Luiz Gabriel Olivério Noronha

Bruno Nunes Pinheiro

Carlos Eduardo de Freitas Oliveira

Carlos Eduardo Ribeiro Júnior

Osmiro Marco Júnior

Sobre a apostila

Esta apostila foi escrita por ex-estudantes da Universidade Santa Cecília, localizada em Santos – SP, como exigência parcial para obtenção do título de Bacharel em Ciência da Computação, sob orientação do Prof. Maurício Neves Asenjo. Seu conteúdo é destinado a estudantes iniciantes de programação, que tenham pelo menos conhecimento de lógica de programação, conceitos de bancos de dados e SQL (*Structured Query Language*).

Nessa apostila nós cobriremos os recursos fundamentais da linguagem Python e de sua biblioteca-padrão, além de alguns assuntos mais avançados e teóricos, que darão ao estudante a base necessária para poder se aprofundar em outros tópicos mais específicos, seja da biblioteca-padrão do Python, como *frameworks* e bibliotecas de terceiros.

Para quem já é desenvolvedor e deseja aprender Python, recomendamos a leitura do livro *Python para Desenvolvedores - 2ª edição*, de Luiz Eduardo Borges, disponível para *download* no seu site oficial, http://ark4n.wordpress.com/python/.

Prefácio da 2ª Edição

A 1ª edição desta apostila foi disponibilizada para *download* no Source Forge em novembro de 2010, época em que o trabalho que a apostila fazia parte foi apresentado na 10ª edição do SEMESP CONIC (Congresso Nacional de Iniciação Científica). Devido a este limite de tempo, alguns conteúdos tiveram de ser deixados de fora e diversos tópicos ficaram confusos.

Esta 2º edição, lançada em novembro de 2011, foi totalmente revisada, os conteúdos que foram deixados de fora na 1ª edição foram incluídos, bem como alguns tópicos foram re-escritos. Dentre os tópicos incluídos podemos citar:

- Os tipos `NoneType` e `Unicode`;
- O capítulo *Trabalhando com bancos de dados*;
- O operador de identidade de objetos.

A partir dessa edição a apostila passa a ser mantida por somente um dos membros do grupo que elaborou a apostila, Luiz Gabriel Olivério Noronha.

Site oficial

A edição mais recente dessa apostila encontra-se disponível no formato PDF para *download* no endereço http://sourceforge.net/projects/apostila-python/. Dúvidas, reclamações, sugestões, correção (e por que não elogios?!) podem ser enviados para o *e-mail* do mantenedor do projeto, gabriel@phasedesign.com.br. Para se manter informado em relação a novos lançamentos e correções da apostila, basta seguir o perfil @GabrielOliverio no Twitter.

Licença

Convenções

Nesta apostila utilizamos diversas convenções de estilos para distinguir mais facilmente os diferentes tipos de informações nela contida.

Exemplos de códigos-fontes são separados do texto e utilizam a fonte monosespaçada `Courier New`; listagens com mais de uma linha aparecem enumeradas para facilitar a identificação na sua explicação, como em:

```
1  print 'ola mundo'
```

Nomes de variáveis, palavras-chave, comandos da linguagem, funções e classes, quando em meio ao texto explicativo, também são grafados com fonte monoespaçada.

Sentenças que foram disparadas no interpretador interativo do Python, além de seguir a convenção de estilo anterior, são precedidas por >>>.

Nomes de pessoas, instituições e palavras de outros idiomas estão destacados em *itálico*, assim como termos importantes em sublinhado.

Sumário

Introdução

O que é Python?

Python é uma linguagem de programação de propósito geral, é utilizada dentre diversas outras aplicações, para o desenvolvimento de aplicativos de console (modo texto), aplicativos que utilizam formulários, aplicações *web*, científicas, *parsers* e também como linguagem de *scripting*, para estender aplicativos e automatizar tarefas.

Dentre suas características pode-se ressaltar que Python:

- ***É interpretada***

  Ao contrário das linguagens compiladas, que transformam o código escrito dos programas para uma plataforma especifica, por exemplo, Windows ou Linux, Python transforma o código do programa em *bytecodes* e estes são executados por um interpretador, o que possibilita o aplicativo ser executado em várias plataformas com poucas ou mesmo nenhuma alteração.

- ***É código-aberto***

  Python é absolutamente livre, mesmo para fins comerciais, permitindo a venda de um produto escrito em Python, ou um produto que venha com o interpretador embutido sem pagar taxas ou licenças para isso.

- ***Possui tipagem dinâmica e forte***

  Tipagem dinâmica se refere à linguagem não associar tipos às variáveis e sim aos seus valores; e como tipagem forte, entende-se que não é possível realizar operações com valores que possuem tipos incompatíveis de dados, como, por exemplo, unir um número a uma *string*.

- ***Possui interpretador interativo***

  Python possui um interpretador interativo que permite testar códigos de maneira fácil e rápida, sem a necessidade de criar um arquivo só para testar um bloco de código.

- ***Tudo em Python é um objeto***

  Tudo em Python - módulos, tipos de dados, variáveis, classes e funções, são objetos e como tais, possuem atributos (dados) e métodos (funções) vinculados que permitem manipular esses atributos.

- ***Suporta múltiplos paradigmas de programação***

  Paradigmas de programação são estilos de resolver problemas específicos na engenharia de softwares. Há vários paradigmas de programação e cada um é mais

adequado para resolver um problema de maneira mais fácil do que o outro. Python suporta os seguintes paradigmas de programação:

- Procedural;
- Orientado a objetos;
- Funcional;

Veremos mais detalhadamente os paradigmas de programação mais adiante na apostila.

História

Python foi criada no início de 1990 por *Guido van Rossum* no CWI – *Centrum Wiskunde & Informatic* (Instituto Nacional de Pesquisa de Matemática e Ciência da Computação, em holandês) nos Países Baixos, para ser a sucessora da linguagem de programação ABC. Seu nome é baseado na série de televisão britânica *Monty Python's Flying Circus* e não na cobra Píton (Python, em ingles) como muitos pensam. Embora *Guido* inclua contribuições de muitos programadores, ele ainda se mantém como o principal autor da linguagem e é conhecido na comunidade como Benevolent Dictator for Life (Ditador Benevolente para a Vida, em inglês).

Em 1995, *Guido* continuou seu trabalho na *Corporation for National Research Initiatives* (Corporação para Iniciativas de Pesquisas Nacionais, em inglês), em *Reston – Virginia*, onde foram lançadas várias versões do Python. Em maio de 2000, *Guido* e a equipe de desenvolvimento do Python se mudaram para o *BeOpen.com* para formar o time *BeOpen PythonLabs*. Em outubro do mesmo ano, o time *PythonLabs* se muda para a *Digital Creations,* agora chamada de *Zope Corporation*.

Em 2001, a *Python Software Foundation* – PSF – foi formada, uma organização sem fins lucrativos criada especificamente para possuir as propriedades intelectuais relacionadas à linguagem Python.

Implementações

Uma implementação de Python deve ser entendida como um programa ou ambiente que fornece suporte à execução de programas escritos em Python ou algum dialeto similar da linguagem.

Há diversos pacotes de software que fornecem o que todos conhecemos como Python, apesar de alguns deles serem mais como distribuições ou variantes de alguma implementação existente do que uma implementação completamente nova da linguagem. A implementação

escrita na linguagem C por *Guido van Rossum* é chamada de CPython ou simplesmente de Python.

Há outras implementações da linguagem, como:

- CLPython: Implementação de Python em Common Lisp
- IronPython: Implementação para a plataforma .Net/Mono. Possibilita desenvolver programas que utilizam as classes do Microsoft Framework .Net / Mono e integrá-los com aplicativos escritos em C#, VB.Net ou quaisquer linguagens que sigam a *Common Language Specification* (CLS);
- Jython: Interpretador de Python implementado em Java. Possibilita compilar programas escritos em Python em arquivos *bytecodes* Java, utilizar as classes da plataforma Java e integrar aplicativos com outros escritos para esta plataforma;
- Python for S60: Conhecido como PyS60, é a implementação de Python portado para a plataforma de software S60 da Nokia, baseado na versão 2.2.2 do Python.

Versões

As implementações oficiais do Python são mantidas pela PSF e no momento se encontram nas versões 2.7.2 e 3.2.2, disponíveis para *download* no endereço http://www.python.org/download.

As versões 2.x e 3.x estão previstas para coexistir por vários lançamentos em paralelo, onde a série 2.x existe em grande parte devido à compatibilidade com bibliotecas de terceiros e possui algumas novas características que estão sendo portadas da série 3.x. Devido a esta compatibilidade, utilizaremos nesta apostila a versão 2.7, que corrige alguns *bugs* da versão anterior e implementa alguns recursos. Apesar de utilizarmos a versão 2.7, nada o impede de utilizar outra versão, desde que seja superior 2.5.

A maioria das distribuições Linux já vem com o interpretador do Python instalado. Para plataforma Windows há um instalador, que inclui o interpretador, a documentação e um ambiente de desenvolvimento integrado (IDE) – o IDLE.

Ambientes de desenvolvimento e editores de texto

Tanto Ambientes de Desenvolvimento Integrado (IDE's – *Integrated Development Environment*) quanto editores de texto para programação, colaboram bastante no desenvolvimento de aplicações em qualquer linguagem de programação, reduzindo o número de erros que o programador possa cometer.

Como dissemos anteriormente, o IDLE já vem incluido no instalador do Python para Windows, e no Linux, pode ser baixado e instalado através de gerenciadores de pacotes ou baixado no site do Python http://www.python.org e compilado através do método CMMI (*Configure*, *Make*, *Make Install*).

O instalador do Python no Windows inclui o IDLE, cujo nome é tanto uma homenagem a um dos integrantes do grupo *Monty Python's Flying Circus*, *Eric Idle*, quanto abreviação de Integrated DeveLopment Editor, e possui recursos como coloração de sintaxe (*syntax highlighting*), autocompletar (*autocomplete*) e um *debugger*. Ele será utilizado nesta apostila devido a sua aparência simples para iniciantes em programação e já vir incluso no instalador do Python, mas nada impede o estudante de utilizar outros IDE's, como:

- PyScripter (Windows) - http://code.google.com/p/pyscripter/;
- SPE (Multi-plataforma) - http://pythonide.blogspot.com/;
- Eclipse com o *plugin* PyDev (Multi-plataforma - melhor escolha) - http://www.eclipse.org/.

Ou de editores, como:

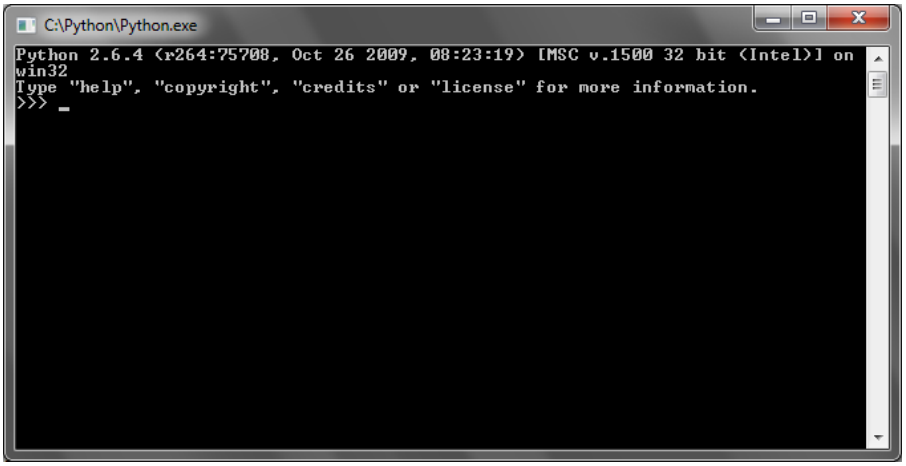
- Notepad++ (Windows) - http://notepad-plus-plus.org/;
- KomodoEdit (Multi-plataforma) - http://www.activestate.com/komodo-edit;
- Vim, Emacs, GEdit, entre muitos outros na plataforma Linux.

Modos de execução

Python possui dois modos de execução: o modo interativo e o modo de compilação e interpretação.

Interativo

O interpretador interativo de Python pode ser encontrado como "Python (command line)" na aba Python 2.7 no menu Iniciar no Windows, ou digitando `python` e pressionando ENTER em um terminal de Linux. Isto também pode ser configurado em ambientes Windows incluindo o caminho do interpretador do Python na variável de sistema PATH. Mais informações no Apêndice A da apostila.



Os sinais >>> mostram que o interpretador está pronto para receber comandos. Digite os seguintes comandos no interpretador interativo:

```
1  >>> print "Hello, Magrathea!"                [ENTER]
2  >>> if 4 > 2:                                [ENTER]
3  ...   print "4 > 2"                          [ENTER]
4  ...                                          [ENTER]
5  4 > 2
6  >>>
```

Na primeira linha vemos `print "Hello, Magrathea!"`. `print` é um comando da linguagem, e imprime um valor, no caso o texto "Hello, Magrathea!". Ao pressionar ENTER após o comando `print` o interpretador está pronto para receber outro comando.

Na segunda linha, ao digitar `if 4 > 2:` e pressionar ENTER, o interpretador mostra `...`, que significa que ele está esperando completar a sentença anterior. Como veremos mais adiante, um `if` testa uma condição, se ela for verdadeira então executa um ou mais comandos, portanto, pelo menos um comando tem que ser inserido (ou um erro seria exibido). Ao entrar com o comando `print "4 > 2"` e pressionar ENTER, o interpretador exibe `...` novamente, esperando mais sentenças. Podemos pressionar ENTER e a sentença estará terminada, o interpretador exibe o resultado, e então estará pronto para receber outro comando.

Nos exemplos acima não precisaríamos necessariamente ter chamado o comando `print` para imprimir os textos no interpretador interativo, podemos passar objetos/valores diretamente para o interpretador do Python e ele se vira para imprimir o valor mais que faz mais sentido de acordo com seu tipo. Utilizamos o comando `print` somente para demonstrar de uma forma didática a utilização do interpretador interativo do Python, mas lembre-se de nunca omitir o comando `print` em um *script* no modo Compilação e interpretação, que veremos logo a diante. Vamos ver alguns exemplos:

```
1  >>> 'String'
2  'String'
3  >>> 1+1
4  2
5  >>> object()
6  <object object at 0x006F14B0>
```

Na linha 1 passamos uma string para o interpretador interativo. Como veremos no próximo capítulo, uma string é o tipo de dado que utilizamos quando queremos definir sequencias de caracteres, ou seja, textos. Na linha 3, passamos uma expressão aritmética, e o interpretador respondeu com o resultado da expressão. Até este ponto o interpretador interativo imprimiu valores coerentes para todas das instruções que passamos, mas na linha 6 o texto impresso pareceu um tanto confuso, não? O valor impresso na linha 6 significa que instanciamos um objeto do tipo `object` e o armazenamos no endereço de memória 0x00gF14B0. Não se preocupe se você não entendeu isso de instanciar objetos, veremos isso bem mais a frente, em Orientação a Objetos.

Compilação e interpretação

O código-fonte de um programa escrito em Python é traduzido para *bytecodes*, que é um formato binário com instruções a serem executadas pelo interpretador. Em linguagens compiladas, o código-fonte é compilado (traduzido) para código de máquina, que depende da plataforma sendo executada. Como o interpretador do Python traduz o código para o interpretador, e há interpretadores para a maioria as plataformas, o código escrito provavelmente será executado em todas elas com poucas, ou mesmo nenhuma modificação.

Exemplo prático

Abra o IDLE, clique em *File* e em seguida em *New Window*. Uma nova janela será exibida, nela, digite os seguintes comandos:

```
1  print "Ola, mundo!"
2  raw_input "Pressione qualquer tecla para continuar..."
```

No menu *File*, clique em *Save*, escolha o local a salvar o arquivo e o nomeie como `ola_mundo.py`. Com o arquivo salvo, podemos executá-lo clicando em *Run* e em seguida em *Run Module*, ou pressionando a tecla de atalho F5. Desta forma, o programa é executado em um *shell* no próprio IDLE, porém podemos também executá-lo clicando duas vezes sobre ele

ou chamando o interpretador Python na linha de comando e passando o nome/caminho do arquivo como argumento [1]:

```
C:\caminho_do_arquivo> python ola_mundo.py
```

Ao fazer isso, o interpretador compilará o código-fonte para *bytecode* e o armazenará em disco. Na próxima vez que o programa for executado o interpretador não precisará compilar o arquivo novamente, ele se encarrega de executar o programa compilado, a não ser que alterações tenham sido feitas no código fonte, onde o interpretador compilará novamente o código-fonte do programa para executá-lo na sua máquina virtual.

O *bytecode* é armazenado em disco na mesma pasta do código-fonte e com o mesmo nome do arquivo, porém com a extensão `.pyc` (*bytecode* normal) ou `.pyo` (*bytecode* otimizado), dependendo da opção que for passada para o interpretador, sendo `.pyc` a opção padrão. O *bytecode* também pode ser empacotado junto com o interpretador em um executável, para facilitar a distribuição da aplicação, eliminando a necessidade de instalar Python em cada computador que for executar o aplicativo.

O resultado do programa será um "Ola, mundo!" e um "Pressione qualquer tecla para continuar..." impressos na tela. O comando `raw_input` exibe um texto na tela e captura um valor digitado e pode armazená-lo em uma variável, como veremos mais adiante. Neste caso, não estamos armazenando nada, utilizamos `raw_input` somente para dar tempo de você ler o texto caso o execute clicando duas vezes sobre o arquivo no seu gerenciador de arquivos. Experimente remover a segunda linha e executar o programa novamente para ver o que acontece – o terminal aparece, o texto é impresso e então o terminal se fecha antes mesmo de dar tempo de você ler o texto.

Atenção

Caso você tenha testado outros textos no exemplo prático anterior, com caracteres *não-ascii*, como letras acentuadas e o "ç", e tenha recebido uma mensagem de erro como `SyntaxError: Non-ASCII character '\xe3' in file ola_mundo.py ...`, não se preocupe - veremos mais a frente como fazemos para representar nossos caracteres latinos no Python.

[1] Para que isso funcione é necessário incluir o caminho onde o interpretador do Python foi instalado (geralmente C:\Python27) na variável de ambiente PATH e reiniciar o computador para que as alterações sejam aplicadas. Mais informações no apêndice A da apostila.

Sintaxe

Um programa escrito em Python é constituído de instruções que podem ser distribuídas em várias linhas no código-fonte. Uma das formas de se fazer isso é utilizando o caractere barra-invertida (\ ). Exemplos:

```
1  x = 3 * 4 \
2  5.0
3  mensagem1 = "Texto de uma linha"
4  mensagem2 = "Texto distribuído \
5  em mais de uma linha"
```

Também podemos distribuir o código em mais linhas com vírgulas, quando estamos armazenando dados em coleções, como listas, tuplas e dicionários, definindo parâmetros e passando argumentos para funções, que veremos mais a diante. Exemplos:

```
1  lista = ['string1',
2           'string2']
3
4  def funcao(param1,
5             param2, param3):
6    print param1, param2, param3
7
8  funcao('string1', 'string2',
9         'string3')
```

Blocos

Python usa espaços em branco e indentação para separar blocos de código, como estruturas de controle e de repetição e na declaração de funções, que veremos a seguir na apostila. O aumento da indentação indica o início de um novo bloco, que termina com diminuição da indentação.

A linha que define a estrutura ou declaração é terminada por dois-pontos e simboliza que as linhas com mais um nível de indentação que seguem fazem parte do seu bloco de código. No exemplo abaixo é demonstrada a definicação de uma função:

```
1  def imprime_dados(nome, idade, genero):
2     print "Nome:", nome
3     print "Idade:", idade
4
5     print "Genero sexual:", genero
6
7  print 42
```

No exemplo anterior, as linhas 2, 3, 4 e 5 fazem parte da função `imprime_dados`, enquanto o `print` da sétima linha não. É importante notar que as linhas 4 e 6 em branco não influenciam no término do bloco, podemos ter linhas em branco dentro de um bloco de

código, e é até conveniente para facilitar sua leitura – o bloco só termina com a diminuição da indentação.

Na maioria das linguagens de programação, a indentação é bastante utilizada para aumentar a legibilidade do código, porém no Python ela é obrigatória, se um erro de indentação estiver presente no código, ocorrerá uma falha na interpretação. O exemplo a seguir demonstra um erro de indentação na linha 3:

```
1  def imprime_dados(nome, idade, genero):
2     print "Nome:", nome
3    print "Idade:", idade
4     print "Genero sexual:", genero
```

Apesar de não recomendado, é possível declarar várias instruções na mesma linha (*inline*) no Python, para isto, o caractere ponto-e-vírgula é utilizado. Exemplo:

```
print "Nome:", nome; print "Idade:", idade
```

O ponto-e-vírgula também pode ser inserido no término de cada linha, porém seu uso é opcional e quase nunca utilizado.

Comentários

Os comentários em uma linguagem de programação são úteis para deixar algum tipo de informação, como lembretes de funcionamento de um processo, erros a serem corrigidos ou código a ser melhorado para a pessoa que for ler e manter o código, este sendo possivelmente até mesmo o programador que escreveu o código.

