlines()`: retorna todo o conteúdo do arquivo em uma lista, uma linha do arquivo por elemento da lista.
- `write(data)`: escreve a string `data` para o arquivo, na posição atual ou ao final do arquivo, dependendo do modo de abertura. Esta função falha se o arquivo foi aberto com modo `'r'`.
- `seek(n)`: muda a posição atual do arquivo para o valor indicado em `n`.
- `close()`: fecha o arquivo.

Qualquer arquivo pode ser aberto e lido desta forma; experimente com esta função, abrindo alguns arquivos locais, lendo e modificando-os.

## 3.2 Leitura do teclado: `raw_input()`

Outra função útil sobretudo para programas Python interativos é a função `raw_input`: lê do teclado uma string, e a retorna. Esta função possui um parâmetro opcional, que é uma mensagem a ser fornecida ao usuário. O exemplo seguinte utiliza um módulo com o nome `leitura.py`:

```
a = raw_input("Entre com um número de 0 a 10: ")
n = int(a)
if not 0 < n < 10:
    print "Número inválido."
if n % 2 == 0:
    print "É Par"
else:
    print "É Ímpar"
```

Este exemplo, quando executado e sendo fornecido o número 7, gera a seguinte saída:

```
>>> import leitura
Entre com um número de 0 a 10: 7
É Ímpar
```

# 4 Orientação a Objetos

Embora termos importantes como classes, objetos e módulos tenham sido introduzidos anteriormente, ainda não discutimos em detalhes os conceitos e a implementação de orientação a objetos (OO) em Python. Python suporta orientação a objetos utilizando um modelo flexível e particularmente homogêneo, que simplifica a compreensão dos mecanismos OO fundamentais da linguagem.

## 4.1 Conceitos de orientação a objetos

Orientação a objetos é um termo que descreve uma série de técnicas para estruturar soluções para problemas computacionais. No nosso caso específico, vamos falar de **programação OO**, que é um paradigma de programação no qual um programa é estruturado em objetos, e que enfatiza os aspectos abstração, encapsulamento, polimorfismo e herança[12].

Convencionalmente, um programa tem um fluxo linear, seguindo por uma função principal (ou o corpo principal do programa, dependendo da linguagem de programação) que invoca funções auxiliares para executar certas tarefas à medida que for necessário. Em Python é perfeitamente possível programar desta forma, convencionalmente chamada de **programação procedural**.

Programas que utilizam conceitos OO, ao invés de definir funções independentes que são utilizadas em conjunto, dividem conceitualmente o 'problema' (ou **domínio**[13]) em partes independentes (**objetos**), que podem conter **atributos** que os descrevem, e que implementam o comportamento do sistema através de funções definidas nestes objetos (**métodos**). Objetos (e seus métodos) fazem referência a outros objetos e métodos; o termo 'envio de mensagens' é utilizado para descrever a comunicação que ocorre entre os métodos dos diferentes objetos.

Na prática, um programa orientado a objetos em Python pode ser descrito como um conjunto de classes — tanto pré-definidas quanto definidas pelo usuário — que possuem atributos e métodos, e que são **instanciadas** em objetos, durante a execução do programa. A seção seguinte concretiza estes conceitos com exemplos.

## 4.2 Objetos, classes e instâncias

Objetos são a unidade fundamental de qualquer sistema orientado a objetos. Em Python, como introduzido na seção 2.3.2, *tudo é um objeto* — tipos, valores, classes, funções, métodos e, é claro, instâncias: todos possuem atributos e métodos associados. Nesta seção serão descritas as estruturas da linguagem para suportar objetos *definidos pelo programador*.

[12] Mais detalhes sobre conceitos fundamentais de OO podem ser obtidos em `http://en.wikipedia.org/wiki/Object-oriented_programming`

[13] O 'problema' ou 'dom´ı nio' de um software compreende o conjunto de tarefas essenciais que este deve realizar; por exemplo, o dom´ı nio de um editor de textos compreende escrever, corrigir e alterar documentos —e um conceito fundamental, muito provavelmente modelado em uma classe OO, seria o Documento.