O caractere # marca o início dos comentários, a partir deste caractere todo texto é ignorado pelo interpretador até o final da linha - com exceção dos comentários funcionais ou "mágicos", que serão exibidos a seguir. Exemplos de comentários não-funcionais:

```
1  # Linha inteira comentada
2  conexao = None        # Variavel instanciada no metodo conecta()
3  smtp_server = "smtp.magrathea.com"  # Servidor de email
```

Os comentários funcionais são interpretados pelo interpretador do Python e possuem duas utilidades:

- Alterar a codificação do arquivo do código-fonte do programa, onde é acrescentado o comentário `# -*- coding: <encoding> -*-` no início do arquivo, onde <encoding> é a codificação do arquivo, geralmente utilizado *utf-8* ou *latin-1* para caracteres que não fazem parte da lingua inglesa, como caracteres latinos;
- Indicar o local do interpretador que será utilizado para executar o programa, onde é acrescentado o comentário `#!` seguido do caminho do interpretador, algo como

`#!/usr/bin/python` e é utilizado somente em sistemas **NIX like*, como o Linux, FreeBSD, OpenBSD, entre outros.

Os dois comentários funcionais explicados anteriormente podem existir em conjunto, exemplo:

```
1  #!/usr/bin/python
2  # -*- coding: latin-1 -*-
```

Atenção

Apesar de nessa apostila não definirmos a codificação nas listagens de códigos-fontes a fim de reduzir o número de linhas, não se esqueça de definir a codificação nos seus arquivos, ou os caracteres latinos neles presentes aparecerão incompreensíveis, na melhor das hipóteses. Veremos mais sobre tipos de caracteres no capítulo seguinte, quando falaremos sobre Unicode.

Identificadores

Identificadores são nomes utilizados para identificar objetos - variáveis, funções e classes, por exemplo. Os identificadores devem começar com uma letra sem acentuação ou com um sublinhado (_), e podem conter números, letras sem acentuação e sublinhados. Python é *case-sensitive*, ou seja, o identificador `python` é diferente dos identificadores `PYTHON` e `Python`.

Variáveis e tipos

Variáveis são espaços reservados na memória utilizados para armazenar valores, como por exemplo, textos, resultados de cálculos, entrada de usuário, resultados de consultas a uma base de dados, etc. Variáveis devem seguir as regras de identificadores, vistos anteriormente.

Tipos de dados

Tipos de dados restringem os valores a serem armazenados nas variáveis. Os tipos pré-definidos no Python podem ser simples - como números – e também os tipos que funcionam como coleções, como as listas e dicionários. Como dito anteriormente, Python possui tipagem dinâmica, o que associa o valor da variável a um tipo no momento da atribuição de valor e aloca o espaço necessário para armazenar seus valores. Para atribuir valores às variáveis é utilizado o operador de atribuição (=). Exemplos:

```
1  nome = "Zaphod"                    # Uma string
2  resposta = 42                      # Um inteiro
```

No exemplo acima a variável `nome` tem o valor `"Zaphod"` atribuído, assim como a variável `resposta` possui o valor `42`. Pode-se também atribuir vários valores de uma só vez às variáveis, por exemplo:

```
x, y, z = 10, -5, 7              # Igual a x = 10, y = -5 e z = 7
```

Os tipos em Python são divididos em duas categorias:

- Mutáveis – que permitem que seus valores sejam alterados;
- Imutáveis – que não permitem que seus valores sejam alterados, ou seja, ao re-atribuir um valor a uma variável, este não é alterado, o valor a ser atribuído é um novo valor.

Tipos Numéricos

Tipos de dados numéricos em Python são imutáveis e se dividem basicamente em quatro, além do `bool`, que é uma extensão do tipo `int`:

Int

Representa números inteiros, como 42, 0, 1 e -3, entre -2147483648 a 2147483647. Ocupa 4 bytes na memória e pode armazenar tanto valores decimais (de base 10), quanto valores octais (de base 8) e hexadecimais (base 16). Para declarar um inteiro octal, o número 0

(zero) tem que ser prefixado ao número, como 0123, e para definir um número hexadecimal, o prefixo 0x ou 0X deve ser utilizado, como 0xFFFFFF ou 0X006699.

Long

Representa números inteiros longos e pode armazenar números tão grandes quanto a memória puder armazenar. Assim como o tipo `int`, o `long` também pode armazenar números tanto decimais, quanto octais e hexadecimais. Para declarar um valor `long` mesmo se o número a ser atribuído estiver na faixa de valores do tipo `int` é necessário sufixar a letra L – minúscula ou maiúscula - como em 524511574362l, 0xDDEFBDAEFBDAECBFBAEL e 0122L. Como a letra L minúscula pode ser confundida facilmente com o número 1, é uma boa prática utilizar o L sempre maiúsculo ao invés de minúsculo.

Float

Representa números reais e que possuem sinal de expoente (e ou E). Esses números são comumente chamados comumente de *floating-point numbers* ou números de ponto flutuante. Exemplos: 0.0042, .005 (o mesmo que 0.005), 1.14159265 e 6.02e23 (o mesmo que 6.02^{23}). Note que o delimitador de casas decimais é o ponto (.) e não a vírgula (,) padrão no Brasil.

Complex

Representa números complexos, como 0.42j e 9.322e-36j.

Bool

O tipo `bool` foi adicionado na versão 2.2 do Python como uma especialização do tipo `int`. Os valores do tipo `bool` podem representar dois valores completamente distintos: `True` (igual ao `int` 1) e `False` (igual ao `int` 0) para, respectivamente, verdadeiro e falso. Exemplos:

```
1  possui_filhos = False              # OK
2  e_funcionario = false              # Erro
3  sair = True                        # OK
4  valida_entrada = true              # Erro
```

Como Python é *case-sensitive*, os valores `false` e `true` atribuídos às variáveis nas linhas 2 e 4 geram erro de interpretação, por isso, redobre atenção se você veio de linguagens como C/C++, Java ou C#.

NoneType

`NoneType` é o tipo de `None`, uma constante embutida do Python, assim como `True` e `False`, e é frequentemente utilizada para representar a ausência de um valor, similar a `null` na linguagem C e derivadas.

Tipos de Sequências

List

Uma lista é um conjunto linear de valores indexados por um número inteiro, chamado de índice, que se inicia em 0 (zero). Os elementos contidos em uma lista podem ser de qualquer tipo, até mesmo outras listas e não precisam ser todos do mesmo tipo. Uma lista é delimitada por colchetes, e seus elementos separados por vírgulas. Exemplos:

```
1  lista1 = []
2  lista2 = [1, 2, 3]
3  lista3 = ["string 1", "string 2"]
4  lista4 = [1, 3.1415, "string", [1, 2], (4, 5)]
```

Nos exemplos anteriores, `lista1` é uma lista iniciada sem elementos. Estes podem ser adicionados ou inseridos na lista através se seus métodos, que serão vistos mais adiante. `lista4` possui 5 elementos, sendo um `int`, um `float`, uma `string`, uma outra lista e uma tupla, que será vista a seguir.

Como dito anteriormente, os elementos das listas possuem índices associados. Com esses índices podemos selecionar os elementos. Exemplos:

```
1  >>> lista = [1, 2, 3]
2  >>> lista[0]                # Imprime 1
3  >>> lista[1]                # Imprime 2
```

Usando índices negativos, as posições são acessadas a partir do final da lista. Exemplos:

```
1  print lista3[-1]            # Imprime string 2
2  print lista4[-2]            # Imprime (4, 5)
```

Python fornece uma maneira simples de criar “fatias” (*slices*) de uma lista. Uma fatia nada mais é do que uma lista gerada a partir de um fragmento de outra lista. Para criar um *slice*, basta especificar os dois índices separados pelo sinal de dois-pontos (:) - o fragmento resultante contém os elementos correspondentes do primeiro índice ao segundo, não incluindo o último elemento. Se omitirmos um dos índices, assume-se início ou fim da lista. Exemplos:

```
1  lista = [1, 2, 3, 4, 5]
2  print lista[0:2]                    # Imprime [1, 2]
3  print lista[2:4]                    # Imprime [3, 4]
4  lista[:]                            # Imprime [1, 2, 3, 4, 5]
```

É possível também fornecer o intervalo entre os elementos, e caso este não seja especificado (como nos exemplos anteriores) o valor padrão é 1. Exemplos:

```
1  lista = [1, 2, 3, 4, 5]
2  print lista[0:2]                    # Imprime [1, 2]
3  print lista[2:4]                    # Imprime [3, 4]
4  lista[:]                            # Imprime [1, 2, 3, 4, 5]
```

Repare que na linha 3 não especificamos nem o início, nem o final do *slice*, fornecemos somente o intervalo com o valor -1, que fez com que a lista seja gerada e exibida do último para o primeiro elemento.

Operações

Vimos que tudo em Python é um objeto, listas são objetos também e possuem métodos (funções) para manipulá-las. Abaixo segue uma relação dos métodos mais comuns e alguns exemplos:

`count(elemento)`: Retorna quantas ocorrências de um elemento existe na lista.

Exemplos:

```
1  >>> lista = [1, 2, 3, 2]
2  >>> lista.count(2)
3  2
4  >>> lista.count(4)
5  0
```

`index(elemento)`: Retorna primeiro índice de elemento na lista. Aciona exceção `ValueError` caso valor não exista.

Exemplo:

```
1  >>> lista = [1, 2, 3, 2]
2  >>> lista.count(2)
3  2
```

`append(elemento)`: Insere elemento no final da lista.

Exemplo:

```
1  >>> lista = [1, 2, 3]
2  >>> lista.append(4)
3  >>> lista
4  [1, 2, 3, 4]
```

`insert(index, elemento)`: Insere elemento no índice passado por argumento.

Exemplo:

```
>>> lista = [1, 2, 3]
>>> lista.insert(0, -1)
>>> lista
[-1, 1, 2, 3]
```

`remove(elemento)`: Remove primeira ocorrência do valor especificado por argumento.

Exemplo:

```
>>> lista = [1, 2, 3, 1, 2, 3]
>>> lista.remove(1)
>>> lista
[2, 3, 1, 2, 3]
```

`pop()`: Remove elemento da lista e o retorna. O método `pop` aceita o argumento `index`, caso um índice não seja especificado, o último elemento da lista é removido.

Exemplos:

```
>>> lista = [1, 2, 3, 4, 5]
>>> lista.pop()         # Remove e retorna elemento 5
5
>>> lista.pop(0)        # Remove e retorna elemento 1
1
>>> lista
[2, 3, 4]
```

`sort()`: Ordena a lista. O método `sort` aceita o parâmetro `reverse`, que possibilita que a ordenação seja crescente ou decrescente. Caso esse argumento não seja especificado a ordenação é feita em ordem crescente.

Exemplos:

```
>>> lista = [5, 2, 1, 3, 4]
>>> lista.sort()
>>> lista
[1, 2, 3, 4, 5]
>>> lista.sort(reverse=True)
>>> lista
[5, 4, 3, 2, 1]
```

Tuple

Uma tupla consiste de um número de valores separados por vírgulas. Tuplas ao contrário das listas que acabamos de ver são imutáveis - uma vez criadas não podem ser modificadas. Exemplos:

```
>>> t1 = 1, 2, 3, 4, 5
>>> t1
(1, 2, 3, 4, 5)
>>> t1[0]
1
>>> t1[2:4]
(3, 4)
>>> t1[::-1]
(5, 4, 3, 2, 1)
```

```
10 >>> t2 = (1, 3, 3)
11 >>> t2[1] = 2
12 Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Como você pode ver nos exemplos acima, as tuplas impressas aparecem sempre entre parênteses, de forma que tuplas aninhadas sejam interpretadas corretamente. Podemos definir uma tupla com ou sem os parênteses, embora muitas vezes estes sejam necessários, caso a tupla faça parte de uma expressão maior ou provoque ambiguidade. Repare também nos exemplos acima que tuplas possuem as mesmas funcionalidades de indexação e *slicing* que as listas vistas anteriormente.

O que fizemos na linha 1 se chama *tuple packing*, oque acontece quando agrupamos dois ou mais valores em uma tupla. A operação inversa, chamada de *sequence unpacking*, também é possível, dessa forma:

```
1  >>> tupla = 1, 2
2  >>> x, y = tupla
3  >>> x
4  1
5  >>> y
6  2
```

Note que a operação de atribuição múltipla vista logo no início do capítulo nada mais é do que uma combinação de *tuple packing* e *sequence unpacking*. Recordando:

```
x, y, z = 10, -5, 7            # Igual a x = 10, y = -5 e z = 7
```

Tuplas podem ter vários usos, como coordenadas num eixo cartesiano (x, y), registros de uma consulta a um banco de dados e onde fizer sentido ou for mais eficiente um conjunto fixo de valores.

'Tips and tricks' de tuplas

Há uma peculiaridade na construção de tuplas vazias ou com um só ítem. Tuplas vazias são construídas por um par vazio de parênteses, dessa forma:

```
>>> t = ()
```

Tuplas com somente um elemento devem ser construídas adicionando uma vírgula após seu valor. Abra o interpretador interativo do Python e dispare as seguintes instruções:

```
1  >>> x = (1)
2  >>> type(x)
3  <type 'int'>
4  >>> y = (1,)
5  >>> type(y)
```

```
6  <type 'tuple'>
```

Repare que ao passarmos a variável `x` para a função embutida `type` – que retorna o tipo de um objeto - o texto `<type 'int'>` é exibido, mostrando que `x` não é uma tupla, mas um inteiro. Já com `y` não ocorre o mesmo, já que inserimos a vírgula após o único elemento da tupla.

Isso ocorre porque o Python utiliza parênteses também para agrupar expressões, como em `2 * (42 + y)`, portanto temos que inserir a vírgula para não haver ambiguidade entre tuplas e expressões.

Operações

Tuplas possuem dois métodos idênticos ao das listas: `index` e `count`. O restante dos métodos de listas realiza modificações nos valores, e como tuplas são imutáveis, não suportam tais operações. Mais infomações sobre tuplas consulte a documentação oficial da linguagem, no endereço http://docs.python.org/tutorial/datastructures.html#tuples-and-sequences.

Dict

Um dicionário é uma coleção de elementos onde é possível utilizar como índice qualquer tipo imutável, como strings, tuplas e tipos numéricos. O termo normalmente utilizado para descrever essa associação entre índice e elemento é *key* / *value* ou chave / valor.

Dicionários[2] são definidos entre chaves ({ e }), as *keys* separadas dos *values* por dois-pontos (:) e os pares de *keys* e *values* separados por vírgulas. Exemplos:

```
1  >>> dados = {"nome": "Ford", "idade": 42}
2  >>> print "Nome:", dados["nome"], " Idade: ", dados["idade"]
3  Nome: Ford, Idade: 42
```

Podemos também adicionar um valor em um dicionário já definido por meio de atribuição, e remover com a instrução `del`. Exemplos:

```
1  >>> dados["telefone"] = "0000-9999"        # Adiciona valor
2  >>> del dados["idade"]                     # Remove valor
```

A instrução `del` apaga uma referência a um objeto. Veremos mais sobre o assunto no final da apostila, em Orientação a Objetos.

Operações

`keys()`: Retorna uma lista das chaves de um dicionário.

[2] Caso você tenha alguma experiência com outras linguagens de programação, é possivel que você já conheça o conceito de dicionários porém com outro nome, como Hash e Map, por exemplo.

Exemplo:

```
>>> dados = {"nome": "Arthur", "idade": 42}
>>> dados.keys()
['idade', 'nome']
```

`values()`: Retorna uma lista dos valores de um dicionário.

Exemplo:

```
>>> dados = {"nome": "Arthur", "idade": 42}
>>> dados.values()
[42, 'Arthur]
```

`items()`: Retorna uma lista contendo os pares de chaves e valores em formas de tuplas.

Exemplo:

```
>>> dados = {"nome": "Arthur", "idade": 42}
>>> dados.items()
[('idade', 42), ('nome', 'Arthur')]
```

`has_key(chave)`: Retorna `True` caso chave exista em um dicionário e `False` caso contrário.

Exemplo:

```
>>> dados = {"nome": "Arthur", "idade": 42}
>>> dados.has_key("idade")
True
```

Strings

Strings são seqüências de caracteres, sejam eles letras, números, espaços, caracteres especiais, e etc., e assim como os tipos numéricos vistos anteriormente, strings também são imutáveis. Strings podem ser delimitadas por aspas simples, duplas ou triplas, sendo que estas devem ser idênticas na abertura e fechamento, ou seja, quando se abre uma string com aspas duplas, deve-se fechar com aspas duplas. Exemplos:

```
>>> string1 = "Python"                    # Aspas duplas
>>> string2 = 'Python'                    # Aspas simples
>>> string3 = """Python"""                # Aspas triplas
>>> string4 = '''Python'''                # Aspas triplas
```

Strings podem ser unidas (concatenadas) utilizando o operador de adição / concatenação (+). Exemplos:

```
>>> string1 = "Python, "
>>> string2 = 'Batteries included!'
>>> string3 = string1 + string2
>>> string4 = 'Python ' + "Batteries included!"

>>> string1                # Python
>>> string2                # Batteries included!
>>> string3                # Python, Batteries included!
>>> string4                # Python, Batteries included!
```

```
>>> string1 + string2        # Python, Batteries included!
```

Seleção e slicing

Assim como listas e tuplas, strings também são sequências, portanto, podemos utilizar as mesmas funcionalidades de indexação e *slicing* de elementos vistos nos outros tipos seqüenciais vistos anteriormente. Exemplo:

```
>>> string = "Python"
>>> string[0]
P
>>> string[1:3]
yt
>>> string[3:]
hon
>>> "Batteries included!"[0:9]
Batteries
```

Sequências de escape

Como vimos, definimos strings envolvendo uma sequência de caracteres entre aspas. Também vimos na introdução da apostila que o caractere barra-invertida (\ ) tem a função de separar uma seqüência em várias linhas, portanto, as aspas e a barra-invertida possuem funções especiais no Python, mas e se precisarmos, por exemplo, inserir aspas ou barras-invertidas em uma string? Para isso precisamos 'escapar' esses caracteres. Sequências de escape no Python consistem de uma barra-invertida (\ ) seguida de uma letra, sinal ou combinações de dígitos. Alguns das sequências de escape que Python dispõe são:

- `\n` Nova linha. Move o cursor para o início da próxima linha
- `\t` Tabulação horizontal
- `\r` Retorno de carro. Move o cursor para o início da linha atual
- `\b` *Backspace*. Retrocede o cursor um caractere
- `\a` Alerta. Emite um alerta (*beep*)
- `\\` Barra invertida. Escapa o caractere \
- `\"` Aspas duplas. Escapa o caractere "
- `\'` Aspa simples. Escapa o caractere '

Exemplos:

```
>>> print "\"Aspas duplas em strings\""
"Aspas duplas em strings"
>>> print '"Aspas duplas em strings"'
"Aspas duplas em strings"
```

```
5  >>> print "'Aspas simples'"
6  'Aspas simples'
7  >>> print '\'Aspas simples\''
8  'Aspas simples'
9  >>> print "Mod.:\t301"
10 Mod.:    301
11 >>> print 'C:\\Python27'
12 C:\Python27
13 >>> print "Mod.:\t301\nCordas:\t6"
14 Mod.:    n301
15 Cordas:    6
```

Repare nos exemplos anteriores que podemos inserir aspas-duplas em strings delimitadas por aspas-simples e vice-versa. Na linha 6 escapamos o caractere barra-invertida, comumente utilizado para expressar um caminho na árvore de diretórios do Windows[3] ou para definir expressões regulares. Nesses casos podemos definir uma string como sendo "crua", chamada comumente de *raw string*, bastando preceder a seqüência pela letra "r", maiúscula ou minúscula. Fazendo isso, todas as barras-invertidas são deixadas na string sem serem interpretadas. Exemplos:

```
1  >>> print r"\n"                    # Imprime \n
2  >>> regex = r"[a-zA-Z0-9_.-]{3,}@([a-zA-Z0-9_-]{3,}\.)+[a-zA-Z0-
9]+"
3  >>> dir = R"C:\Arquivos de programas"
```

As aspas triplas são muito úteis para criar strings com várias linhas, assim, as linhas podem ser quebradas diretamente sem o uso do caractere de escape "\n". Exemplo:

```
1  mensagem = """\
2  ls [-opcoes]
3      -l    Exibe permissões, data de criação e tipo do arquivo
4      -a    Exibe inclusive arquivos ocultos"""
```

Na linha 1 utilizamos uma barra-invertida depois das aspas para dispor a sentença em outra linha somente para melhorar o alinhamento no código, podemos igualmente definir a string dessa forma:

```
1  mensagem = """ls [-opcoes]
2      -l    Exibe permissões, data de criação e tipo do arquivo
3      -a    Exibe inclusive arquivos ocultos"""
```

Ambos resultam a mesma string:

```
ls [-opcoes]
    -l    Exibe permissões, data de criação e tipo do arquivo
    -a    Exibe inclusive arquivos ocultos
```

[3] Como veremos mais adiante, em Trabalhando com Arquivos, sempre utilizamos o caractere / (barra) para definir um caminho em uma árvore de diretórios, independente do sistema operacional a ser executado.

Formatação (ou interpolação) de strings

Podemos formatar strings em Python pelo menos de três maneiras:

- Por posição, com o operador de interpolação `%` associado de um símbolo de formatação;
- Por palavra-chave, utilizando *placeholders* e um dicionário;
- Através do método `format`, disponível a partir da versão 2.6 do Python.

Interpolação por posição

O primeiro método é um dos mais populares, até mesmo porque Python o pegou "emprestado" da linguagem C, e é utilizado em diversas linguagens baseadas em C/C++.

Como vimos, para formatar strings dessa maneira utilizamos o operador `%` associado de um dos símbolos existentes no Python, que são:

- `%s` String (converte objeto utilizando a função `str`)
- `%r` String (converte objeto utilizando a função `repr`)
- `%i` ou `%d` Números inteiros
- `%o` Números octais
- `%h` Números hexadecimais
- `%f` Números reais (ou *floats*)
- `%e` Números reais exponenciais
- `%%` Sinal de porcentagem

Exemplos:

```
1  >>> nome = "Zaphod"
2  >>> idade = 42
3  >>> mensagem = "Nome: %s     Idade: %d" % (nome, idade)
4  >>> print mensagem
5  Nome: Zaphod     Idade: 42
```

Repare que passamos as variáveis `nome` e `idade` para o operador `%` através de uma tupla depois do fechamento das aspas e que não omitimos os parênteses; caso isso ocorra, uma exceção `TypeError` é acionada.

As funções `str` e `repr`, utilizadas pelos símbolos de formatação `%s` e `%r`, são utilizadas para representar objetos, `str` para uma representação mais amigável, e `repr` para uma representação mais específica do objeto. Veremos mais sobre as funções `str` e `repr` mais a diante, em Orientação a Objetos.

Como Python é uma linguagem fortemente tipada, para concatenar uma string com um `int`, primeiro teríamos que converter o valor da variável `idade` em string e então concatenar os valores, dessa forma:

```
1  mensagem = "Nome: " + nome + "Idade: " + str(idade)
```

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

```
1  >>> euler = 2.7313
2  >>> pi = 3.14159265
3  >>> resp = 42
4  >>> print """\
5  ... Pi: %10.3f.
6  ... Euler: %.7f.
7  ... Resposta: %3d""" % (pi, euler, resp)
8  Pi:      3.142.
9  Euler: 2.7313000.
10 Resposta:  42
```

Chamamos esse método de formatação por posição, pois se mudarmos a posição dos os valores na tupla, por exemplo, inverter as variáveis `pi` e `euler`, que são do mesmo tipo, a string formatada será diferente.

Formatação por palavra-chave

Esse segundo método é bastante semelhante ao primeiro. Exemplo:

```
1  >>> mensagem = 'Nome: %(nome)s. Idade: %(idade)d.' % {'nome':
'Zaphod', 'idade': 42}
2  >>> print mensagem # Imprime Nome: Zaphod. Idade: 42.
```

Repare que ao invés de passarmos os valores através de uma tupla para o operador `%`, utilizamos um dicionário, onde a chave corresponde ao *placeholder* na string a ser formatada, e que os símbolos utilizados para formatar strings no primeiro método também devem ser utilizados nesse, após o fechamento dos parênteses.

A formatação de tipos numéricos através dos modificadores `m.n` também funciona da mesma maneira que no método de formatação por posição que vimos anteriormente. Exemplos:

```
1  >>> euler = 2.7313
2  >>> pi = 3.14159265
3  >>> resp = 42
```

```
>>> print """\
... Pi: %(pi)10.3f.
... Euler: %(euler).7f.
... Resposta: %(resposta)3d""" % {'euler': euler, 'resposta': resp,
'pi': pi}
Pi:       3.142.
Euler: 2.7313000.
Resposta:  42
```

Método format

Como foi dito anteriormente, outra maneira de formatação foi disponibilizada na versão 2.6 do Python, o método `format` dos objetos string. Esse método é o novo padrão de formatação de strings no Python 3.0, e consiste em definir *placeholders*, que podem ser tanto numéricos (por posição) quanto por palavra-chave, entre um par de chaves ({ }). Veja alguns exemplos:

```
>>> string1 = "{0} + {1} = {2}"
>>> string1.format(1, 1, 2)
'1 + 1 = 2'
>>> string2 = "Nome: {nome}. Idade {idade}"
>>> string2.format(nome='Zaphod', idade=42)
'Nome: Zaphod. Idade 42'
```

No primeiro exemplo definimos os *placeholders* utilizando índices numéricos e passamos os valores para o método `format` como argumentos por posição. No segundo exemplo definimos os *placeholders* utilizando palavras-chave e fornecemos os valores para o método `format` como argumentos por palavras-chave [4].

Assim como nos métodos que vimos anteriormente, podemos controlar como os valores que irão substituir os *placeholders* serão exibidos, mas de uma maneira diferente: definimos os símbolos de formatação após o índice numérico/*keyword*, precedido por um dois-pontos (:). Exemplo:

```
"""\
{saudacao}!
Agora sao {hora:02d}:{minuto:03d}""".format(saudacao='Bom dia',
hora=3, minuto=5)
```

Diferentemente dos métodos de formatação anteriores, não precisamos obrigatoriamente inserir o símbolo que corresponde ao tipo do valor que irá substituir cada *placeholder* na string, os tipos possuem símbolos de formatação *default*.

No primeiro exemplo que vimos, `nome` é uma string, cujo símbolo é `s`, e caso não definimos explicitamente seu símbolo, `s` é utilizado implicitamente; o mesmo ocorre com

[4] Você deve estar se perguntando sobre o que são esses tais de argumentos por posição e por palavras-chave que falamos tanto agora, não é?! Não se preocupe, teremos um tópico só deles mais a diante, em Funções.

`idade`, um `int`, cujo símbolo *default* é `d`. Esse novo método de formatação é mais completo que os anteriores e inclui mais simbolos de formatação.

Para strings:

- `'s'` String – *default* para strings

Para inteiros:

- `'d'` Números inteiros decimais (base 10) – *default* para inteiros
- `'b'` Número inteiro binário (base 2)
- `'o'` Números inteiros octais (base 8)
- `'x'` Números inteiros hexadecimais (base 16) utilizando letras minúsculas para dígitos superiores a 9 (de A a F)
- `'X'` Números inteiros hexadecimais (base 16) utilizando letras maiúsculas para dígitos superiores a 9
- `'n'` Número. O mesmo que `'d'`, exceto que utiliza as configurações de localização para inserir os separadores apropriados de casas decimais
- `'c'` Caracter. Converte o inteiro para o caractere Unicode correspondente antes de imprimir.

Para floats e decimais

- `'e'` Notação exponencial. Imprime o número em notação científica utilizando a letra ‘e’ para indicar o expoente
- `'E'` Notação exponencial. O mesmo que ‘e’ exceto que utiliza a letra ‘E’ para indicar o expoente
- `'f'` ou `'F'` Números reais (ou *floats*)
- `'%'` Porcentagem. Multiplica o número por 100 e exibe no formato `'f'` seguido de um sinal de porcentagem.

Há mais símbolos e opções de formatação disponíveis nesse método, como o preenchimento de espaços vazios, adição de sinal (+) e (-) a números, controle de precisão e etc. Consulte a documentação oficial da linguagem para mais informações em http://docs.python.org/library/stdtypes.html#string-formatting-operations.

Operações

`capitalize()`: Retorna uma cópia da string com o primeiro caractere maiúsculo.

Exemplo:

```
1  string = "python"
2  print string.capitalize()     # Imprime Python
```

`upper()`: Retorna uma cópia da string com todos os caracteres maiúsculos.

Exemplo:

```
1  string = "python"
2  print string.upper()     # Imprime PYTHON
```

`lower()`: Retorna uma cópia da string com todos os caracteres minúsculos.

Exemplo:

```
1  string = "PYTHON"
2  print string.lower()     # Imprime python
```

`strip()`: Retorna uma cópia da string com os espaços em branco antes e depois removidos.

Exemplo:

```
1  string = "    Python    "
2  print string.strip() + "!!!"     # Imprime Python!!!
```

`startswith(prefixo)`: Retorna True se uma string começa com o prefixo passado por argumento, e False caso contrário.

Exemplo:

```
1  string = "Python"
2  print string.startswith("py")     # Imprime False
```

Note que, como strings são imutáveis, os métodos que aparentemente possam modificar seus valores, como `capitalize()` e `lower()`, somente fazem uma cópia do valor atual da string, tranformam-no e o retornam modificado, deixando seu valor inalterado. Para que a string seja 'modificada', precisamos atribuir o valor retornado pelo método. Exemplo:

```
1  string = "PYTHON"
2  string = string.lower()
```

Há muitos outros métodos disponíveis em objetos strings, listamos somente alguns deles. Para mais informações consulte a documentação oficial no endereço http://docs.python.org/library/stdtypes.html#string-methods.

Unicode

Se você reparar nas strings definidas até este ponto da apostila, irá notar que até agora não utilizamos nem caracteres acentuados, nem o caractere ce-cedilha (ç). Vamos fazer um teste, abra o IDLE ou outro editor ou IDE de sua preferência, crie um arquivo e entre com as seguintes instruções:

```
1  print 'Pão com maçã'
2  raw_input('')
```

Salve esse arquivo e o execute (clicando duas vezes sobre seu ícone ou através da linha de comando, navegando até o local onde o arquivo foi salvo e executando o seguinte comando no terminal ou *prompt* de comando, caso esteja no Windows[5]):

```
python nome_do_arquivo.py
```

A seguinte mensagem aparecerá logo em seguida:

```
C:\ > python teste.py
  File "teste.py", line 1
SyntaxError: Non-ASCII character '\xe3' in file teste.py on line 1,
but no encoding declared; see http://www.python.org/peps/pep-0263.html
for details
```

O que diz na mensagem acima é que inserimos um caractere não-ASCII no nosso código-fonte e não declaramos uma codificação no arquivo. Para resolver esse problema e podermos lidar com quaisquer caracteres no nosso código-fonte, temos que fazer duas coisas, 1) declarar o *encoding* do arquivo como `latin-1` (o mesmo que `iso-8859-1`) ou `utf-8` e; 2) definir nossos textos como Unicode, prefixando o caractere `u` antes das aspas, assim:

```
1  -*- coding: latin-1 -*-
2  print u'Pão com maçã'
```

Explicando brevemente, Unicode é um padrão que permite aos computadores representar e manipular de maneira consistente, textos de quaisquer sistemas de escritas existentes. Você pode obter mais informações sobre o Unicode em http://www.unicode.org/, http://pt.wikipedia.org/wiki/Unicode e http://docs.python.org/howto/unicode.html.

Os métodos disponíveis em strings, como `replace`, `upper`, `lower`, `capitalize`, `format`, dentre outros, além dos recursos de formatação, *slicing* e indexação, também estão disponíveis para sequências Unicode.

Determinando o tipo de uma variável

A função embutida `type` é utilizada para descobrir o tipo de uma variável. Exemplos:

```
1  >>> a = 1
2  >>> type(a)
3  <type 'int'>
4  >>> b = "Python"
5  >>> type(b)
6  <type 'str'>
7  >>> c = [1, 2, 3]
8  >>> type(c)
```

[5] Isso só vai funcionar no Windows se você configurou o a variável de ambiente PATH, conforme é explicado no Apêndice A.

```
9  <type 'list'>
10 >>> d = (1, 2, 3)
11 >>> type(d)
12 <type 'tuple'>
13 type(a) == type(b)
14 True
15 type(b) == int
16 True
```

Determinando o tamanho de uma variável

A função `len` retorna o tamanho de uma variável dependendo do seu tipo. Exemplos:

```
1  >>> a = "Python"
2  >>> len(a)
3  6
4  >>> b = [1, 2, 3]
5  >>> len(b)
6  3
7  >>> c = {"a": 1, "b": 2}
8  >>> len(c)
9  2
```

Quando a função `len` recebe uma string, retorna a quantidade de caracteres; quando recebe uma lista, tupla ou dicionário, retorna a quantidade de elementos, e assim por diante. As funções `len` e `type` vistas anteriormente representam funções de reflexão ou introspecção e veremos mais delas no decorrer da apostila.

Conversões

Python disponibiliza algumas funções para que possamos converter um valor de um tipo em outro. Exemplos:

```
1  >>> a = "1"
2  >>> b = int(a)
3  >>> type(b)
4  <type 'int'>
5  >>> c = 1
6  >>> d = float(c)
7  >>> type(d)
8  <type 'float'>
9  >>> e = [1, 2, 3]
10 >>> f = tuple(e)
11 >>> type(f)
12 <type 'tuple'>
13 >>> g = "False"
14 >>> h = bool(g)
15 >>> type(h)
16 <type 'bool'>
```

Há obviamente algumas conversões impossíveis de realizar, como uma string "Python" em `int`, ou `float`, por exemplo. Quando uma conversão resulta em um erro, uma exceção `ValueError` é acionada.

Além dos tipos demonstrados anteriormente, Python dispõe ainda de muitos outros, como as coleções `set` e `frozenset`, não mostradas nessa apostila. Para mais informações consulte a documentação oficial no endereço http://docs.python.org/library/stdtypes.html.

Avaliação de True e False

Qualquer objeto em Python pode ser avaliado como `True` e `False` para uso em estruturas como o `if`, `for` e `while`, que veremos mais adiante ou como operandos de operadores lógicos, que serão vistos no próximo capítulo.

Todos os seguintes valores são avaliados como `False`:

- `False`
- `None`
- Zero (0) de quaisquer tipos numéricos como, `0`, `0L`, `0.0` e `0j`.
- Qualquer sequência vazia, como, `''`, `()`, `[]`.
- Qualquer dicionário vazio, como `{}`.

Todos os valores não relacionados na listagem anterior são avaliados como `True`.

Operadores

Operadores Aritméticos

Há sete operadores aritméticos no Python:

- + Soma
- - Subtração
- * Multiplicação
- / Divisão
- // Divisão truncada
- % Módulo
- ** Exponenciação

O interpretador interativo do Python também pode ser utilizado como uma calculadora, digitando os cálculos e recebendo os valores. Exemplos:

```
1  >>> 2 + 2                  # Imprime 4
2  >>> 5 / 2                  # Imprime 2
3  >>> 10 % 5                 # Imprime 0
4  >>> 12.356 / 2             # Imprime 6.1779999999999999
5  >>> 5 * 5                  # Imprime 25
6  >>> 2 ** 3                 # Imprime 8
7  >>> 5 - 2                  # Imprime 3
8  >>> 12.356 // 2            # Imprime 6.0
```

Explicando:

- Linha 1: Uma soma de 2 com 2;
- Linha 2: Uma divisão de 5 por 2 resultando em 2. Em Python, assim como em outras linguagens, o resultado de uma divisão entre dois inteiros será sempre um inteiro, portanto, o resultado de 5 / 2 é 2 e não 2.5;
- Linha 3: O operador % (módulo) é utilizado para obter o resto da divisão de dois números inteiros, no caso, a divisão de 10 por 5 resulta em 2 e o resto é 0;
- Linha 4: A solução para o problema da linha 2: dividir 12.356 (um número real), por 2 – um inteiro, resulta em outro número real;
- Linha 5: Uma multiplicação de 5 por 5;
- Linha 6: O operador ** (exponenciação) eleva o operando da esquerda à potência do operando da direita, portanto 2 elevado à terceira potência é igual a 8;
- Linha 7: Subtrai 2 de 5;
- Linha 8: Divide o número, mas retorna somente parte inteira da divisão.

A exponenciação pode ser feita também por meio da função `pow`. Exemplo:

```
>>> pow(2, 3)              # Imprime 8
>>> pow(5, 2)              # Imprime 25
```

O módulo `math` da biblioteca-padrão do Python implementa outras funções matemáticas, para cálculo de fatorial, seno, co-seno, logaritmo, raiz, etc.

Além dos operadores matemáticos descritos acima, Python possui os operadores compostos de atribuição:

- += Soma e atribui
- -= Subtrai e atribui
- *= Multiplica e atribui
- /= Divide e atribui
- //= Divide e atribui somente a parte inteira do resultado
- %= Divide e atribui resto da divisão
- **= Eleva número e atribui

Exemplos:

```
>>> i = 0     # Atribui 0 para i
>>> i += 10   # Incrementa 10 em i. (i = 10)
>>> i /= 2    # Divide 10 por 2 e atribui para i. (i = 5)
>>> i // 2    # Divide 5 por 2 e atribui parte inteira para i. (i =
2)
>>> i += 8    # Incrementa i em 8. (i = 10)
>>> i -= 4    # Subtrai 4 de i e atribui resultado. (i = 6)
>>> i **= 2   # Eleva 6 ao quadrado e atribui para i. (i = 36)
>>> i %= 10   # Divide 36 por 10 e atribui resto para i. (i = 6)
```

Os operadores acima são muito utilizados em laços de repetição, principalmente o de incremento / concatenação (+=) e decremento (-=). Sem eles, teríamos que atribuir o resultado de uma operação com operadores simples. Abaixo seguem os mesmos exemplos, só que sem os operadores de atribuição:

```
>>> i = 0
>>> i = i + 10
>>> i = i / 2
>>> i = i // 2
>>> i = i + 8
>>> i = i - 4
>>> i = i ** 2
>>> i = i % 10
```

Como vimos anteriormente, operadores de adição podem ser utilizados em strings. Os operadores de multiplicação também podem ser. Exemplos:

```
a = "Python"
print a * 2             # Imprime PythonPython
```

Operadores de adição e multiplicação também podem ser utilizados em listas e tuplas. Exemplos:

```
1  a = [-1, -2]
2  b = [1, 2, 3]
3  print b * 3          # Imprime [1, 2, 3, 1, 2, 3, 1, 2, 3]
4  print a + b          # Imprime [-1, -2, 1, 2, 3]
5  c = (1, 2)
6  print c + (3, 4)     # Imprime (1, 2, 3, 4)
```

Operadores lógicos

Além dos operadores matemáticos, Python possui operadores lógicos, que retornam sempre um valor *booleano*, ou seja, `True` ou `False`. Estes operadores são utilizados com frequência nas estruturas de controle, e laços de repetição, que veremos mais adiante.

Os operadores lógicos em Python são:

- `>` Maior
- `<` Menor
- `>=` Maior ou igual
- `<=` Menor ou igual
- `==` Igual
- `not` Não
- `!=` Não-igual [6]
- `and` E
- `or` Ou

Exemplos:

```
1  >>> print 1 > 2            # False
2  >>> print 2 == 3           # False
3  >>> print 5 == 5           # True
4  >>> print 10 > 1           # True
5  >>> print 2 != 1           # True
6  >>> print 5 <= 5           # True
7  >>> print 5 < 5            # False
8  >>> print 7 >= 3           # True
```

O operador `not` é utilizado para negar (inverter) o resultado de uma operação lógica, por exemplo:

```
print not 5 < 3              # True
```

[6] O operador `!=` pode ser também escrito `<>`, porém está obsoleto e é mantido por questões de compatibilidade com códigos antigos. Novos códigos devem sempre utilizar `!=` ao invés de `<>`.

A expressão acima imprime o inverso do resultado de 5 < 3. Podemos ainda armazenar o resultado de uma expressão *booleana* em uma variável. Exemplos:

```
1  idade = 22
2  maior_idade = idade >= 18
3  senha, senha_digitada = "123", "456"
4  libera_entrada = senha_digitada == senha
5  print libera_entrada            # Imprime False
```

Identidade de objetos

O operador `is` e `is not` são utilizados para comparar a igualdade de dois objetos. A diferença entre o operador `is` e o de igualdade (==) está justamente na diferença que, enquanto o operador == compara os valores armazenados nos dois objetos, o operador `is` compara se as duas referências "apontam" para a mesma referência na memória, isto é, se os dois objetos são iguais. O exemplo a seguir demonstra bem a diferença entre os operadores == e `is`:

```
1  >>> x = [1, 2, 3]
2  >>> y = x
3  >>> z = x[:]
4  >>> x is y
5  True
6  >>> x
7  [1, 2, 3]
8  >>> y
9  [1, 2, 3]
10 >>> z
11 [1, 2, 3]
12 >>> x is z
13 False
14 >>> x == z
15 True
```

No exemplo anterior, `y` aponta para a mesma referência em memória que `x`, enquanto que `z` somente possui o mesmo valor que `x`, não são o mesmo objeto. Ao comparmos `x` com `y` através do operador `is`, este retorna `True`, assim como quando comparamos a variável `x` com `y` através do operador ==. Veremos mais sobre referências de objetos no capítulo de Paradigmas de Programação, em Programação Orientada a Objetos.

Presença em Sequências

Para seqüências, como strings, listas e tuplas, existe o operador `in`, que verifica se um elemento está presente na seqüência e o `not in`, que retorna o inverso do operador `in`. Exemplos:

```
1  print "p" in "python"                    # True
2  print 1 in [1, 2, 3]                    # True
3  print "d" in ("a", "b", "c")            # False
4  print "d" not in ("a", "b", "c")        # True
```

Estruturas

Estrutura de decisão

A função de uma estrutura de decisão é verificar o conteúdo de uma expressão lógica (ou *booleana*) e direcionar o fluxo do programa. Python possui uma estrutura de decisão, o `if`, que é definida da seguinte forma:

```
1  if expressao:
2    expressao_1
3    expressao_2
4  else:
5    expressao_3
6    expressao_4
```

Se o resultado da expressão lógica for `True`, então `expressao_1` e `expressao_2` são executadas, senão, `expressao_3` e `expressao_4` são executadas. O `else` não é obrigatório, porém, se um `if` for declarado sem ele e se a expressão for avaliada como `False`, nada será executado. Exemplo de um `if` com `else`:

```
1  a = 5
2  b = 3
3
4  if a >= b:
5      print "a é maior ou igual a b"
6  else:
7       print "a é menor do que b"
```

No exemplo anterior, se `a` for maior ou igual a `b`, então a mensagem `"a é maior ou igual a b"` é impressa, caso contrário, a mensagem `"a é menor do que b"` é impressa na tela.

Python oferece um adicional à estrutura `if`: o `elif`, que permite realizar outros testes caso a expressão avaliada pelo `if` seja falsa. Exemplo:

```
1   a = 5
2   b = 3
3
4   if a > b:
5       print u"a é maior do que b"
6   elif a == b:
7    print u"a é igual a b"
8   else:
9       print u"a é menor do que b"
```

Pode-se inserir quantos `elif`'s necessários no `if`, contanto que o `else` seja único e o último. Tal recurso adicional do `if` se deve ao fato de Python não possuir a estrutura *switch,* comum às linguagens derivadas da linguagem C, como C++, Java, PHP e C#.

Estruturas de repetição

Estruturas ou *laços* de repetição são construções da linguagem que permitem repetir um bloco de código um número determinado ou indeterminado de vezes. Python possui duas construções, uma adequada para cada caso.

For

O laço `for` é adequado para o primeiro caso, quando se sabe quantas vezes irá repetir um bloco de código. Exemplo:

```
1  for i in range(1, 10):
2    print "i =", i
```

A função `range` chamada na definição do `for`, retorna uma lista contendo uma faixa de valores, no caso, de 1 a 10. A variável `i` é utilizada para iterar na lista retornada por `range`, a cada iteração `i` recebe um valor, primeiro 1, depois 2, e assim por diante até o 10. A função `range` recebe três argumentos – o primeiro é o início do intervalo, o segundo é fim, e o terceiro é o incremento. O incremento padrão é 1, bem como o início, portanto, chamar `range(10)`, `range(1, 10)` ou `range(1, 10, 1)` resulta a mesma lista. Veremos mais sobre argumentos padrão ou *default* na parte de Funções, mais adiante na apostila.

Qualquer objeto iterável pode ser utilizado em um `for`, como uma string, lista ou tupla. Veremos mais sobre iteradores no final da apostila, em Programação Funcional. Exemplo:

```
1  for s in "Python":
2    print s
```

No exemplo anterior os caracteres da string `"Python"` são impressos um por linha com a função `print`.

While

O laço `while` é utilizado quando não sabemos exatamente qual o número de repetições será realizada. Exemplos:

```
1  i = 0
2  while i < 10:
3      print i
4    i = i + 1
```

- Linha 1: Variável `i` utilizada no `while` é inicializada com 0;
- Linha 2: `while` testa se `i` é menor do que 10;
- Linha 3: Caso o teste da linha 2 seja verdadeiro, escreve o valor de `i` na tela;

- Linha 4: Incrementa o valor de `i` em 1;

Após a linha 4 ser executada o foco (cursor) volta para o `while` (linha 1), que testa `i` sucessivamente até que o valor não seja menor do que 10. Repare que o `while` testa uma expressão lógica, vista anteriormente na apostila. Caso esta expressão retorne sempre `True`, o `while` será repetido infinitamente, gerando um *loop* infinito.

Controle de laços

Algumas vezes você vai enfrentar uma situação em que precisa sair de um laço completamente quando uma condição externa é acionada ou quando quer "pular" uma parte do laço e iterar novamente. Python fornece dois comandos para essas situações, o `break` e o `continue`.

Break

Pode ser utilizado tanto em um laço `for` quanto `while`. O comando `break`, assim como na linguagem C e derivadas, "quebra" um laço. Exemplo:

```
1  i = 0
2  while i < 10:
3     if i == 5:
4        break
5     else:
5        print i
6        i += 1
```

No exemplo anterior, quando `i` é igual a 5 o laço `while` termina, assim, só os números 1, 2, 3 e 4 são impressos na tela.

Continue

Também pode ser utilizado tanto no laço `for` quanto `while`. O `continue` faz com que controle passe para o início do laço. Exemplo:

```
1  for letra in 'Python':
2     if letra == 'h':
3        continue
4     print letra
```

Nesse caso, as letras da string `'Python'` são impressas, exceto pela letra `'h'`.

Pass

O `pass`, na verdade, não é um comando de controle de laços (você deve ter percebido, já que dissemos que Python possui dois comandos de controle de laços e não três). O `pass` não faz exatamente nada, e é utilizado quando uma expressão é sintaticamente necessária, mas o programa não precisa de nenhuma ação. Exemplos:

```
1  def teste(*args):
2     pass             # Lembrar de implementar isso depois!!!
3
4  class ClasseVazia:
5     pass
6
7  while True:
8     pass             # Aguarda interrupção por teclado (Ctrl+C)
```

O exemplo da função `teste` mostra que o `pass` pode ser utilizado como um *place-holder*, quando se está trabalhando em um novo código, permitindo assim continuar pensando em um nível mais abstrato e implementar a função depois.

O segundo exemplo demonstra a definição de uma classe vazia. Classes serão explicadas no final da apostila, em Programação Orientada a Objetos

O terceiro exemplo é um `while` infinito que é executado até que o programa seja interrompido por um Ctrl + C.

Funções

O que são?

Funções são blocos de código identificados por um nome, que podem receber valores pré-definidos, chamados de argumentos e retornar valores. As funções no Python são definidas na forma:

```
1  def funcao():
2      expressao_1
3      expressao_2
4      return valor
```

O nome da função deve seguir as normas de identificadores válidos explicados anteriormente na apostila. Funções podem retornar um valor e este deve ser especificado na cláusula `return`, como na linha 4 do exemplo anterior. Exemplo de uma função:

```
1  def imprime_valores():
2     lista = [0, 1, 2, 3]
3     for i in lista:
4        print i
```

Repare que a função anterior não retorna um valor, ela somente imprime todos os números da lista. Se você já teve contato com linguagens de programação como C, C++, C# e Java, deve ter percebido que não foi preciso declarar a função como *void* para especificar que ela não retorna um valor. Exemplo de função que retorna um valor:

```
1  def soma():
2     lista = [0, 1, 2, 3]
3     soma = 0
4     for num in lista:
5        soma += num
6     return soma
7
8  print "Soma =", soma()
```

No exemplo acima, criamos uma função chamada `soma`. Nela, definimos uma lista contendo quatro números inteiros, que são somados e o total retornado pela função.

Parâmetros e argumentos

Para permitir a entrada de valores na função, chamados de argumentos, podemos declarar parâmetros, que são variáveis de entrada de uma função. Abaixo redefinimos a função criada anteriormente, incluindo o parâmetro `lista`:

```
1  def soma(lista):
```

```
2       soma = 0
3       for num in lista:
4           soma += num
5       return soma
6
7   print "Soma =", soma([0, 1, 2, 3])
```

A função acima retorna a soma de todos os elementos de uma lista, agora, passada por argumento na linha 7.

Atenção

É comum os estudantes iniciantes de programação acharem que os termos 'parâmetro' e 'argumento' designam a mesma coisa, porém são termos distintos. Parâmetro é a variável que você define na assinatura da função, entre parênteses. Argumento é o valor que você passa para um parâmetro. Não é possível passar um argumento para uma função que não tem um parâmetro definido.

Parâmetros com argumentos *default*

Muitas vezes é necessário criar uma função quase idêntica a outra, a única diferença é que uma recebe argumentos e a outra não. A função abaixo, por exemplo, retorna um intervalo de números, dados o início, fim e o incremento:

```
1   def intervalo(inicio, fim, incremento):
2       retorno = []
3       i = inicio
4       while i <= fim:
5           retorno.append(i)
6           i += incremento
7       return retorno
8
9   print intervalo(1, 10, 2)          # Imprime [1, 3, 5, 7, 9]
```

E se pudéssemos, por exemplo, especificar que quando um valor não fosse passado para `inicio` e `fim`, os valores para esses sejam ambos iguais a 1, assim como a função `range`, utilizada na estrutura `for` no capítulo passado? Assim, se precisássemos um intervalo de valores entre 1 e 10, chamaríamos apenas `intervalo(10)`. Para isso Python oferece um recurso interessante: os argumentos padrão ou *default* fazem um parâmetro se tornar "opcional". Para especificar um valor padrão para um parâmetro, basta atribuí-lo na lista de parâmetros. Exemplo:

```
1   def intervalo(fim, inicio=1, incremento=1):
2       retorno = []
3       i = inicio
4       while i <= fim:
5           retorno.append(i)
6           i += incremento
```

```
7       return retorno
8
9   print intervalo(10)         # Imprime [1, 2, 3, 4, 5, 6, 7, 8, 9,
10]
10  print intervalo(10, 1, 2)     # Imprime [1, 3, 5, 7, 9]
```

No exemplo acima definimos a função `intervalo`, especificando o valor 1 para os parâmetros `inicio` e `incremento`. Note que o parâmetro `fim` é o primeiro da lista, isto é porque não podemos definir um parâmetro não-opcional depois de um opcional, assim, estes devem ser os últimos da lista.

O Python possibilita também passar argumentos para as funções por palavra-chave (*keyword*). Exemplos utilizando a função `intervalo` definida anteriormente:

```
1   print intervalo(inicio=1, fim=10, incremento=2) # Imprime [1, 3, 5,
7, 9]
2   print intervalo(10, incremento=3)           # Imprime [1, 4, 7, 10]
3   print intervalo(5, inicio=2)                # Imprime [2, 3, 4, 5]
4   print intervalo(inicio=1, fim=5)            # Imprime [1, 2, 3, 4, 5]
```

Repare no exemplo acima que não modificamos a função `intervalo`, a "mágica" toda acontece ao chamar a função, onde especificamos o identificador do parâmetro seguido de um valor. A passagem de argumentos por palavra-chave, como podemos ver no exemplo anterior, possibilita passar valores para os argumentos fora da ordem que os parâmetros foram definidos.

Argumentos de tamanho variável

Podemos definir uma função que receba um número desconhecido de argumentos. Para isso, podemos definir um parâmetro prefixado por um asterisco (*) na lista de parâmetros. Exemplo:

```
1   def soma(*nums):
2       result = 0
3       for n in nums:
4           result += n
5       return result
6
7   print soma(1, 2, 3)         # Imprime 6
8   print soma(1, 2)            # Imprime 3
```

Na função acima, iteramos sobre o parâmetro `nums` acumulando o resultado e o retornando no final, assim, podemos passar quantos argumentos necessários para a função calcular.

Decoradores

Como vimos no início da apostila, tudo em Python é um objeto e funções não são uma exceção. Um decorador é uma função que recebe uma outra função como argumento, a modifica e a retorna. Decoradores tanto podem ser usados para criar ou alterar características das funções, como adicionar atributos, quanto para "envolver" as funções, acrescentando uma camada em torno delas com novas funcionalidades. Exemplo do livro Python para Desenvolvedores, 2ª edição, de Luiz Eduardo Borges:

```
1  # Função decoradora
2  def dumpargs(f):
3      # Função que envolverá a outra
4      def func(*args):
5          # Mostra os argumentos passados para a função
6          print args
7          # Retorna o resultado da função original
8          return f(*args)
9
10     # Retorna a função modificada
11     return func
12
13 @dumpargs
14 def multiply(*nums):
15     m = 1
16     for n in nums:
17         m = m * n
18     return m
19
20 print multiply(1, 2, 3)
```

O exemplo acima imprime:

```
(1, 2, 3)
6
```

A função `dumpargs` envolve a função passada por argumento adicionando uma funcionalidade – retornar uma tupla contendo os argumentos passados para a função – por isso quando chamamos a função `multiply` ele imprimiu tanto a tupla `(1, 2, 3)` quanto o resultado da soma dos argumentos passados.

Note que para aplicarmos uma função decoradora, utilizamos `@dumpargs`. Essa sintaxe está disponível a partir da versão 2.4 do Python. Até então, tínhamos que atribuir o retorno do operador na função a ser modificada. Exemplo utilizando as funções anteriores:

```
1  multiply = dumpargs(multiply)
2  print multiply(1, 2, 3)
```

Veremos em Programação Orientada a Objetos, que Python disponibiliza alguns decoradores, como `@staticmethod`, utilizado para definir um método estático.

O conceito de uma função que recebe uma função e a retorna é conhecido como objetos de primeira-classe e o veremos mais detalhadamente no final da apostila, em Programação Funcional.

DocStrings

DocStrings, como o próprio nome supõe, são strings de documentação, utilizadas para documentar funções, módulos, classes e seus métodos, que veremos mais adiante nessa apostila. São utilizadas para gerar documentação automática com a ferramenta `pydoc` e muitos editores, como os citados na introdução da apostila, as utilizam para fornecer informações, como a utilização de funções, seus parâmetros, utilização de módulos, e etc.

Para definir uma string de documentação, basta fornecer como primeira declaração de um objeto (função, módulo, classe, método, etc.) uma string, normalmente definida entre aspas triplas. Exemplo:

```
1  def intervalo(fim, inicio=1, incremento=1):
2      """Retorna uma faixa de valores inteiros.
3
4      fim -- final da faixa de valores - obrigatório
5      inicio -- inicio da faixa de valores (default 1)
6      incremento -- incremento entre inicio e fim (default 1)
7
8      """
9      retorno = []
10     i = inicio
11     while i <= fim:
12        retorno.append(i)
13        i += incremento
14     return retorno
```

Na primeira linha da DocString demos uma descrição rápida da função – o que ela faz e retorna - e nas linhas seguintes descrevemos a utilização dos parâmetros e seus valores *default*. Não há regras quanto a definição das strings de documentação, porém a PEP 257[7] documenta a semântica e as convenções associadas a estas, como por exemplo, utilizar aspas triplas, a forma de descrever as funções, convenções sobre espaços em branco, etc. Mais informações em http://www.python.org/dev/peps/pep-0257/.

[7] PEP, sigla de Python *Enhancement Proposal*, ou, Proposta de melhoria do Python, consiste de documentos que fornecem informações para a comunidade do Python, como consensos e boas-práticas, descrevem novos recursos da linguagem, seus processos ou ambiente. Cada PEP possui um número de identificação, iniciado pelo 0001, PEP que descreve detalhadamente o que são as elas, quais são seus tipos e etc. Você pode encontrar mais informações em http://python.org/dev/peps/pep-0001.

Módulos

Módulos nada mais são do que arquivos-fonte que podem ser importados para um programa e são utilizados quando desejamos utilizar atributos (funções, objetos, classes, etc.) já definidos, em nossos programas. Um módulo é executado quando é importado, e é compilado e armazenado com extensão `.pyc` ou `.pyo` quando importado pela primeira vez.

Para um módulo ser importado ele precisa possuir uma das extensões de arquivos-fonte Python, ou seja, `.py`, `.pyc` e `.pyo` e precisa estar no PYTHONPATH - a lista de diretórios e arquivos onde o interpretador procura os módulos quando são importados. O PYTHONPATH normalmente é composto do diretório corrente do arquivo, o diretório de instalação do Python e alguns de seus diretórios internos – como o Lib, lib e DLLs (no Windows) – entre outros. Para verificar qual o PYTHONPATH de um módulo, a biblioteca-padrão do Python disponibiliza o objeto `path`, localizado no módulo `sys`.

A biblioteca padrão do Python inclui um número muito grande de módulos já prontos e compilados para que não precisemos "reinventar a roda". Dentre esses módulos, encontra-se, por exemplo, o `calendar`, para manipulação de calendários, o `datetime`, para manipulação de datas, o `time`, para manipulação de tempo, o `sqlite3` para trabalhar com bancos de dados SQLite (que veremos mais a diante, em Trabalhando com Bancos de Dados) e muitos outros. Devido a essa grande quantidade de módulos disponíveis na biblioteca-padrão, diz-se que Python vem com baterias inclusas, ou *"Batteries included"*.

Além desses módulos da biblioteca-padrão e de bibliotecas de terceiros, é normal que com o crescimento de um programa, exista o desejo de separá-lo em vários arquivos para uma melhor organização, por exemplo, um arquivo para classes, outro para manipulação de dados, outro para validação, outro com funções de utilidades, etc. Um módulo pode ser importado dentro de outro módulo ou dentro do próprio interpretador interativo do Python para se ter acesso ao que foi definido dentro do arquivo.

Criando um módulo

Iremos agora criar um módulo contendo um conjunto de funções que manipulam dois números (soma, diferença, produto, quociente e potência) e iremos chamá-lo de `numeros.py`.

```
1  # numeros.py
2  u"""Faz a manipulacao de dois numeros, através das funções
3  soma, diferenca, produto, quociente e potencia"""
4
```

```
5  def soma(x, y):
6     u"""Esta função retorna a soma dos dois números passados como
7    parâmetros"""
8        return x + y
9
10 def diferenca(x, y):
11    u"""Esta função retorna a diferença dos dois números passados
como parâmetros"""
12    return x - y
13
14 def produto(x, y):
15    u"""Esta função retorna o produto dos dois números passados"""
16    return x * y
17
18 def quociente(x, y):
19    u"""Retorna o quociente dos dois números."""
20    return x / y
21
22 def potencia(x, y):
23    u"""Retorna a potencia dos dois números"""
24    potencia = 1
25       while y > 0:
26          potencia = potencia * x
27          y = y - 1
28    return potencia
```

Importando módulos

Com o módulo `numeros.py` em mãos, crie outro módulo e salve-o como `teste_numeros.py` no mesmo diretório que o `numeros.py`. Para que possamos utilizar os atributos contidos em `numeros.py`, primeiramente este precisa ser importado para o seu *namespace* com o comando `import`. Insira o seguinte código para importar o módulo `numeros.py`:

```
import numeros
```

Note que não há a necessidade de escrever a extensão `.py` no nome do módulo quando o importamos. Agora que o módulo foi importado para o nosso programa, utilizamos a seguinte sintaxe para utilizar suas funções:

```
nome_do_modulo.nome_da_funcao()
```

Para utilizar uma função do módulo `numeros`, a função `soma`, por exemplo, devemos fazer:

```
print numeros.soma(2, 3)
```

Se por acaso utilizarmos o código abaixo, chamando uma função que não existe, uma exceção é acionada:

```
numeros.outra()
```

Para sabermos quais funções estão definidas em um determinado módulo, podemos utilizar função embutida `dir`, que retorna uma lista de strings com o nome de todos os atributos definidos em um módulo. Exemplo:

```
1  import numeros
2  print dir(numeros)
```

Iremos visualizar o seguinte:

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'diferenca', 'potencia', 'produto', 'quociente', 'soma']
```

Os atributos `'__builtins__'`, `'__doc__'`, `'__file__'`, `'__name__'` e `'__package__'` são atributos embutidos de módulos Python e serão explicados mais adiante neste capítulo, exceto por `__package__`, que será explicado na parte de Pacotes. Repare que seus nomes são precedidos e sucedidos por dois __ (*underscores*) – essa é a sintaxe do Python para identificar seus atributos e métodos especiais. Módulos são objetos *singleton*, ou seja, é carregada somente uma instância deles na memória, e esta fica disponível de forma global para o programa. Para re-importar um módulo, basta utilizar a função `reload`, desta forma:

```
1  import numeros
2  reload(numeros)
```

Note que quando chamarmos a função `soma` do módulo `numeros` nós a referenciamos pelo nome do módulo - esse tipo de importação é chamado de importação absoluta. Há ainda outras formas para importar um módulo, como por exemplo:

```
from numeros import soma, produto
```

Assim, ao invés de chamar a função `soma` pelo nome do módulo, podemos simplesmente fazer:

```
1  print soma(2, 4)
2  print produto(3, 9)
```

No exemplo anterior importamos somente as funções `soma` e `produto` do módulo `numeros`, incluindo-as no nosso *namespace*, assim, não precisamos referenciar o módulo quando chamarmos suas funções.

Podemos ainda importar todos os atributos de um módulo com o curinga asterisco (*). Exemplo:

```
from numeros import *
```

Esta importação é chamada de importação relativa e não é considerada uma boa prática de programação, já que o módulo importado pode conter atributos com nomes que já estão sendo utilizados pelo módulo que a importou, alterando o comportamento do programa e até gerar erros.

A importação absoluta "separa" os *namespaces* dos módulos importados, e por isso, é considerada uma prática de programação melhor que a importação relativa. A PEP 328 trata especificamente de boas-práticas em relação à importação de módulos, para mais informações, consulte a página dessa PEP em http://python.org/dev/peps/pep-0328/.

Quando um módulo possui um nome comprido e não queremos repetí-lo toda vez que for necessário, podemos usar a seguinte sintaxe:

```
import numeros as n
```

Ou ainda:

```
1  import numeros
2  n = numeros
```

Dessa forma, o módulo `numeros` é associado à variável `n`. Assim, é necessário somente referenciar o módulo como `n` ao invés de `numeros`.

Como vimos anteriormente, módulos possuem atributos embutidos, como `__name__` e `__doc__`. O atributo `__name__` armazena o nome do módulo e é freqüentemente utilizado para verificar se o módulo corrente está sendo importado ou executado isoladamente. Exemplo:

```
1  import sys
2
3  """DocString do módulo"""
4
5  if __name__ == "__main__":
6     nome = sys.argv[0]      # O mesmo que __file__
7     args = sys.argv[1:]     # Lista de argumentos passados
8
9     if len(args) > 0:
10       print "Argumentos de", nome
11       for arg in args:
12          print arg
13    else:
14       print "Nenhum argumento passado"
15 else:
16    print u"Módulo importado, nada a fazer"
```

No exemplo acima comparamos o valor de `__name__` a "__main__", isto porque quando o módulo é executado por si só o valor de `__name__` é __main__, e quando é importado, o `__name__` é nome do módulo. Por exemplo, se salvarmos o módulo anterior

como `modulo1.py` e o importamos de outro módulo ou do interpretador interativo podemos imprimir o valor de `__name__` da seguinte forma:

```
1  import modulo1
2  print modulo1.__name__      # Imprime modulo1
3  print modulo1.__doc__     # Imprime DocString do módulo
```

Note que no módulo anterior importamos o módulo `sys`, presente na biblioteca-padrão do Python. Este módulo contém, entre outros atributos, o objeto `argv` – uma lista que contém todos os argumentos passados por linha de comando para o programa. O primeiro valor de `argv` é o nome do arquivo, por isso, atribuímos o valor da variável `nome` ao primeiro elemento da lista, e o valor da variável `args` aos elementos da segunda posição em diante. O atributo embutido `__file__` armazena o nome do arquivo e seu caminho, assim como `argv[0]`. Caso o módulo tenha sido importado, a mensagem `"Módulo importado, nada a fazer"` é impressa e nada mais. O atributo `__doc__` armazena a DocString do módulo e imprimimos seu texto na linha 3 do exemplo anterior.