**Classes** Em Python, a estrutura fundamental para definir novos objetos é a **classe**. Uma classe é definida em código-fonte, e possui nome, um conjunto de atributos e métodos.

Por exemplo, em um programa que manipula formas geométricas, seria possível conceber uma classe denominada `Retangulo`:

- Esta classe possuiria dois atributos: `lado_a` e `lado_b`, que representariam as dimensões dos seus lados.
- A classe poderia realizar operações como `calcula_area` e `calcula_perimetro`, e que retornariam os valores apropriados.

Um exemplo de uma implementação possível desta classe está no módulo `retangulo.py` a seguir:

```
class Retangulo:
    lado_a = None
    lado_b = None

    def __init__(self, lado_a, lado_b):
        self.lado_a = lado_a
        self.lado_b = lado_b
        print "Criando nova instância Retangulo"

    def calcula_area(self):
        return self.lado_a * self.lado_b

    def calcula perímetro(self):
        return 2 * self.lado_a + 2 * self.lado_b
```

Esta classe define os dois atributos descritos acima, e três métodos. Os três métodos definidos incluem *sempre*, como primeiro argumento, uma variável denominada **self**, que é manipulada no interior do método. Este é um ponto fundamental da sintaxe Python para métodos: o primeiro argumento é especial, e convenciona-se utilizar o nome self para ele; logo a seguir será discutido para que existe.

Dois dos métodos codificados correspondem às operações descritas inicialmente, e há um método especial incluído: `__init__()`. O nome deste método tem significância particular em Python: é o método **construtor**, um método **opcional** invocado quando a classe é **instanciada**, que é o nome dado à ação de criar objetos a partir de uma classe.

**Instâncias** A instância é objeto criado com base em uma classe definida. Este conceito é peculiar, e leva algum tempo para se fixar. Uma descrição abstrata da dualidade classe-instância: a classe é apenas uma matriz, que especifica objetos, mas que não pode ser utilizada diretamente; a instância representa o objeto concretizado a partir de uma classe. Eu costumo dizer que a classe é 'morta', existindo apenas no código-fonte, e que a instância é 'viva', porque durante a execução do programa são as instâncias que de fato 'funcionam' através da invocação de métodos e manipulação de atributos.

Conceitualmente, a instância possui duas propriedades fundamentais: a classe a partir da qual foi criada, que define suas propriedades e métodos padrão, e um **estado**, que representa o conjunto de valores das propriedades e métodos definidos naquela instância específica. A instância possui um **ciclo de vida**: é criada (e neste momento seu construtor é invocado), manipulada conforme necessário, e destruída quando não for mais útil para o programa. O estado da instância evolui ao longo do seu ciclo de vida: seus atributos são definidos e têm seu valor alterado através de seus métodos e de manipulação realizada por outros objetos.

> O que Python chama de 'instância' é freqüentemente denominado 'objeto' em outras linguagens, o que cria alguma confusão uma vez que *qualquer* dado em Python pode ser considerado um 'objeto'. Em Python, instâncias são objetos **criados a partir de uma classe definida pelo programador**.

Retomando o nosso exemplo acima: a partir da classe `Retangulo` que foi definida, poderíamos instanciar objetos retângulo específicos: um com lados de comprimento 1 e 2, e outro com lados de comprimento 2 e 3:

```
>>> from retangulo import Retangulo
>>> r1 = Retangulo(1, 2)
Criando nova instância Retângulo
>>> r2 = Retangulo(2, 3)
Criando nova instância Retângulo
```

Observe que ao instanciar o retângulo:

- Foi importado e utilizado o nome da classe seguido de parênteses.
- Foram fornecidos como argumentos — entre parênteses — dois valores, correspondendo aos comprimentos dos lados diferentes dos retângulos (1 e 2 para o primeiro retângulo, e 2 e 3 para o segundo).
- Estes argumentos são passados — transparentemente — para o método construtor da classe `Retangulo`. O código do método está reproduzido aqui para facilitar a leitura:

```
def __init__(self, lado_a, lado_b):
    self.lado_a = lado_a
    self.lado_b = lado_b
    print "Criando nova instância Retangulo"
```

Aqui cabe uma pausa para revelar o propósito da variável `self`, definida como primeiro argumento dos métodos. Esta variável representa *a instância sobre a qual aquele método foi invocado*. Esta propriedade é de importância fundamental para OO em Python, porque através desta variável é que atributos e métodos desta instância podem ser manipulados no código dos seus métodos.

Continuando com a análise do bloco de código acima:

- Nas duas primeiras linhas do método — onde é feita atribuição — o código do construtor está atribuindo valores para dois atributos, `lado_a` e `lado_b`, *na instância*, aqui representada pelo argumento `self`. Neste momento, o **estado** da instância passa a conter os dois atributos novos.
- O construtor inclui uma instrução `print` didática que imprime uma mensagem para demonstrar que foi executado; a mensagem aparece na saída do interpretador.

Uma vez instanciados os retângulos, podemos acessar seus métodos. De maneira idêntica aos métodos da lista apresentados na seção 2.3.2, a sintaxe utiliza um ponto seguido do nome do método acompanhado de parênteses:

```
>>> print r1.calcula_area()
2
>>> print r2.calcula_perimetro()
10
```

Conforme esperado, as funções retornaram os valores apropriados para cada instância. Fazendo mais uma demonstração do uso do argumento `self`, vamos observar o código de um dos métodos:

```
def calcula_area(self):
    return self.lado_a * self.lado_b
```

O onipresente argumento `self` aqui é utilizado como meio de acesso aos atributos `lado_a` e `lado_b`. Este código permite visualizar o funcionamento pleno deste mecanismo: ao ser invocado o método `calcula_area` sobre a instância `r1`, o argumento `self` assume como valor esta mesma instância; portanto, acessar atributos de `self` internamente ao método equivale, na prática, a acessar atributos de `r1` externamente.

Em Python é possível, inclusive, acessar os atributos da instância diretamente, sem a necessidade de usar um método:

```
>>> print r1.lado_a
1
>>> print r1.lado_b
2
```

Os valores, logicamente, correspondem aos inicialmente fornecidos à instância por meio do seu construtor.

**Atributos Privados e Protegidos** Algumas linguagens permitem restringir acesso aos atributos de uma instância, oferecendo o conceito de **variável privada**. Python não possui uma construção sintática literalmente equivalente, mas existem duas formas de indicar que um atributo não deve ser acessado externamente:

- A primeira forma é implementada por meio de uma convenção, não havendo suporte específico na linguagem em si: convenciona-se que atributos e métodos cujo nome é iniciado por um sublinhado (como `_metodo_a`) não devem ser acessados externamente em situações 'normais'.

- A segunda forma estende esta convenção com suporte no próprio interpretador: métodos e atributos cujo nome é iniciado por dois sublinhados (como `__metodo_a`) são considerados de fato privados, e têm seus nomes alterados de maneira transparente pelo interpretador para assegurar esta proteção. Este mecanismo é descrito em maior detalhes na seção *Private Variables* do tutorial Python.