Espaço de nomes

A partir do momento que são criadas, as variáveis dependem de um local para serem armazenadas. Os *namespaces* são onde ficam registradas as variáveis, e funcionam como dicionários que são atualizados dinamicamente em tempo de execução, onde as chaves (*keys*) são os nomes das variáveis e os valores (*values*) são os valores contidos nelas. Existem três *namespaces* no Python: local, global e embutido.

Local

Cada função (ou método) possui seu próprio *namespace* – chamado de local, que mantém no seu dicionário as variáveis da função, inclusive seus parâmetros. Variáveis definidas em funções só podem ser acessadas de dentro delas. Exemplo:

```
1  def funcao_exemplo(x, y):
2      soma = x + y
3      print locals()
4
5  funcao_exemplo(1, 2)     # Imprime {'soma': 3, 'y': 2, 'x': 1}
```

Repare que a função `locals` retorna na forma de um dicionário o *namespace* local da função em que está inserido. No *namespace* da função acima estão inseridas a variável `soma`, definida no corpo da função, os parâmetros `x` e `y` e seus respectivos valores.

Global

Cada módulo possui seu próprio espaço de nomes, chamado de global, onde encontra-se todas as variáveis, funções e classes definidas no módulo, além dos módulos importados no módulo corrente. Exemplo:

```
1  # modulo_exemplo.py
2
3  import numeros
4
5  var_local = 42
6
7  def funcao_exemplo():
8      pass
9
10 class classe_exemplo:
11     pass
12
13 print globals(
```

O código anterior gera a seguinte saída:

```
{'funcao_exemplo': <function funcao_exemplo at 0x02CA47F0>,
'__builtins__': <module '__builtin__' (built-in)>, 'numeros': <module
```

```
'numeros' from 'C:/Modulos/numeros.pyc'>, 'classe_exemplo': <class
__main__.classe_exemplo at 0x02C9B900>, '__package__': None,
'var_local': 42, '__name__': '__main__', '__doc__': None}
```

A função `globals` funciona igual `locals`, porém retorna um dicionário contendo todos os atributos globais do módulo. Acima podemos reparar que, além da função, classe e variável definidas no módulo, a função `globals` retornou o módulo importado `numeros.py` e os atributos embutidos `__file__`, `__name__` e `__doc__`, explicados anteriormente.

As variáveis globais podem ser "ofuscadas" por variáveis locais, pois o *namespace* local é consultado antes do global ao referenciarmos um objeto de dentro de uma função. Para evitar isso é preciso declarar a variável como global no escopo local. Exemplo:

```
1  total = 0;
2
3  def soma(x, y):
4     total = x + y            # Variável total definida na função
5     print "Valor de total dentro da função:", total
6
7  soma(10, 20)                                    # Imprime 30
8  print "Valor de total fora da função:", total    # Imprime 0
```

No exemplo acima, definimos uma variável `total` no escopo global iniciada com 0 (zero). Dentro da função `soma` definimos outra variável `total`, para armazenar o resultado da soma dos dois parâmetros. Na linha 5, `total` vale 30 – resultado da soma de 10 e 20 passados por argumento. Na linha 8, imprimimos o valor de `total` do escopo global, cujo valor não foi alterado desde sua inicialização. Para que a variável `total` do escopo global seja utilizada pela função `soma`, devemos definí-la como global. Exemplo:

```
1  total = 0
2
3  def soma(x, y):
4     global total
5     total = x + y    # Variável total definida na função
6     print "Valor de total dentro da função:", total
7
8  soma(10, 20)
9  print "Valor de total fora da função:", total
```

Que imprime:

```
Valor de total dentro da função: 30
Valor de total fora da função: 30
```

Embutido ou *Built-in*

É onde todas as funções, constantes e exceções embutidas do Python são definidas. É acessível de qualquer módulo, é criado quando o interpretador Python é aberto, e dura até ele

ser fechado. Os atributos desse *namespace* ficam em um módulo chamado `__builtins__`, e alguns exemplos de atributos definidos são:

- As funções `print`, `raw_input`, etc.;
- Funções de coerção – `int`, `float`, `str`, etc.;
- Tipos e seus métodos: `int`, `float`, `complex`, `str`, `list`, `tuple`, `dict`, etc;

Pacotes

Pacotes, ou *packages*, são pastas que contém um arquivo-fonte, normalmente vazio, chamado `__init__.py`. Eles são utilizados como coleções para organizar módulos de forma hierárquica. Exemplo:

```
+ bancos_dados                -> Pasta (Pacote)
| +-__init__.py               -> Identifica pasta como pacote
| +-mysql.py                  -> Módulo para MySQL
|   +-conecta()               -> Função do módulo mysql
|   +-desconecta()            -> Função do módulo mysql
|   +-executa()               -> Função do módulo mysql
| +-sqlite.py                 -> Módulo para SQLite
|   +-conecta()               -> Função do módulo sqlite
|   +-desconecta()            -> Função do módulo sqlite
|   +-executa()               -> Função do módulo sqlite
|
+ aplicacao.py
```

Acima apresentamos uma estrutura hierárquica onde `banco_dados` é uma pasta que contém o arquivo `__init__.py`, que identifica a pasta como um pacote. Nela estão contidos os módulos `mysql.py` e `sqlite.py`, ambos possuindo os métodos `conecta`, `desconecta` e `executa`. O módulo `aplicacao.py` está no mesmo nível hierárquico do pacote `banco_dados`, então para acessar dele os módulos contidos no pacote `banco_dados`, temos que referenciá-lo. Exemplo:

```
1  from base_dados import sqlite
2
3  sqlite.conecta()
```

No exemplo acima importamos o módulo `sqlite` do pacote `base_dados`. Note que apesar de utilizarmos o `from` para a importação, ainda precisamos identificar o módulo quando chamamos a função `conecta`, ou seja, é uma importação relativa. Para importar todos os módulos de um pacote, basta utilizar o curinga asterisco, exemplo:

```
1  from base_dados import *
2
3  sqlite.conecta()
4  mysql.conecta()
```

Para fazer uma importação absoluta de um módulo contido em um pacote, a sintaxe é similar a da mostrada anteriormente. Exemplo:

```
1  from base_dados.sqlite import *
2
3  conecta()
4  executa()
5  desconecta()
```

Caso se lembre, quando vimos os atributos embutidos de módulos ficamos de explicar o atributo `__package__`. Este atributo simplesmente armazena a qual pacote o módulo corrente pertence.

É comum que o arquivo `__init__.py` esteja vazio, que exista somente para tornar a pasta em que está contido em um pacote, mas também podemos inserir código-fonte nele como em qualquer outro arquivo. Dentro dele podemos definir código de inicialização do pacote – que será executado sempre que um ou mais módulos do pacote forem importados – e controlar quais os módulos do pacote serão importados quando realizamos a importação relativa utilizando o curinga asterisco – `from pacote import *` – definindo o nome desses módulos em uma variável chamada `__all__` na forma de tupla ou de lista.

Tomando como base a estrutura de pacotes definida no início desse capítulo, podemos definir o seguinte código no arquivo `__init__.py` do pacote `banco_dados`:

```
1  -*- coding: latin-1 -*-
2  __all__ = ('mysql',)
3  print u'Módulo' __name__ 'importado'
```

Na linha 2 definimos a variável `__all__` e atribuímos a ela uma tupla contendo o nome do módulo `mysql`, assim, quando executarmos a instrução `from banco_dados import *`, somente o módulo `mysql` será importado para nosso *namespace*.

A instrução `print` da linha 3 será executada todas as vezes que houver uma importação envolvendo o pacote `banco_dados`, seja ele por importação absoluta ou relativa, importando um módulo contido nele ou o pacote como um todo.

Trabalhando com arquivos

A biblioteca-padrão do Python, como sabemos, disponibiliza uma infinidade de módulos já prontos. Arquivos no Python são representados por objetos do tipo `file`, que oferecem métodos para diversas operações de arquivos.

Criando arquivos

Para criar um arquivo, podemos chamar tanto a função embutida `open`, quanto o construtor do tipo `file` e passar o caminho do arquivo e o modo de abertura como argumento. O modo de abertura é uma string que indica como o arquivo será aberto – para escrita, somente-leitura e etc. Exemplo:

```
1  arquivo = file(r"C:/arquivo.txt", 'w')
2  arquivo.write("Linha 1\n")
3  arquivo.write("Linha 2\n")
4  arquivo.close()
```

No exemplo acima criamos um arquivo e escrevemos duas linhas nele. Se executarmos essa instrução duas vezes, o arquivo antigo será sobrescrito pelo novo, pois utilizamos o modo de escrita “w”. Mais sobre modos de abertura a seguir.

Atenção

Repare que passamos o caminho completo do arquivo para a função `file`, mas preste atenção em um detalhe importante: não utilizamos barras-invertidas (\ ) para descrever caminhos no Python, mesmo no Windows, onde isso é padrão. Ao invés, prefira utilizar barras (/), assim como nos sistemas **nix like*, como Linux, BSD, Mac OS, e muitos outros. Uma das características mais interessantes e exploradas do Python é ser multi-plataforma; definir caminhos com barra-invertida irá funcionar no Windows, porém não no Linux e outros sistemas, perdendo assim a portabilidade que a linguagem e plataforma lhe oferecem.

Para não perder em portabilidade, além de não utilizar as barras-invertidas em caminhos de arquivos, evite sempre que possível executar comandos dependentes de plataforma de dentro de programas/*scripts*, como `dir`, `md`, `mv` e etc, dando preferência às funções e aos métodos disponibilizados pela biblioteca-padrão do Python.

Modos de abertura

Os principais modos de abertura de arquivos no Python são:

- `"r"` Abre arquivo somente para leitura. O ponteiro do arquivo é localizado no início do arquivo.
- `"r+"` Abre arquivo tanto para leitura quando para escrita. O ponteiro do arquivo é localizado no início do arquivo.
- `"w"` Abre arquivo somente para escrita. Se o arquivo não existe, o cria, se existe, o sobrescreve.
- `"w+"` Abre arquivo para leitura e escrita. Assim como os dois acima, se o arquivo não existe, o cria, se existe, o sobrescreve.
- `"a"` Abre arquivo somente para acrescentar (*append*). Ponteiro é localizado no fim do arquivo se arquivo já existe. Se arquivo não existe, o cria.
- `"a+"` Abre arquivo tanto para ler quanto para acrescentar. O ponteiro é localizado no fim do arquivo caso ele já exista, caso contrário, o cria.

Caso não mencionarmos o modo de abertura do arquivo, "r" é utilizado por padrão. Arquivos podem conter tanto texto quanto dados binários, e para isto há adicionalmente modos de abertura no formato binário, onde geralmente há um sufixo "b" no nome, como em "rb", "rb+", "wb", "wb+", "ab" e "ab+".

Lendo arquivos

Podemos ler o conteúdo de um arquivo de diversas maneiras, e a primeira que vamos ver é com o método `read`. Esse método aceita como argumento o número de *bytes* a serem lidos, e quando esse argumento não é fornecido o método lê o conteúdo todo do arquivo. Exemplo:

```
1  arquivo = open(r"C:/arquivo.txt", 'r')
2  print arquivo.read()
3  arquivo.close()
```

No exemplo acima lemos e imprimimos todo o conteúdo do arquivo criado no primeiro exemplo. Abaixo fornecemos um número de *bytes* a serem lidos, exemplo:

```
1  arquivo = file(r"C:/arquivo.txt", 'r')
2  print arquivo.read(5)         # Imprime Linha
3  arquivo.close()
```

Outra forma de ler arquivos é pelo método `readline`, que lê uma linha inteira do arquivo. Exemplo:

```
1  >>> arquivo = file(r"C:/arquivo.txt", 'r')
2  >>> arquivo.readline()         # Imprime Linha 1
3  >>> arquivo.readline()         # Imprime Linha 2
```

```
4  >>> arquivo.close()
```

Objetos do tipo `file` suportam iteradores, que serão vistos em Programação Funcional, o que permite iterar sobre o objeto `file` em um *loop* `for` sem precisar chamar nenhuma função explicitamente. Essa solução, além de gerar um código mais simples, é mais rápida e eficiente em termos de memória. Exemplo:

```
1  arquivo = open(r"C:/arquivo.txt", 'r')
2  for linha in arquivo:
3     print linha,
4  arquivo.close()
```

Repare que inserimos uma vírgula no final da instrução `print linha`, na linha 3, o que faz com que a função `print` não insira um "`\n`" após ser executada. Essa alternativa é uma abordagem mais simples, mas que não fornece nenhum tipo de controle mais refinado.

O método `readlines` é ainda outro meio de ler o conteúdo de um arquivo em Python, ele retorna uma lista de strings contendo as linhas do arquivo. Exemplo:

```
1  arquivo = file(r"C:/arquivo.txt", 'r')
2  print arquivo.readlines()          # Imprime Linha
3  arquivo.close()
```

Posicionando-se em arquivos

Para obter a posição corrente em um arquivo utilizamos o método `tell`, que retorna um inteiro medido em *bytes* a partir do início do objeto `file`. Exemplo:

```
1  arq = file(r'C:/arquivo.txt')
2  conteudo_1 = arq.read(5)
3  print arq.tell()                  # Imprime 5
```

Para alterar a posição atual em um arquivo utilizamos o método `seek`, que recebe dois argumentos. O primeiro é o *offset* – o número de *bytes* em queremos nos mover – e o segundo é a referência de onde queremos nos localizar. Utilizando como referência o valor 0 (*default*), nos posicionamos em relação ao início do arquivo; o valor 1 faz com que nos posicionemos em relação à posição atual e 2 utiliza como referência o final do arquivo. Exemplo:

```
1  arq = open(r'C:/arquivo_2.txt','w')
2  arq.write('123456789abcd')
3  arq.close()
4
5  arq = open(r'C:/arquivo 2.txt')
6  arq.seek(5)
7  print arq.read(1)          # Imprime 6
8  arq.close()
```

No exemplo acima posicionamos o cursor no sexto *byte* do arquivo em relação ao seu início. Exemplo utilizando o final do arquivo como referência:

```
1  arq = file(r'C:/arquivo_2.txt')
2  arq.seek(-4, 2)
3  print arq.read(4)         # Imprime abcd
4  arq.close()
```

Como pode perceber nesse exemplo, ao passarmos um *offset* negativo, recuamos o ponteiro do arquivo, nesse caso, em -4 *bytes*.

Atributos e métodos de arquivos

Assim como qualquer objeto em Python, arquivos possuem atributos e métodos relacionados. A seguir veremos alguns dos atributos que objetos do tipo `file` possuem:

- `closed`: Retorna True se arquivo está fechado e False, caso não;
- `mode`: Retorna o modo de abertura do arquivo;
- `name`: Retorna o nome do arquivo;

Exemplos:

```
1  arquivo = file(r'C:/arquivo.txt', 'w')
2  print """\
3  Nome do arquivo: %s
4  Modo de abertura: %s
5  Está fechado?: %s""" % (arquivo.name, arquivo.mode, arquivo.closed)
```

Que imprime:

```
Nome do arquivo: C:/arquivo.txt
Modo de abertura: w
Está fechado?: False
```

Atenção

Repare que em todos os exemplos anteriores chamamos o método `close` nos arquivos que abrimos, tanto para leitura quanto para gravação. O método `close`, como deve presumir, fecha o arquivo e libera seus recursos da memória.

Como veremos no exemplo a seguir, quando abrimos um arquivo, escrevemos *bytes* nele e não executamos o método `close`, as alterações não são salvas no arquivo em disco e só ficam registradas no *buffer* do arquivo, em memória. Para descarregar o *buffer* manualmente há o método `flush`. Para demonstrar a utilização do método `flush`, abra o interpretador interativo do Python e entre com os seguintes comandos:

```
1  >>> arq = open(r'C:/arquivo.txt', 'w')
```

```
2  >>> arq.write('Linha 1\n')     # Abra o arquivo com seu editor
preferido e veja que ele existe mas ainda está vazio.
3  >>> arq.flush() # Feche o arquivo e o abra novamente – a string foi
gravada em disco!
4  >>> arq.write('Linha 2\n')
5  >>> arq.close()
```

Como pode perceber, o método `write`, não escreve strings diretamente no arquivo em disco, mas no *buffer*, que é automaticamente descarregado quando o arquivo é fechado. Python, assim como a grande maioria das linguagens e plataformas, utiliza os *buffers* para evitar realizar vários acessos ao disco, o que poderia diminuir drasticamente o desempenho da aplicação.

Tratamento de exceções

O que são exceções?

Uma exceção é um evento que ocorre durante a execução de um programa e interrompe o fluxo normal das suas instruções. Uma divisão por zero, por exemplo, é uma exceção, e caso não seja tratada é propagada através das chamadas de funções até o módulo principal do programa, interrompendo sua execução e exibindo uma mensagem de *traceback*. Exemplo de uma divisão por zero não-tratada:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

No exemplo acima a exceção `ZeroDivisionError` foi gerada, e como não foi tratada, teve seus dados de *traceback* exibidos, como a linha e módulo em que a exceção ocorreu.

Try / Except

As instruções `try` e `except` permitem o tratamento de exceções no Python. Se uma exceção ocorre dentro de um bloco `try`, ela deve ser manipulada através de uma instrução `except` adequada. Exemplo:

```
1  try:
2     print 1 / 0
3  except ZeroDivisionError:
4     print "Erro ao dividir por zero"
```

No exemplo anterior, quando a exceção `ZeroDivisionError` ocorre, a mensagem `"Erro ao dividir por zero"` é impressa na tela ao invés da mensagem de *traceback* vista anteriormente.

Quando passamos o nome da exceção que desejamos manipular para a instrução `except`, só esta exceção é capturada. Podemos também inserir várias instruções `except` para diferentes exceções. Exemplo:

```
1  try:
2     x = int(raw_input("Digite um número: "))
3     y = int(raw_input("Digite outro: "))
4     resultado = x / y
5     print "Resultado da divisão:", resultado
6  except ValueError:
7     print "O valor digitado não é um número válido"
```

```
8  except ZeroDivisionError:
9     print "O valor digitado para y deve ser maior do que 0 (zero)"
```

No exemplo acima inserimos duas instruções `except`, uma para tratar exceções do tipo `ValueError` e outra para `ZeroDivisionError`. Caso o usuário digite um valor inválido (não numérico) ou não digite nada, a primeira exceção é acionada. Se o usuário digitar 0 (zero) na segunda entrada de dados a segunda exceção é acionada.

A última instrução `except` pode omitir o nome da exceção, assim, ela pode servir de curinga – caso uma exceção seja acionada e nenhuma das instruções `except` anteriores a capture, ela faz o trabalho. Exemplo:

```
1  try:
2     nome_arq = raw_input(r"Caminho do arquivo: ")
3     arquivo = open(nome_arq)
4     print arquivo.read()
6     arquivo.close()
7  except IOError:
8     print "Erro no arquivo '%s'", nome_arq
9  except:
10    print "Um erro inesperado ocorreu!"
```

No exemplo acima, caso o arquivo não seja encontrado, ou qualquer outro problema que ocorra com o arquivo, a mensagem do primeiro `except` é exibida. Caso outra exceção ocorra e não seja tratada pelo primeiro `except`, a mensagem do segundo `except` é exibida na tela. Apesar de não ser recomendado na maioria dos casos, podemos inserir um `except` curinga em um bloco `try` para tratar de todas as exceções.

Acionando exceções

A instrução `raise` nos possibilita forçar uma exceção a ser acionada. Exemplo:

```
1  try:
2     nome = raw_input("Nome: ")
3     if len(nome) == 0:
4        raise ValueError("Digite seu nome!")
5
6     idade = int(raw_input("Idade: "))
7     if idade < 18:
8        raise ValueError("Você precisa ser maior de idade!")
9  except ValueError, mensagem:
10    print mensagem
11 except:
12    print "Um erro inesperado ocorreu!"
```

Caso o nome não tenha sido digitado, ou seja, a quantidade de caracteres retornada pela função `len` seja igual a 0 (zero), a exceção `ValueError` é acionada manualmente, passando uma variável por argumento. O mesmo ocorre quando o usuário digita um número

menor que 18 para a idade. Repare que passamos uma variável chamada `mensagem` na linha 9 para armazenar a mensagem de erro retornada pela exceção `ValueError` e imprimí-la na tela.

Finally

A instrução `finally` é acionada sempre no final de um bloco `try` / `except`, caso uma exceção seja acionada ou não. É comumente utilizada para liberar recursos, como arquivos, conexões de bancos de dados e de rede. Exemplo:

```
1  try:
2     arquivo = open(r"C:/arquivo.txt", "w")
3     try:
4        arquivo.write("Linha 1\n");
5     except:
6        print "Erro ao escrever no arquivo!"
7  except IOError:
8     print "Erro ao abrir o arquivo!"
9     finally:
10       arquivo.close()
```

No exemplo acima temos dois blocos `try` / `except` aninhados. O `except` mais interno é acionado somente quando um erro ao escrever no arquivo ocorre. O `except` mais externo é acionado caso algum erro ocorra ao abrir o arquivo. O bloco `finally` é acionado sempre, queira o arquivo seja aberto com sucesso ou não, garantindo assim que ele seja fechado adequadamente, liberando seus recursos da memória.