## 4.3 Heranç a

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. Os exemplos até agora evitaram o uso de herança, mas nesta seção é possível apresentar a sintaxe geral para definição de uma classe:

```
class nome-classe(base1, base2, ..., basen):
    atributo-1 = valor-1
    .
    .
    atributo-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 Foo:
    a = 1
    def cheese(self):
        print "cheese"

    def foo(self):
        print "foo"
```

```
class Bar(Foo):
    def bar(self):
        print "bar"

    def foo(self):              # método redefinido
        print "foo de bar!"
```

uma instância da classe `Bar` tem acesso aos métodos `cheese()`, `bar()` e `foo()`, este último sendo redefinido localmente:

```
>>> b = Bar()
>>> b.cheese()
cheese
>>> b.foo()         # saída demonstra método redefinido
foo de bar!         # em Bar
>>> b.bar()
foo
>>> print b.a       # acesso transparente ao atributo
1                   # definido em Foo
```

enquanto uma instância da classe `Foo` tem acesso apenas às funções definidas nela, `foo()` e `cheese`:

```
>>> f = Foo()
>>> f.foo()
foo
```

**Invocando métodos de classes-base** Para acessar os métodos de uma classe-base, usamos uma construção diferente para invocá-los, que permite especificar qual classe armazena o método sendo chamado. Seguindo o exemplo, vamos novamente a redefinir o método `Bar.foo()`:

```
class Bar(Foo):
    # ...
    def foo(self):
        Foo.foo(self)
        print "foo de bar!"
```

Nesta versão, o método `foo()` inclui uma chamada ao método `Foo.foo()`, que conforme indicado pelo seu nome, é uma referência direta ao método da classe base. Ao instanciar um objeto desta classe:

```
>>> b = Bar()
>>> b.foo()
foo
foo de bar!
```

pode-se observar que são executados ambos os métodos especificados. Este padrão, aqui demonstrado de forma muito simples, pode ser utilizado em situações mais elaboradas; seu uso mais freqüente é para invocar, a partir de um construtor de uma classe, o construtor das suas classes-base.

**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`.

**Atributos de classe *versus* atributos de instância** Uma particularidade em Python, que deriva da forma transparente como variáveis são acessadas, é a distinção entre atributos definidos em uma classe, e atributos definidos em uma instância desta classe. Observe o código a seguir:

```
class Foo:
    a = 1
```

A classe acima define uma variável `a` com o valor 1. Ao instanciar esta classe,

```
>>> f = Foo()
>>> print f.a
1
```

observa-se que a variável parece estar definida na instância. Esta observação convida a algumas indagações:

- Onde está definida a variável `a` – na classe ou na instância?
- Se atribuirmos um novo valor a `f.a`, como abaixo:

  ```
  >>> f.a = 2
  ```

  estamos alterando a classe ou a instância?
- Uma vez atribuído o novo valor, que valor aparecerá para o atributo `a` no próximo objeto instanciado a partir de `Foo`?

As respostas para estas perguntas são todas relacionadas a um mecanismo central em Python, que é o **protocolo `getattr`**. Este protocolo dita como atributos são transparentemente localizados em uma hierarquia de classes e suas instâncias, e segue a seguinte receita:

1. Ao acessar um atributo de uma instância (por meio de uma variável qualquer ou `self`) o interpretador tenta localizar o atributo no estado da instância.

2. Caso não seja localizado, busca-se o atributo na classe da instância em questão. Por sinal, este passo é o que permite que métodos de uma classe sejam acessíveis a partir de suas instâncias.

3. Caso não seja localizado, busca-se o atributo entre as classes base definidas para a classe da instância.

4. Ao **atribuir** uma variável em uma instância, este atributo é sempre definido no estado local da instância.

Uma vez compreendido este mecanismo, é possível elucidar respostas para as questões acima. No exemplo, a variável `a` está definida na classe `Foo`, e pelo ponto 2 acima descrito, é acessível como se fosse definida pela própria instância. Ao atribuir um valor novo a `f.a`, estamos definindo uma nova variável `a` no estado local da variável `f`, o que não tem nenhum impacto sobre a variável `a` definida em `Foo`, nem sobre novas instâncias criadas a partir desta.

Se o descrito acima parece confuso, não se preocupe; o mecanismo normalmente funciona exatamente da maneira que se esperaria de uma linguagem orientada a objetos. Existe apenas uma situação 'perigosa', que ocorre quando usamos atributos de classe com valores mutáveis, como listas e dicionários.

```
class Foo:
    a = [1,2]
```

Nesta situação, quando criamos uma instância a partir de Foo, a variável a pode ser **alterada** por meio desta instância. Como não foi realizada **atribuição**, a regra 4 descrita acima não se aplica:

```
>>> f = Foo()
>>> f.a.append(3)
>>> g = Foo()
>>> print g.a
[1, 2, 3]
```

e a variável da classe é de fato modificada. Esta particularidade é freqüentemente fonte de bugs difíceis de localizar, e por este motivo se recomenda fortemente que **não se utilize variáveis de tipos mutáveis em classes**.

## 4.4 Introspecção e reflexão

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 está organizado este 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.

# 5 Alguns 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*[14] atual; `sleep(n)`, que pausa a execução por n segundos; e `strftime(n)`, que formata um timestamp 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.

[14] O n´umero de segundos desde 1º de janeiro, 1970, que por sinal ´e a data padr˜ao do in´ı cio do tempo no Unix.

- `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 seqüência `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.

## 5.1 Módulos independentes

Além dos módulos distribuídos com o Python, existem vários módulos auxiliares. Justamente por serem numerosos e independentemente fornecidos, não é possível descrevê-los na sua totalidade; vou apenas citá-los; podem ser obtidas maiores informações nos links providos.

- **win32pipe**: permite, na plataforma Windows, executar programas win32 e capturar sua saída em uma string para manipulação posterior. Acompanha a distribuição Activestate Python: `http://www.activestate.com/Products/ActivePython/`.

- **PIL**: Python Imaging Library, que oferece funções para processamento, manipulação e exibição de imagens. `http://www.pythonware.com/products/pil/`

- **NumPy**: provê mecanismos simples e de alto desempenho para manipular matrizes multidimensionais; ideal para operações numéricas de alto volume que necessitem de velocidade. `http://numpy.sourceforge.net/`

- **HTMLgen**: uma biblioteca de classes que gera documentos HTML conforme padrões prédefinidos. Oferece classes para manipular tabelas, listas, e outros elementos de formatação. `http://starship.python.net/crew/friedrich/HTMLgen/html/`

- **DB-API**: Database Application Programming Interface; na realidade, um conjunto de módulos que acessam bases de dados de uma forma padronizada. A API especifica uma forma homogênea de se fazer consultas e operações em bases de dados relacionais (SQL); diversos módulos implementam esta API para bases de dados específicas. `http://www.python.org/topics/database/`

- **mx**: oferece uma série de extensões à linguagem, incluindo operações complexas de data e hora, funções nativas estendidas, e ferramentas para processamento de texto. `http://www.egenix.com/files/python/`

- **PyGTK**: É outro pacote que permite construir aplicações gráficas com o Python; pode ser usado em conjunto com o Glade, um construtor visual de interfaces.
  `http://www.pygtk.org/`
- **wxPython**: uma biblioteca de classes que permite construir aplicações gráficas multi-plataforma usando Python. Há um construtor visual de interfaces disponível, o Boa Constructor. `http://www.wxpython.org/`

Todos os módulos citados se comportam como módulos Python 'normais'; são utilizados por meio da instrução `import`, e boa parte possui documentação e símbolos internos listáveis.

Esta não é uma lista exaustiva, e há muitos outros módulos úteis; há boas referências que listam módulos externos, incluindo o índice de pacotes oficial PyPI:
`http://www.python.org/pypi`.

# 6 Fechamento

Aqui termina este tutorial, que cobre os aspectos fundamentais da linguagem Python. Com base no texto, é possível enfrentar uma tarefa prática, que irá exercitar seu conhecimento e ampliar sua experiência. Escolha um problema, e tente passar da sua modelagem para Python.