Trabalhando com bancos de dados

O Python suporta a grande maioria dos Sistemas Gerenciadores de Bancos de dados (SGBDs) existentes por meio de DBIs (*Database Interfaces*), também chamados de DB-APIs (Database Application Programming Interfaces).

Um DBI é um *driver* específico para um SGBD, portanto, para se conectar a um banco de dados MySQL devemos baixar e instalar o módulo para MySQL.

Nesta apostila utilizaremos o SQLite, uma biblioteca escrita na linguagem C que fornece um banco de dados leve baseado em disco. Há dois principais motivos para essa escolha, o primeiro é que o SQLite não precisa de um servidor, portanto não teremos que instalar um servidor, como o MySQL, iniciar seu serviço, configurar uma conta de usuário e etc. O segundo é que a biblioteca-padrão do Python inclui os módulos para se conectar a bancos de dados SQLite desde a versão 2.5, portanto, nada de baixar o DBI do SQLite para Python.

O SQLite é um banco de dados projetado para ser simples, portanto, alguns recursos como *stored procedures* e funções *builtin* presentes em SGBDs como o MySQL, PostgreSQL e Oracle foram deixados de fora. Apesar de termos escolhido o SQLite para trabalhar com banco de dados nesta apostila, tenha em mente que o SQLite não foi projetado para aplicações cliente/servidor, nem para armazenar grandes quantidades de dados ou para ser acessado por um grande número de usuários simultaneamente. Segundo os desenvolvedores do SQLite, suas aplicações mais apropriadas são aplicativos *desktop* que precisem armazenar informações localmente, *websites* com baixo/médio tráfego, dispositivos como celulares, PDAs, *set-top boxes*, e para fins didáticos – como é o nosso caso.

A partir desse ponto da apostila iniciaremos a criação de um aplicativo bem simples de agenda de contatos, chamado de pyAgenda (eu sei, não sou bom com nomes). Esse aplicativo será desenvolvido com uma arquitetura em camadas que separa o acesso a dados (DAO – *Data Access Object*), os modelos ou entidades – no nosso caso, a classe `Contato` – e a interação com o usuário.

Neste capítulo criaremos a base de dados do aplicativo e a tabela que armazenará seus contatos, e aprenderemos como executar *queries* SQL com Python. No próximo capítulo partiremos para a Orientação a Objetos, criaremos a classe de entidade, a interface gráfica do aplicativo e utilizaremos os conceitos aplicados nesse capítulo para criar a classe de acesso a dados.

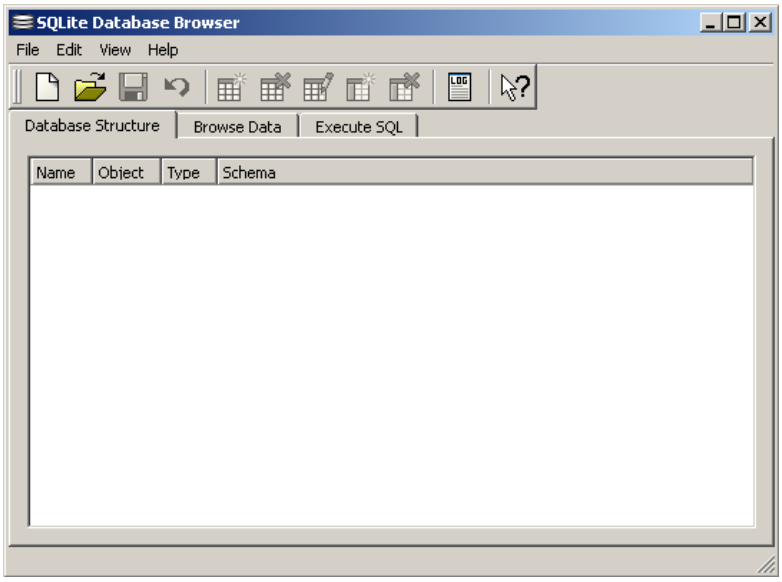
Como é de se esperar, é necessário que você possua algum conhecimento de SQL (*Structured Query Language*) para poder compreender completamente o conteúdo esse

capítulo, e como essa apostila não é de SQL, mas Python, não cobriremos nem a sintaxe da linguagem, nem os conceitos de tabelas, registros, tipos de dados e demais assuntos pertinentes a bancos de dados. Recomendamos ao estudante o excelente tutorial de SQL do W3Schools, disponível em http://www.w3schools.com/sql (em inglês).

Criando o banco de dados

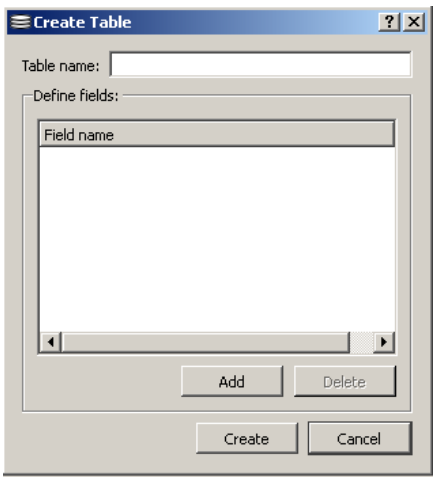
Há várias maneiras de criar bancos de dados SQLite, uma delas é utilizar um aplicativo gráfico, como o SQLite Browser, disponível para ambientes Windows, Linux, BSD e Mac. Outra forma, também independente de plataforma, é o *plugin* para o Mozilla Firefox, SQLite Manager. Ainda outra maneira, mais flexível e que requer um pouco mais de conhecimento de SQL, além de comandos de gerenciamento específicos do SQLite, é baixar um binário pré-compilado (executável) do SQLite no endereço http://www.sqlite.org/download.html e executar comandos SQL para criar as tabelas.

Para criar nosso banco de dados utilizando o *SQLite Browser*, baixe sua última versão no endereço citado acima. Após o término do *download*, descompacte o arquivo e entre na pasta em que foi extraído. No Windows não é necessário instalá-lo, bastando somente executar o arquivo *SQLite Database Browser 2.0 b1.exe* [8]. A edição para Linux possui dependências, como a biblioteca QT, e precisa ser compilada a partir do seu código fonte, como demonstra as instruções em http://sqlitebrowser.sourceforge.net/development.html. A *interface* do programa se parece com isso:

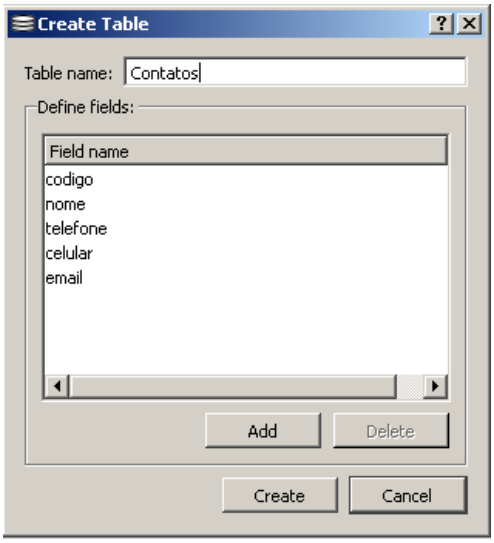


[8] O nome do executável varia de acordo com a versão do aplicativo.

Para criar um novo banco de dados, vá em *File*, *New Database*. Escolha um local que deseja armazenar seus projetos desenvolvidos em Python e crie uma pasta (nada mal se ela se chamasse algo como *pyAgenda*, não?) e nomeie nosso banco de dados como `agenda.sqlite`. O formulário para criação de tabela será exibido logo após o banco de dados ser criado:



No campo *Table name*, digite *Contatos*. Agora vamos adicionar os campos na tabela *Contatos* – clique em *Add*, digite codigo no campo *Field name* e selecione INTEGER PRIMARY KEY no campo *Field type*. Clique novamente no botão *Add*, digite nome no campo *Field name* e selecione TEXT no campo seguinte. Repita esse procedimento até que o formulário se pareça com isso:



Com todos os campos adicionados, clique em *Create*. Pronto, agora só precisamos salvar a base de dados. Para isso vá em *File*, *Save database*.

Para criar nosso banco de dados utilizando o binário pré-compilado do SQLite precisamos nos certificar de que esteja presente no nosso sistema. Algumas distribuições Linux já incluem o SQLite por padrão nas suas instalações, como é o caso do Fedora. No Windows, como já foi dito, é necessário baixar uma versão apropriada para o sistema no *link* previamente citado, descompactar o arquivo, navegar através do *prompt* de comandos até o diretório onde executável se localiza, e chamar o aplicativo passando o nome do banco de dados a ser criado. Supondo que o binário do SQLite esteja em `D:\Downloads\sqlite-shell-win32-x86-3070900\`, precisamos executar os seguintes comandos no *prompt*:

```
1  C:\>
2  C:\> D:
3  D:\> cd Downloads\sqlite-shell-win32-x86-3070900
4  D:\Downloads\sqlite-shell-win32-x86-3070900> sqlite3 agenda.sqlite
```

No Linux, presumindo que o SQLite esteja instalado e presente no PATH do sistema, podemos abrimos um terminal e disparar a seguinte instrução:

```
$ sqlite3 agenda.sqlite
```

Tanto no Windows quanto no Linux, o *prompt* muda quando o interpretador do SQLite está sendo executado, algo como

```
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Agora só nos falta inserir o comando SQL para criar a tabela `contatos`:

```
CREATE TABLE Contatos (codigo INTEGER NOT NULL PRIMARY KEY
AUTOINCREMENT, nome VARCHAR(255) NOT NULL, telefone VARCHAR(14) NOT
NULL, celular VARCHAR(14) NOT NULL, email VARCHAR(100) NOT NULL);
```

Para sair do interpretador do SQLite basta pressionar CTRL + C / CTRL + Break no Windows, ou CTRL + C no Linux.

Atenção

Caso você tenha criado o banco de dados fora da pasta destinada para nosso aplicativo, mova-o para dentro dela antes de passar para o próximo tópico.

Estabelecendo conexão

Antes de qualquer coisa, crie um módulo dentro da pasta que você criou para armazenar nosso aplicativo. Utilizaremos esse módulo para testar os comandos para nos conectar ao banco de dados e executar nossas *queries*.

O módulo para trabalhar com banco de dados SQLite no Python é o `sqlite3`, portanto, precisamos importá-lo para nosso módulo, dessa forma:

```
import sqlite3
```

Para executarmos uma *query* no nosso banco de dados primeiro precisamos nos conectar a ele através da função `connect`, que recebe como um dos seus argumentos o caminho do banco de dados:

```
conexao = sqlite3.connect('agenda.sqlite')
```

Repare que não precisamos nesse caso passar o caminho todo do banco de dados, já que o módulo que você criou se encontra no mesmo diretório que o arquivo do banco de dados. Caso precisemos nos conectar a um banco de dados em um local diferente do nosso arquivo, podemos passar o caminho absoluto do banco, mas não se esqueça de usar barras (/) ao invés de barras-invertidas (\ ) para descrever o caminho.

Executando *queries*

A função `connect` executada anteriormente cria um objeto do tipo `Connection`, que por sua vez possui um método chamado `cursor`, que cria um objeto do tipo `Cursor` que utilizaremos para executar nossas *queries*. Exemplo:

```
cursor = conexao.cursor()
```

Agora que temos nossos objetos `Connection` e `Cursor`, temos que definir o comando SQL que iremos executar:

```
comando = 'INSERT INTO Contatos (nome, telefone, celular, email)
VALUES (?, ?, ?, ?)'
```

A string acima requer alguma explicação na parte das interrogações, não? Se você seguiu nosso conselho e aprendeu SQL antes de chegar nessa parte, você deve saber que onde estão os quatro pontos de interrogação vão os valores que serão inseridos na tabela *Contatos* pela cláusula INSERT. Os pontos de interrogação servem como *placeholders* para os valores

que serão inseridos a seguir. Para executarmos a *query*, chamamos o método `execute` do objeto `cursor`, passando a string SQL e os valores a serem inseridos através de uma tupla:

```
cursor.execute(comando, ('Zaphod', '9999-9999', '0000-0000',
'zaphod@magrathea.xyz'))
```

Repare que poderíamos utilizado o operador de formatação de strings (%) para montar o comando SQL e passar os valores para o método `execute`, porém isso não é recomendado pois deixa o aplicativo vulnerável a ataques *SQL Injection*. Para mais informações sobre ataques desse tipo, visite http://pt.wikipedia.org/wiki/SQL_injection.

Como não temos mais nenhum comando para executar no nosso banco de dados, podemos dar o *commit* e fechar a conexão com banco:

```
1  conexao.commit()
2  conexao.close()
```

Se nenhuma mensagem de erro foi exibida, nosso comando foi devidamente executado. Vamos agora consultar nossa tabela Contatos:

```
1  conexao = sqlite3.connect('agenda.sqlite')
2  cursor = conexao.cursor()
3  comando = 'SELECT * FROM Contatos'
4  cursor.execute(comando)
5  contatos = cursor.fetchall()
6  conexao.close()
7  print contatos
```

O código acima deve imprimir algo como `[(1, u'Zaphod', u'9999-9999', u'0000-0000', u'zaphod@magrathea.xyz')]` na tela. Repare que a única diferença da listagem anterior da que executamos nossa cláusula INSERT é que chamamos o método `fetchall` do objeto `cursor` depois de executar o método `execute`. Esse método retorna todos os registros restantes do *result set* como uma lista de tuplas. Vamos agora deixar nossa listagem de contatos mais bem apresentável:

```
1  conexao = sqlite3.connect('agenda.sqlite')
2  cursor = conexao.cursor()
3  comando = 'SELECT * FROM Contatos'
4  cursor.execute(comando)
5  contatos = cursor.fetchall()
6  conexao.close()
7
8  mensagem = u'''\
9  Nome: %s
10 Telefone: %s
11 Celular: %s
12 E-mail: %s
13 -----------------------------'''
14
```

```
15 if len(contatos) > 0:
16    for contato in contatos:
17       print mensagem % (contato[1], contato[2], contato[3],
contato[4])
18 else:
19    print 'Nenhum contato encontrado!'
20
21 raw_input('Pressione qualquer tecla para continuar...')
```

Acima utilizamos a função embutida `len` que retorna o tamanho ou quantidade de itens de um objeto, no caso, a quantidade de tuplas que a lista `contatos` possui. Caso existam itens na lista, iteramos sobre ela e imprimimos seus valores. Além do método `fetchall` existem ainda o `fetchone`, que retorna o próximo item do iterador. Ficou confuso? Iterador? Abordaremos iteradores no próximo capítulo, quando falaremos de Programação Funcional, mas adiantando o assunto, um iterador é um objeto que representa um fluxo de dados, retornando um elemento do fluxo de cada vez. Podemos usar um objeto `Cursor` como um iterador, e utilizaremos o interpretador interativo para exemplificar seu uso:

```
1  >>> import sqlite3
2  >>> conexao = sqlite3.connect(u'C:/pyAgenda/agenda.sqlite')
3  >>> cursor = conexao.cursor()
4  >>> cursor.execute('SELECT * FROM Contatos')
5  <sqlite3.Cursor object at 0x02400B30>
6  >>> for contato in cursor:
7  ...     contato
8  ...
9  (1, u'Zaphod', u'9999-9999', u'0000-0000', u'zaphod@magrathea.xyz')
10 (2, u'Ford', u'9999-0000', u'0000-9999', u'ford@betelgeuse.xyz')
11 >>> cursor.execute('SELECT * FROM Contatos')
12 <sqlite3.Cursor object at 0x02400B30>
13 >>> cursor.fetchone()
14 (1, u'Zaphod', u'9999-9999', u'0000-0000', u'zaphod@magrathea.xyz')
15 >>> cursor.fetchone()
16 (2, u'Ford', u'9999-0000', u'0000-9999', u'ford@betelgeuse.xyz')
17 >>> cursor.fetchone()
18 >>> cursor.execute('SELECT * FROM Contatos')
19 <sqlite3.Cursor object at 0x02400B30>
20 >>> cursor.fetchmany(1)
21 [(1, u'Zaphod', u'9999-9999', u'0000-0000',
u'zaphod@magrathea.xyz')]
22 >>> cursor.fetchmany(1)
23 [(2, u'Ford', u'9999-0000', u'0000-9999', u'ford@betelgeuse.xyz')]
24 >>> cursor.execute('SELECT * FROM Contatos')
25 <sqlite3.Cursor object at 0x02400B30>
26 >>> cursor.fetchone()
27 (1, u'Zaphod', u'9999-9999', u'0000-0000', u'zaphod@magrathea.xyz')
28 >>> cursor.fetchall()
29 [(2, u'Ford', u'9999-0000', u'0000-9999', u'ford@betelgeuse.xyz')]
```

Na linha 4 listamos todos os registros da tabela *Contatos* através do método `execute` do objeto `cursor`, e nas linhas 6 e 7 utilizamos `cursor` como um iterador e iteramos sobre ele através de um laço `for`. Note que na linha 11 listamos novamente todos os registros da

tabela *Contatos* – veremos o porquê disso dentro de instantes. Repare na utilização do método `fetchone` nas linhas 13, 15 e 17. Na linha 13 o método retornou o primeiro registro do *result set*, na 15, retornou o segundo, e na 17 não retornou nada, isso porque não há mais registros na tabela.

Nas linhas 20 e 22 utilizamos mais um método novo – o `fetchmany` – que recebe como argumento o número de registros a serem buscados. Agora repare nas linhas 26 e 28. Na primeira executamos `fetchone`, que retorna o primeiro registro, em seguida executamos `fetchall`, que retornou o segundo registro. Estranho? Não exatamente. Lembra-se que vimos que o método `fetchall` retorna todos os registros restantes do *result set*? Entenda esse "restantes" como "ainda não retornados".

Apesar de termos demonstrado aqui como executar *queries* no SQLite, o procedimento é muito semelhante em outros SGBDs, graças a PEP 249, que descreve as diretrizes que os DBIs para Python devem seguir. A maioria dos DBIs seguem essas diretrizes, portanto, conectar-se e executar uma *query* é bem parecido no SQLite, MySQL, PostgreSQL, Oracle ou qualquer outro SGBD que possua um DBI para Python que siga as diretrizes descritas na PEP 249. Para obter uma lista dos DBIs disponíveis para Python, acesse http://wiki.python.org/moin/DatabaseInterfaces.

Para mais informações sobre o módulo `sqlite3`, visite sua documentação oficial no endereço http://www.python.org/documentacaodosqlite3.

Paradigmas de programação

Um Paradigma de Programação é o que determina o ponto de vista que o programador tem sobre a estruturação e execução de um programa. Há diversos paradigmas, dentre eles destaca-se o Procedural, o Orientado a Objetos e o Funcional - cada um suporta um conjunto de conceitos que os tornam melhor para um determinado tipo de problema. A utilização de ferramentas corretas aumenta a produtividade de desenvolvimento, facilidade na manutenção e confiabilidade do código.

Os projetores de algumas linguagens de programação escolheram enfatizar um paradigma particular, como é o caso de SmallTalk. Isto torna difícil desenvolver programas utilizando uma abordagem diferente, aumentando tanto dificuldade na escrita do código quanto na manutenção, tornando o código mais suscetível a erros. Outras linguagens são multi-paradigmas – suportam vários paradigmas de programação, como é o caso de Python, C++ e Lisp – isto permite aos programadores escreverem programas e bibliotecas que utilizem tanto código Procedural quanto Orientado a Objetos e Funcional. Por exemplo, a interface gráfica pode ser escrita utilizando a programação Orientada a Objetos, enquanto o processamento é Procedural ou Funcional.

Paradigma Imperativo

O paradigma Imperativo baseia-se no conceito de comandos e atualização de variáveis. Como os programas são escritos para modelar processos e objetos do mundo real e tais objetos freqüentemente possuem estados que variam com o tempo, variáveis naturalmente modelam tais objetos. Além disso, todas as arquiteturas de computadores existentes hoje se baseiam na arquitetura de *von Neumann*, caracterizada pelo acesso direto e possibilidade de alteração de valores armazenados em posições de memória. Isso torna os ambientes de execução das linguagens Imperativas bastante eficientes.

Os projetos das linguagens de programação Imperativas foram muito influenciados na arquitetura de computadores de *von Neumann*, introduzida em meados dos anos 40 e a predominante ainda hoje. A arquitetura de *von Neumann* é o modelo criado por *John von Neumann*, que usa uma unidade central de processamento (CPU), e uma estrutura de armazenamento separada (memória) que armazena tanto instruções (programas) quanto dados. Linguagens imperativas foram influenciadas quanto a:

- Estados (variáveis) – que representam as células de memória que armazenam os dados antes e depois do processamento;

- Ordens seqüenciais – refletindo a CPU seqüencial;
- Instruções de atribuição – representando a transmissão dos dados entre a memória e o processador.

A listagem abaixo demonstra um algoritmo de ordenação de vetores (listas), chamado *Bubble Sort*, utilizando o paradigma Imperativo:

```
1  lista = [10, 5, 7, 0, 9, 8, 1, 4, 3, 6, 2]
2  lista_ord = lista[:]        # Copia todos os elementos da lista
3
4  for i in range(len(lista_ord) - 1, 0, -1):
5     for j in range(0, i):
6        if lista_ord[j] < lista_ord[j + 1]:
7           lista_ord[j], lista_ord[j + 1] = lista_ord[j + 1],
lista_ord[j]
8
9  print lista_ord
```

Na linha 1 definimos uma lista desordenada de inteiros. Na linha 2, copiamos todos os elementos da lista desordenada para a que vamos ordenar. Na linha 4 criamos um `for` que itera do último elemento da lista até o primeiro e na linha 5 declaramos outro `for`, aninhado com o anterior, que itera de 0 até i. As sextas e sétimas linhas realizam as comparações e alteram das posições dos elementos da lista.

Tanto a Programação Orientada a Objetos, quanto o paradigma Procedural são paradigmas imperativos, e os veremos a seguir.

Paradigma Procedural

O Paradigma Procedural é um paradigma Imperativo baseado no conceito de procedimentos, também chamados de rotinas, métodos ou simplesmente de funções, sendo os dois últimos utilizados na documentação oficial da linguagem Python. A programação procedural é geralmente uma escolha melhor que programação seqüencial e não-estruturada em muitas situações que envolvem uma complexidade média e requerem facilidade de manutenção. Possíveis benefícios são:

- A habilidade de reutilizar o mesmo código em diferentes lugares no programa sem copiá-lo;
- Uma forma mais fácil de organizar o fluxo do programa;
- A habilidade de ser fortemente modular e estruturado.

Abaixo segue um exemplo do algoritmo de ordenação *Bubble Sort* em Python utilizando o paradigma Procedural:

```
1  def bubble_sort(lista):
```

```
2     for i in range(len(lista) - 1, 0, -1):
3        for j in range(0, i):
4           if lista[j] < lista[j + 1]:
5              lista[j], lista[j + 1] = lista[j + 1], lista[j]
6
7     return lista
8
9  lista = [10, 5, 7, 0, 9, 8, 1, 4, 3, 6, 2]
10 lista_ord = bubble_sort(lista_1)
11
12 print lista_ord
```

Observe que no exemplo de código anterior, criamos uma função – chamada `bubble_sort`, que recebe uma lista como argumento. Dentro dela, organizamos a lista e a retornamos, armazenando-a em outra variável – `lista_ord`. Quando precisarmos organizar outra lista, basta chamar a função `bubble_sort` passando a lista como argumento e recebendo outra organizada.

O algoritmo utilizado no exemplo anterior foi o *Bubble Sort*, que já tinha sido utilizado no exemplo de ordenação de lista utilizando o paradigma não-estruturado. Em suma, a única modificação realizada foi a utilização de uma função para modularizar o código de ordenação, e esta é a única diferença dentre os dois paradigmas citados. Para Vujoševic-Janicic (2008), quando a programação imperativa é combinada com sub-programas (funções), ela é chamada de programação Procedural. Apesar de os paradigmas serem utilizados como sinônimos, o uso de funções tem um efeito muito grande em como os programas se parecem e como são construídos. A partir de meados dos anos 60, a programação estruturada promove técnicas para melhorar a manutenção e qualidade dos programas imperativos como um todo, sendo a Programação Orientada a Objetos, que veremos a seguir, uma extensão dessa abordagem.

Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) é um paradigma de programação baseado na composição e interação entre objetos. Um objeto consiste de um conjunto de operações encapsuladas (métodos) e um "estado" (determinado pelo valor dos seus atributos) que grava e recupera os efeitos destas operações.

A idéia por trás da Programação Orientada a Objetos é tentar aproximar a modelagem dos dados e processos (mundo virtual) ao mundo real que é baseado em objetos, como entidades concretas – carro, cliente ou um arquivo de computador – ou entidades conceituais, como uma estratégia de jogo ou uma política de escalonamento de um sistema operacional.

As definições – atributos e métodos – que os objetos devem possuir são definidos em classes. Uma classe chamada `Cliente` em um sistema de locadora de vídeos, por exemplo,

pode possuir atributos como `codigo`, `nome`, `telefone` e `endereco`, assim como os métodos `insere` e `altera`. Um objeto – chamado também de instância de uma classe – é uma materialização de uma classe. Ao instanciar um objeto de uma classe, como `Cliente` do exemplo anterior, podemos atribuir o valor "Zaphod" ao campo `nome`, 42 ao `codigo`, "9999-0000" ao `telefone` e "Rua Magrathea, 1024" ao `endereco`.

Classes *new-style* e *old-style*

Há dois tipos de classes no Python: as *old-style* e as *new-style*. Até a versão 2.1 do Python as classes *old-style* eram as únicas disponíveis e, até então, o conceito de classe não era relacionado ao conceito de tipo, ou seja, se `x` é uma instância de uma classe *old-style*, `x.__class__` designa o nome da classe de `x`, mas `type(x)` sempre retornava `<type 'instance'>`, o que refletia o fato de que todas as instâncias de classes *old-styles*, independentemente de suas classes, eram implementadas com um tipo embutido chamado `instance`.

As classes *new-style* surgiram na versão 2.2 do Python com o intuito de unificar classes e tipos. Classes *new-style* são, nada mais nada menos que tipos definidos pelo usuário, portanto, `type(x)` é o mesmo que `x.__class__`. Essa unificação trouxe alguns benefícios imediatos, como por exemplo, a possibilidade da utilização de Propriedades, que veremos mais adiante e *metaclasses*, assunto que não veremos nessa apostila.

Por razões de compatibilidade as classes são *old-style* por padrão. Classes *new-style* são criadas especificando-se outra classe *new-style* como classe-base. Veremos mais sobre isso adiante, em Herança. Classes *old-style* existem somente nas versões 2.x do Python, na 3.0 elas já foram removidas, portanto, é recomendado criar classes *new-style* a não ser que você tenha códigos que precisem rodar em versões anteriores a 2.2.

Criando classes

Uma classe é definida na forma:

```
class NomeClasse:
   <atributos e métodos>
```

Os nomes das classes devem seguir as regras de identificadores explicadas na Introdução da apostila, ou seja, devem ser iniciados por letras e sublinhados e conter letras, números e sublinhados. Exemplo:

```
1  class ClasseSimples(object):
2     pass
```

Acima definimos uma classe que não possui nenhum atributo nem método – a palavra-chave `pass` foi utilizada para suprir a necessidade de pelo menos uma expressão dentro da classe.

Note que após o nome da classe inserimos entre parênteses a palavra `object`. Como veremos mais a diante, essa é a sintaxe para herdar uma classe de outra (ou outras) no Python. O que fizemos na classe anterior foi derivar a classe `ClasseSimples` da classe `object` – que é uma classe *new-style* – definindo então outra classe *new-style*.

O acesso aos atributos e métodos de uma classe é realizado da seguinte forma:

```
1  instancia.atributo
2  instancia.metodo()
```

A seguir veremos exemplos de como acessar atributos e métodos de uma classe.

Variáveis de instância

As variáveis de instância são os campos que armazenam os dados dos objetos, no caso da classe `Cliente` são os atributos `codigo`, `nome`, `telefone` e `endereco`. Abaixo segue a definição da classe `Cliente` para demonstrar a utilização das variáveis de instância:

```
1  class Cliente(object):
2     def __init__(self, codigo, nome, telefone, endereco):
3        self.codigo = codigo
4        self.nome = nome
5        self.telefone = telefone
6        self.endereco = endereco
7
8  cliente1 = Cliente(42, 'Zaphod', '0000-9999', 'Rua Magrathea, S/N')
9  cliente2 = Cliente(1024, 'Ford', '9999-0000', 'Av. Jelz, 42')
```

A função `__init__` definida no exemplo acima é uma função especial, chamada de inicializador, e é executada quando o objeto é criado. A função `__init__` nessa classe define 5 parâmetros – o parâmetro obrigatório `self` e quatro parâmetros pelos quais passamos os valores para os campos `codigo`, `nome`, `telefone` e `endereco` dos objetos que estamos instanciando nas linhas 8 e 9. O parâmetro `self` representa a instância da classe e o utilizamos nas linhas 3, 4, 5 e 6 para definir as variáveis de instância da classe `Cliente` seguindo a mesma forma de acesso de qualquer objeto – `instancia.atributo`. O parâmetro `self` deve ser o primeiro da lista de parâmetros, porém seu nome é uma convenção – ele pode ter qualquer nome, como `instancia` ou `this` – mas é uma boa prática manter o nome `self`.

Nas linhas 8 e 9 instanciamos dois objetos da classe `Cliente`, passando valores para os parâmetros definidos no inicializador. Note que não há necessidade de passar um valor para

o parâmetro `self`, isto é porque o interpretador já fornece um valor automaticamente para ele. No inicializador, o parâmetro `self` representa o objeto recentemente instanciado, e para quem já teve contato com Java, C++, C# ou PHP, ele é como o `this` para acessar de dentro da classe seus atributos e métodos da instância.

Apesar de as classes definirem quais atributos (variáveis) e métodos os objetos terão em comum, estes podem ser definidos dinamicamente, ou seja, podem ser consultados, incluídos e excluídos em tempo de execução – veremos isso mais adiante.

Variáveis de classe

São variáveis compartilhadas entre as instâncias de uma classe. Um exemplo disso é uma variável que armazena a quantidade de objetos instanciados da classe. Exemplo:

```
1  class Cliente(object):
2     instancias = 0
3     def __init__(self, codigo, nome, telefone, endereco):
4        Cliente.instancias += 1
5        self.codigo = codigo
6        self.nome = nome
7        self.telefone = telefone
8        self.endereco = endereco
9
10 cliente1 = Cliente(42, 'Zaphod', '0000-9999', 'Rua Magrathea, 1')
11 cliente2 = Cliente(1024, 'Ford', '9999-0000', 'Av. Jelz, 42')
12 print 'Quantidade de instancias: ', Cliente.instancias
```

Modificamos o exemplo anterior incluindo a variável de classe `instancias`. Quando um objeto é instanciado, o método `__init__` é chamado e esse incrementa o valor da variável de classe `instancias` em 1 na linha 4. Repare que a variável de classe é definida dentro da classe, mas fora da função `__init__`, e para acessá-la é utilizado o nome da classe seguido de um ponto (.) e o nome da variável. Para quem já teve contato com Java ou C#, as variáveis de classe são similares às variáveis estáticas.

Métodos

Métodos são funções contidas em uma classe e dividem-se em métodos de classe, de instância e estáticos.

Métodos de instância

Métodos de objeto ou instância podem referenciar variáveis de objeto e chamar outros métodos de instância e para isso devem receber o parâmetro `self` para representar a instância da classe. Exemplo:

```
1  class Cliente(object):
2    def __init__(self, codigo, nome, telefone, endereco):
3      self.codigo = codigo
```

```
4       self.nome = nome
5       self.telefone = telefone
6       self.endereco = endereco
7
8     def insere(self):
9        print 'Cliente #%i - %s inserido!' % (self.codigo, self.nome)
10
11 cliente1 = Cliente(42, 'Ford Prefect', '0000-9999', 'Betelgeuse 7')
12 cliente1.insere()  # Imprime 'Contato #42 - Ford Prefect inserido!'
```

No exemplo acima, retiramos a variável de classe `instancias` e adicionamos o método `insere`, que recebe somente o parâmetro obrigatório `self`, e imprime na tela o cliente que foi inserido. Note que utilizamos o parâmetro `self` para exibir os valores das variáveis de instância da classe `Cliente` na linha 9 e, assim como no método especial `__init__`, não precisamos passar um valor ele. O parâmetro `self` nos métodos de instância é utilizado pelo interpretador para passar a instância pela qual chamamos os métodos. Para constatar o mesmo, basta criar um método que recebe um parâmetro além de `self` e chamar esse método sem passar um valor – uma mensagem de erro é exibida dizendo que o método recebe dois parâmetros e que só um foi fornecido.

Métodos de classe

Os métodos de classe são similares aos métodos de instância, porém só podem referenciar as variáveis e métodos de classe. Exemplo:

```
1  class Cliente(object):
2     instancias = 0
3     def __init__(self, codigo, nome, telefone, endereco):
4        Cliente.instancias += 1
5        self.codigo = codigo
6        self.nome = nome
7        self.telefone = telefone
8        self.endereco = endereco
9
10    def insere(self):
11       print 'Cliente #%i - %s inserido!' % (self.codigo, self.nome)
12
13    @classmethod
14    def imprime_instancias(cls, formatado = False):
15       if formatado:
16          print 'N° de instâncias de Cliente: ', cls.instancias
17       else: print cls.instancias
18
19 cliente1 = Cliente(42, 'Osmiro', '0000-9999', 'Rua Magrathea, 1')
20 cliente1.imprime_instancias()              # Imprime 1
21 Cliente.imprime_instancias()              # Imprime 1
```

O método `imprime_instancias` recebe dois parâmetros – o primeiro é `cls` que representa a classe em si, parâmetro pelo qual acessamos a variável de classe `instancias`. Esse parâmetro é obrigatório e deve ser o primeiro da lista de parâmetros, assim como o `self`

em métodos de instância. O parâmetro `cls` possui esse nome por convenção, e assim como `self`, pode ter seu nome alterado apesar de não ser uma boa prática.

Repare que utilizamos o decorador `@classmethod` na linha 13 para transformar o método `imprime_instancias` em um método de classe.

Métodos estáticos

Métodos estáticos não possuem nenhuma ligação com os atributos do objeto ou da classe, e por isso, não podem ser executados a partir de um objeto. A diferença entre os métodos vistos anteriormente e os métodos estáticos é que esses não recebem nenhum parâmetro especial – como `self` e `cls` - eles são como funções normais dentro de uma classe e são chamados através da classe, não do objeto. Exemplo:

```
1  class Cliente(object):
2     instancias = 0
3     def __init__(self, codigo, nome, telefone, endereco):
4        Cliente.instancias += 1
5        self.codigo = codigo
6        self.nome = nome
7        self.telefone = telefone
8        self.endereco = endereco
9
10    def insere(self):
11       print 'Cliente #%i - %s inserido!' % (self.codigo, self.nome)
12
13    @classmethod
14    def imprime_instancias(cls, formatado = False):
15       if formatado:
16          print 'N° de instâncias de Cliente: ', cls.instancias
17       else: print cls.instancias
18
19    @staticmethod
20    def valida(codigo, nome, telefone, endereco):
21       try:
22          if int(codigo) >= 0 and len(str(nome)) > 0 and \
23             len(str(telefone)) > 0 and len(str(endereco)):
24             return True
25          else:
26             return False
27       except:
28          return False
29
30 print Cliente.valida(1, 'Zaphod', '(00)9999-9999', 'Rua Magrathea,
42')          # Imprime True
31 print Cliente.valida(1, 'Ford', '', '')         # Imprime False
```

Note que, assim como usamos o decorador `@classmethod` para criar um método estático, utilizamos o `@staticmethod` para transformar o método em estático.

Atributos e métodos dinâmicos

No Python podemos adicionar atributos e métodos em uma instância diretamente, sem precisar defini-las em uma classe. Exemplo seguindo a classe `Cliente` definida no exemplo anterior:

```
1  cliente1 = Cliente(42, 'Ford Prefect', '0000-9999', 'Betelgeuse 7')
2  cliente1.email = 'fordprefect@xyz.com'
3  cliente1.celular = '0099-9900'
4
5  def exclui(self):
6     print 'Cliente #%i - %s excluido!' % (self.codigo, self.nome)
7
8  cliente1.exclui = exclui
```

Adicionamos dois atributos e o método `exclui` à instância `cliente1` nas linhas 2, 3 e 8. Além de adicionar, podemos consultar e excluir atributos com a ajuda de algumas funções embutidas:

- `hastattr(instancia, 'atributo')`: Verifica se atributo existe em uma instância. Retorna `True` caso exista e `False` caso não;
- `getattr(instancia, 'atributo')`: Retorna o valor de atributo da instância;
- `setattr(instancia, 'atributo', 'valor')`: Passa valor para atributo da instância;
- `delattr(instancia, 'atributo')`: Remove atributo da instância.

Exemplos:

```
1  cliente1 = Cliente(42, 'Ford Prefect', '0000-9999', 'Betelgeuse 7')
2  cliente1.email = 'fordprefect@@xyz.com '
3  if not hasattr(cliente1, 'celular'):
4     setattr(cliente1, 'celular', '0099-9900')
5  print getattr(cliente1, 'celular')     # Imprime '0099-9900'
6  delattr(cliente1, 'celular')
```

Note que na linha 2 adicionamos o atributo `email` ao objeto `cliente1` por meio de atribuição, na linha 4 adicionamos o atributo `celular` pela função `setattr` – tanto pela atribuição quanto pela função `setattr` o resultado é o mesmo. O mesmo acontece com a função `getattr`, onde poderíamos sem problemas fazer:

```
print cliente1.celular     # Imprime '0099-9900'
```

Como tudo em Python é um objeto, podemos fazer isso, por exemplo:

```
1  def ola_mundo():
2     return "Ola mundo"
3
```

```
4  ola_mundo.atributo = "Atributo de ola_mundo"
5  print ola_mundo.atributo    # Imprime Atributo de ola_mundo
```

No exemplo acima definimos dinamicamente um atributo à função `ola_mundo` e o recuperamos como qualquer atributo de objeto – `instancia.atributo`. Repare que não executamos a função criada anteriormente, caso fizéssemos, por exemplo, `ola_mundo().atributo`, a exceção `AttributeError` seria acionada, já que o tipo `str` (do valor retornado pela função) não possui o atributo que tentamos recuperar.

Referências a objetos

Sabemos que variáveis são utilizadas para armazenar valores, e estivemos fazendo isso por todo o decorrer dessa apostila. Com instâncias de classes a coisa funciona de forma um pouco diferente – não é a instância propriamente dita que é armazenada, porém uma referência a ela.

No capítulo de Operadores vimos os operadores `is` e `is not`, que usamos para comparar se dois objetos são iguais, ou seja, se as variáveis que os armazenam apontam para o mesmo lugar na memória. Repare no seguinte exemplo:

```
1  >>> class Teste(object):
2  ...     def __init__(self):
3  ...         self.atributo = 0
4  ...
5  >>> x = Teste()
6  >>> x.atributo = 10
7  >>> y = x
8  >>> x.atributo
9  10
10 >>> y.atributo
11 10
12 >>> x.atributo = 20
13 >>> y.atributo
14 20
```

Nesse exemplo criamos uma classe no interpretador interativo chamada de `Teste`, e por meio do método `__init__` definimos o campo `atributo`. Na linha 5 instanciamos nossa classe e armazenamos uma referência à variável `x`, e atribuímos `10` ao seu único atributo. Em seguida, na linha 7, informamos que `y` é igual a `x`, ou seja, a variável `y` "aponta" para o mesmo objeto em memória que `x`. A graça começa aqui: na linha 12 mudamos o valor de `atributo` de 10 para 20, e o mesmo pode ser constatado na linha 13 – o campo `atributo` de `y` também possui o valor 20.

É importante notar que alterar o valor de 10 para 20 em `x` não mudou a variável `y` - como as duas referenciam o mesmo objeto, se ele é modificado, suas alterações podem ser percebidas em quaiquer variáveis que apontam para ele.

Assim como podemos criar variáveis e armazenar valores e referências a elas, podemos também excluí-las através do comando `del`. Tomando como base o exemplo anterior, podemos excluir a variável `x`, deixando somente `y` referenciando um objeto `Teste` em memória:

```
>>> del x
>>> x.atributo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> y.atributo
20
```

Herança

A Herança é um dos pilares da Programação Orientada a Objeto. Com ela, pode-se criar uma classe baseando-se em outra, ou outras, já que Python suporta herança múltipla. Criar uma classe baseada em outra faz com que a classe herdada, ou classe-filha, literalmente herde todos os atributos e métodos das classes-base (chamadas também de superclasses, dentre outros termos). Para mencionar a, ou as superclasses, basta listá-las entre parênteses, separando-as por vírgulas depois do nome da classe-filha (ou subclasse / classes especializada). Para exemplificar, vamos modificar a estrutura da classe `Cliente`, criando uma classe `Contato` e especializando a classe `Cliente`:

```
class Contato(object):
    'Classe Contato'
    def __init__(self, codigo, nome, telefone, email):
        self.codigo = codigo
        self.nome = nome
        self.telefone = telefone
        self.email = email

    def insere(self):
        print 'Contato # %i - %s' % (self.codigo, self.nome)

    def imprime(self):
        print u"""\
Código: %i
Nome: %s
Telefone: %s
Email: %s""" % (self.codigo, self.nome, self.telefone, self.email)

class Cliente(Contato):
    'Classe Cliente, que herda da classe Contato'
    def __init__(self, codigo, nome, telefone, email, cpf,
cartao_credito):
        Contato.__init__(self, codigo, nome, telefone, email)
        self.cpf = cpf
        self.cartao_credito = cartao_credito

 contato1 = Contato(1, 'Ford', '9999-0000', 'ford@pre.fect')
```

```
27  cliente1 = Cliente(42, 'Zaphod', '0000-9999', 'zaphod@xyz', '128',
'256')
28  cliente1.imprime()
```

No exemplo acima, definimos na classe `Contato` os atributos `codigo`, `nome`, `telefone`, `email` e `endereco`, o método `insere` e `imprime`. Derivamos da classe `Contato` a classe `Cliente`, que adiciona os atributos `cpf` e `cartao_credito`. No inicializador da classe `Cliente` chamamos o inicializador da classe `Contato`, passando os parâmetros recebidos no `__init__` de `Cliente` para o `__init__` de `Contato`. Note que precisamos passar `self` explicitamente para o inicializador da classe-base `Contato`. Ao chamarmos o método `imprime` do objeto `cliente1` o código, nome, telefone e email são impressos, mas faltam o CPF e o nº do cartão de crédito presentes na classe `Cliente`. Para isso, precisamos sobrescrever o método `imprime` da classe herdada `Cliente`. Veremos isso a seguir.

Sobrescrita de métodos (*Method overriding*)

Sobrescrever um método permite a uma subclasse fornecer uma implementação específica de um método já definido em uma superclasse. A implementação desse método na subclasse "substitui" a implementação herdada da superclasse fornecendo um método com o mesmo nome e mesmos parâmetros. No caso da subclasse `Cliente`, precisamos imprimir os campos `cpf` e `cartao_credito`, além daqueles já presentes na implementação de `imprime` da classe `Contato`. Seguem as alterações abaixo:

```
1  class Cliente(Contato):
2      'Classe Cliente, que herda da classe Contato'
3      def __init__(self, codigo, nome, telefone, email, cpf,
cartao_credito):
4          Contato.__init__(self, codigo, nome, telefone, email)
5          self.cpf = cpf
6          self.cartao_credito = cartao_credito
7
8      def imprime(self):
9          print u"""\
10 Código: %i
11 Nome: %s
12 Telefone: %s
13 Email: %s
14 self.cpf: %s
15 self.cartao_credito: %s""" % (self.codigo, self.nome,
self.telefone, self.email)
16
17 cliente1 = Cliente(42, 'Arthur Dent', '0000-9999',
'arthur@xyz.com', '123', '558')
18 cliente1.imprime()
```

No exemplo acima redefinimos o método `imprime`, fazendo com que ele imprima além dos dados presentes na sua implementação da classe `Contato`, os dados CPF e Nº do

cartão de crédito do cliente. Quando chamamos o método `imprime` na linha 18, é a implementação do método definida na classe `Cliente` que é executada, e não o método herdado da classe `Contato`.

Sobrecarga de métodos (*Method overloading*)

Em linguagens como Java, C# e C++ é possível sobrecarregar funções e métodos definindo várias versões dele baseado na sua lista de parâmetros, assim, podemos definir um método que não recebe parâmetros, outro que recebe um parâmetro, outro que recebe dois, e assim por diante, todos com o mesmo nome.

Python ao contrário dessas linguagens não suporta esse recurso, porém, podemos utilizar os parâmetros com argumentos *default,* vistos anteriormente, para uma implementação bastante similar. Como Python não suporta o sobrecarregamento de métodos, definir um método, por exemplo, `imprime()`, e depois definir um `imprime(parametro)`, faz com que este útimo sobrescreva o primeiro.

Atenção

Há bastante confusão entre os termos *sobrecarregamento de métodos* (ou *method overloading*) e sobrescrita de métodos (ou *method overriding*) – eles são termos completamente distintos, portanto, não os confunda.

Atributos e métodos embutidos

Como vimos anteriormente, módulos possuem atributos embutidos – como `__name__` e `__file__`. Toda classe e objeto criado em Python também possui alguns atributos e métodos embutidos que podem ser acessados como qualquer outro membro:

Atributos

- `__dict__`: Retorna um dicionário contendo o *namespace* da classe ou objeto;
- `__doc__`: String de documentação da classe;
- `__name__`: Retorna o nome da classe;
- `__module__`: O nome do módulo em que a classe foi definida;
- `__bases__`: Retorna uma tupla com os nomes das classes-base. Caso a classe não tenha sido derivada de uma classe explicitamente a tupla estará vazia.

Abaixo segue um exemplo demonstrando a utilização dos atributos embutidos seguindo a classe `Cliente` – derivada de `Contato` - definida no último exemplo:

```
1  cliente1 = Cliente(42, 'Arthur Dent', '0000-9999', 'R. Bet., 7')
2  atrs_cliente1 = cliente1.__dict__
```

```
3  atrs_Cliente = Cliente.__dict__
4
5  print "Objeto 'cliente1'"
6  print 'Namespace:'
7  for atr in atrs_cliente1:
8     print atr, '->', atrs_cliente1[atr]
9
10 print "Classe 'Cliente'"
11 print 'DocString:', Cliente.__doc__
12 print 'Classes-base:', Cliente.__bases__
13 print 'Namespace:'
14 for atr in atrs_Cliente:
15    print atr, '->', atrs_Cliente[atr]
```

Nas linhas 7 e 8 iteramos no dicionário `atrs_cliente`, que contém o *namespace* do objeto `cliente1` e exibindo-o. Nas linhas 14 e 15, iteramos e exibimos o *namespace* da classe `Cliente`. Repare que ao exibirmos o *namespace* do objeto `cliente1`, somente suas variáveis de instância foram exibidas, e ao exibirmos o *namespace* da classe `Cliente`, além das variáveis de instância, o método `insere`, `__init__`, atributos e métodos embutidos foram exibidos. Se tivéssemos declarado uma variável de classe, como `instancias` – definida em um dos primeiros exemplos - ela também seria exibida na listagem.

Métodos

Os métodos embutidos de classes possuem um comportamento padrão, e como quaisquer outros métodos, podem (e às vezes devem) ser sobrescritos. Vejamos alguns deles:

- `__init__`: Chamado quando a instância é criada;
- `__del__`: Chamado quando a instância está para ser destruída, chamado de destrutor;
- `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__` e `__ge__`: Chamados de método de comparação rica. São executados por operadores de comparação, respectivamente `<`, `<=`, `==`, `!=`, `>` e `>=`;
- `__cmp__`: Executado pelos operadores de comparação quando os operadores de comparação rica descritos anteriormente não foram definidos;
- `__repr__` e `__str__`: Definem como os objetos de uma classe devem ser representados em forma de string – veremos detalhadamente esses métodos na próxima seção.

  Para mais informações sobre os métodos acima, e outros não descritos nesta apostila, consulte a documentação oficial da linguagem, em http://docs.python.org/reference/datamodel.html?#basic-customization.

Representação de objetos

Lembra que no primeiro capítulo da apostila falamos que normalmente não precisamos chamar o comando `print` no interpretador interativo do Python, já que ele “se

vira" para imprimir o valor que mais faz sentido de acordo com seu tipo? O exemplo que demos foi:

```
1  >>> 'String'
2  'String'
3  >>> 1+1
4  2
5  >>> object()
6  <object object at 0x006F14B0>
```

No exemplo acima, para o interpretador do Python saber como representar cada tipo de objeto, ele utiliza seus métodos `__repr__`. O motivo pelo qual o `object` do exemplo acima imprime seu endereço de memória é bastante simples – esse é o comportamento padrão desse método, portanto, é bem comum sobrescrevê-lo em nossas classes, já que seu comportamento padrão não é necessariamente útil na maioria dos casos.

Além do método `__repr__` existe ainda o `__str__`, e apesar de aparentemente possuírem a mesma função, há diferenças fundamentais entre eles, como veremos a seguir.

*_repr_*

De acordo com a documentação oficial do Python, o método `__repr__` é chamado pela função embutida `repr` para criar a representação "oficial" de um objeto. O objetivo desse método é ser livre de ambiguidade, a fim de representar um objeto em específico, em geral na forma `<...descrição útil...>`. Apesar de pouco utilizado, nesse método podemos retornar uma expressão válida do Python com o fim de recriar um objeto com o mesmo valor, através da função embutida `eval`.

*_str_*

O método `__str__` é chamado pelas funções `str` e `print` e é utilizada para criar a representação "informal" de um objeto. Uma das características que difere o método `__repr__` do `__str__` é que esse último deve retornar uma representação mais concisa e legível, destinada aos usuários, não aos desenvolvedores, portanto, o método `__str__` não tem que ser uma expressão válida do Python. Caso `__str__` não seja sobrescrito pela classe, mas `__repr__` o seja, `__str__` possuirá o mesmo comportamento deste.

*_repr_ vs _str_*

Como regra geral, o método `__repr__` deve ser focado para os desenvolvedores, a fim de representar um objeto em específico, enquanto o `__str__` deve ser focado para usuários do aplicativo, a fim de exibir uma informação mais legível e concisa.

Tomemos como exemplo a biblioteca-padrão do Python, com ambas as representações de um objeto `datetime`, do módulo `datetime`:

```
>>> import datetime
>>> str(datetime.datetime.now())
'2011-11-16 14:51:27.597000'
>>> repr(datetime.datetime.now())
'datetime.datetime(2011, 11, 16, 14, 51, 32, 868000)'
```

Enquanto a representação `str` exibe de forma concisa as informações de data e hora, a representação `repr` não deixa dúvida que o objeto em questão é um `datetime.datetime`, além de exibir os dados do objeto, o que nos auxilia bastante quando criamos testes unitários com auxílio do módulo `unittest`, ou gravamos eventos do sistema em um arquivo de *log*, utilizando o módulo `logging`, ambos da biblioteca-padrão do Python.

Tips and tricks

Quando um objeto está contido em um *container*, como uma sequencia – lista ou tupla, por exemplo – ele será representado pelo método `repr`, e não `str`, assim, se sobrescrevermos o método `__str__` e não o `__repr__`, o comportamento padrão deste último é utilizado. Exemplo:

```
>>> class Foo:
      def __init__(self, id):
         self.id = id
      def __str__(self):
       return '<foo # %i>' % self.id
>>> str(Foo(1))
'<foo # 1>'
>>> [ Foo(1), Foo(2) ]
[<__main__.Foo instance at 0x021869B8>, <__main__.Foo instance at
0x02186828>]
```

As classes `Contato` e `Cliente` já com suas implementações dos métodos `__repr__` e `__str__` poderiam ficar mais ou menos da seguinte forma:

```
class Contato(object):
   'Classe Contato'
   def __init__(self, codigo, nome, telefone, email):
      self.codigo = codigo
      self.nome = nome
      self.telefone = telefone
      self.email = email

   def insere(self):
      print 'Contato # %i - %s' % (self.codigo, self.nome)

   def imprime(self):
      print u"""\
Código: %i
Nome: %s
```

```
16 Telefone: %s
17 Email: %s""" % (self.codigo, self.nome, self.telefone, self.email)
18
19     def __repr__(self):
20       return u'<Contato # %i - %s>' % (self.codigo, self.nome)
21
22     def __str__(self):
23       return u'%s' % self.nome
24
25
26 class Cliente(Contato):
27     'Classe Cliente, que herda da classe Contato'
28     def __init__(self, codigo, nome, telefone, email, cpf,
cartao_credito):
29       Contato.__init__(self, codigo, nome, telefone, email)
30       self.cpf = cpf
31       self.cartao_credito = cartao_credito
32
33     def __repr__(self):
34       return u'<Cliente # %i - %s>' % (self.codigo, self.nome)
35
36     def __str__(self):
37       return u'%s' % self.nome
```

Agora que temos nossas próprias implementações de `__repr__` e `__str__` em nossas classes, podemos instanciá-las e imprimir seus objetos de forma direta. Exemplo:

```
1  contato = Contato(42, 'Zaphod', '9999-0000',
'zaphod@beeblebrox.xyz')
2  cliente = Cliente(8, 'Ford', '0000-9999', 'ford@prefect.xyz',
'123', '321')
3  print contato              # Imprime Zaphod
4  print cliente              # Imprime Ford
5  print repr(contato)        # Imprime <Contato # 42 - Zaphod>
6  print repr(cliente)        # Imprime <Cliente # 8 - Ford>
```

Variáveis e métodos privados

Em linguagens de programação como Java, C++ e C#, há palavras-chave que servem de modificadores de acesso, utilizados para liberar ou restringir o acesso às variáveis e métodos de fora da classe e por classes herdadas. Em Python, esse conceito não existe, contudo, há uma convenção na maioria dos códigos escritos em Python: um identificador prefixado por um *underscore* (_) deve ser tratado como um membro não-público.

Isso para pessoas que vêm de outras linguagens – como as previamente citadas – pode parecer um absurdo, mas o pensamento por trás disso é simples e assunto de discussões infindáveis: para que precisamos “esconder” os métodos e atributos de uma classe, se quem os manipulará são programadores (que provavelmente) conhecem a convenção de membros não-públicos? Se o atributo ou método inicia com *underscore*, o programador que fizer uso desse recurso, o faz por conta e risco, já que sabe que ele é de uso interno, não faz parte da sua *interface* e pode retornar um resultado inesperado, principalmente se não conhecer sua

implementação. Python, como você irá perceber à medida que se aprofundar, é uma linguagem muito pouco restritiva.

Propriedades

Propriedades são atributos calculados em tempo de execução, e são criados através da função decoradora `property`. O uso de propriedades permite, entre outras coisas, validar os valores passados aos atributos, criar atributos somente-leitura e mudar de um atributo convencional para uma propriedade sem a necessidade de alterar as aplicações que utilizam a classe. Exemplo:

```
1  class Funcionario(Contato):
2    def __init__(self, codigo, nome, telefone, email, cpf, salario,
bonificacao):
3      Contato.__init__(self, codigo, nome, telefone, email)
4      self.cpf = cpf
5      self._salario = salario
6      self._bonificacao = bonificacao
7
8    def reajusta(self, percentual):
9      self._salario *= percentual
10
11   def reajusta_bonificacao(self, percentual):
12     self._bonificacao *= percentual
13
14   @property
15   def salario(self):
16     return self._salario + self._bonificacao
17
18 f = Funcionario(42, 'Zaphod', '0000-9999', 'zaphod@beeblebrox.xyz',
'000.000.000-00', 3000, 130)
19
20 print u'Salário:', f.salario      # Imprime Salário: 3130
```

Na linha 15 definimos um método chamado `salario` que retorna o salário acrescido da bonificação do funcionário. Esse método é modificado pelo decorador `@property`, que permite que ele seja chamado como qualquer atributo da classe. Note que essa propriedade é somente-leitura, se fizéssemos algo como `f.salario = 3200`, receberíamos uma exceção `AttributeError`.

Podemos também criar uma propriedade pela função `property` diretamente. Exemplo anterior mais detalhado:

```
1  class Funcionario(Contato):
2    def __init__(self, codigo, nome, telefone, email, cpf, salario,
bonificacao):
3      Contato.__init__(self, codigo, nome, telefone, email)
4      self.cpf = cpf
5      self._salario = salario
6      self._bonificacao = bonificacao
7
```

```
8     def reajusta(self, percentual):
9       self._salario *= percentual
10
11    def reajusta_bonificacao(self, percentual):
12      self._bonificacao *= percentual
13
14    def set_salario(self, salario):
15      self._salario = salario
16
17    def get_salario(self):
18      return self._salario + self._bonificacao
19
20    salario = property(get_salario, set_salario)
21
22  f = Funcionario(42, 'Zaphod', '0000-9999', 'zaphod@beeblebrox.xyz',
'000.000.000-00', 3000, 130)
23
24  print u'Salário:', f.salario      # Imprime Salário: 3130
25  f.salario = 3200
26  print u'Salário:', f.salario      # Imprime Salário: 3330
```

No exemplo acima definimos dois métodos – `get_salario`, que retorna o salário do funcionário acrescido da bonificação, e `set_salario` que recebe o salário por argumento e atribui para o atributo `_salario`.

Na linha 20, definimos o atributo `salario` pela função `property`, que recebe os métodos *getter* e *setter* – `get_salario` e `set_salario`. Note que agora a propriedade `salario` deixa de ser somente-leitura, na linha 25 atribuímos o valor 3200 a ela, e o valor de 3330 é exibido na tela quando imprimimos o valor do seu campo `salario`.

O uso de propriedades simplifica muito o uso da classe e é particularmente interessante para quem desenvolve bibliotecas para serem usadas por outras pessoas. Sem as propriedades, o usuário da biblioteca teria que chamar métodos *getter* e *setter* para respectivamente, recuperar e atribuir o salário do funcionário.

Exemplo prático

A seguir demonstraremos a continuação do desenvolvimento do nosso aplicativo de agenda de contatos utilizando todos os conceitos assimilados até agora. No capítulo anterior – Trabalhando com Bancos de Dados – criamos a base de dados SQLite e a tabela que irá armazenar nossos contatos.

Para a interação com os usuários utilizamos o *framework* de interface gráfica nativo do Python, o Tkinter. O Tkinter vem por padrão na maioria das instalações do Python e é, assim como o Python, multiplataforma.

Tenha em mente que não estamos presos ao Tk, podemos desenvolver com GTK, QT, wxWindows, entre outros, cabe somente ao desenvolvedor escolher. Escolhemos utilizar o Tkinter pela sua simplicidade e por já vir incluído na instalação do Python, apesar de não ser

muito boa para desenvolver aplicativos com grande apelo visual, devido à pobreza dos seus componentes visuais. Não nos aprofundaremos muito nem no TK, nem na sintaxe do SQL por serem assuntos bastante extensos para serem explicados nessa apostila.

Criando a classe Contato

Dentro da pasta de nosso aplicativo, crie uma pasta chamada `model` e insira dentro dela um arquivo `__init__.py` para transformá-la em um pacote. Dentro desse pacote, crie um módulo chamado de `contato.py` e insira o seguinte código para a classe `Contato`:

```
1  # -*- coding: latin-1 -*-
2
3  import re
4
5  class Contato(object):
6      '''Classe que representa cada contato no banco de dados.'''
7
8      def __init__(self, codigo=-1, nome='', telefone='', celular='',
email=''):
9          self._codigo = codigo
10         self._nome = nome
11         self._telefone = telefone
12         self._celular = celular
13         self._email = email
14
15     def get_codigo(self):
16         return self._codigo
17
18     def set_codigo(self, codigo):
19         if codigo < 0:
20             raise ValueError(u'Código de contato inválido')
21         else:
22             self._codigo = codigo
23
24     def get_nome(self):
25         return self._nome
26
27     def set_nome(self, nome):
28         if not len(nome):
29             raise ValueError(u'Digite o nome do contato')
30         else:
31             self._nome = nome.strip()
32
33     def get_telefone(self):
34         return self._telefone
35
36     def set_telefone(self, telefone):
37         if not len(telefone):
38             raise ValueError(u'Digite o telefone do contato')
39         elif not re.search(r'^\([0-9]{2}\) [0-9]{4}-[0-9]{4}$',
telefone):
40             raise ValueError(u"Telefone inválido. Digite um número
no formato '(xx) xxxx-xxxx'")
41         else:
42             self._telefone = telefone.strip()
43
44     def get_celular(self):
```

```
45          return self._celular
46
47      def set_celular(self, celular):
48          if len(celular):
49              if not re.search(r'^\([0-9]{2}\) [0-9]{4}-[0-9]{4}$',
celular):
50                  raise ValueError(u"Celular inválido. Digite um
número no formato '(xx) xxxx-xxxx' ou deixe em branco")
51              else:
52                  self._celular = celular.strip()
53
54      def get_email(self):
55          return self._email
56
57      def set_email(self, email):
58          if not len(email):
59              raise ValueError(u'Digite o e-mail do contato')
60          elif not re.search(r'^[a-zA-Z0-9._-]+@([a-zA-Z0-9._-
]+\.)+[a-zA-Z0-9_-]+$', email):
61              raise ValueError(u'E-mail inválido. Digite um e-mail
válido')
62          else:
63              self._email = email.strip()
64
65      def limpa(self):
66          self._codigo = -1
67          self._nome = ""
68          self._email = ""
69          self._celular = ""
70          self._telefone = ""
71
72      def __repr__(self):
73          return '<Contato # %i: %s>' % (self.get_codigo(),
self.get_nome())
74
75      def __str__(self):
76          return self.nome
77      codigo = property(get_codigo, set_codigo)
78      nome = property(get_nome, set_nome)
79      telefone = property(get_telefone, set_telefone)
80      email = property(get_email, set_email)
81      celular = property(get_celular, set_celular)
```

Utilizaremos a classe `Contato` tanto para encapsular as informações do contato a ser inserido, atualizado e excluído, quanto para obter e exibir as informações dos contatos armazenados no banco de dados.

Nos métodos *setter* realizamos todas as validações pertinentes aos dados a serem armazenados, impossibilitando aos usuários inserirem *e-mails* e números de telefone inválidos, bem como deixar de fornecer dados obrigatórios como, como o nome e o telefone do contato. Repare que nos métodos *setter* dos campos obrigatórios nós realizamos algumas comparações, a primeira delas é verificar através da função `len` se o campo foi preenchido. Como `len` retorna `0` para strings vazias, e `0` é avaliado como `False`, então invertemos o

resultado e acionamos a exceção `ValueError` passando a mensagem de erro pertinente como argumento.

Na linha 3 importamos o módulo `re`, que contém objetos para trabalhar com expressões regulares. Uma expressão regular, de acordo com Aurélio Marinho Jargas, é um método formal de se especificar um padrão de texto, utilizado para, entre muitas outras coisas, realizar validações, como de telefones e *e-mails*, fazer substituições, buscas e etc. Nesse caso utilizamos expressões regulares para validar os telefones e *e-mails* dos contatos. Dentre os objetos contidos no módulo `re` estão inclusos as funções `match` e `search`, e utilizamos essa segunda para procurar um padrão, ou *pattern*, em uma determinada string – ambos passados como argumentos para função `search`. O *pattern* definido para validação de telefones valida números no formato (99) 9999-9999.

Está fora do escopo dessa apostila a explicação de Expressões Regulares devido à grande complexidade do assunto. Há ótimos recursos sobre o assunto, tanto impressos como na Internet, como o livro do já citado Aurélio Marinho Jargas e seu *website* http://aurelio.net.

Criando a classe ContatoDAO

Agora, criaremos a classe responsável por lidar com o acesso a dados. Dentro do diretório raiz do nosso aplicativo, crie um pacote chamado `dao`, e dentro dele, um módulo chamado `contato_dao.py`, contendo o código:

```
1  # -*- coding: latin-1 -*-
2
3  import sqlite3
4  import traceback
5
6  from model.contato import Contato
7
8  class ContatoDAO(object):
9      def __init__(self, banco_dados='contatos.sqlite'):
10         self.banco_dados = banco_dados
11
12     def pesquisa(self, contato):
13         sql = 'SELECT * FROM Contatos WHERE codigo = ?'
14         conexao = sqlite3.connect(self.banco_dados)
15         try:
16             cursor = conexao.cursor()
17             cursor.execute(sql, (contato.codigo,))
18             rs = cursor.fetchone()
19
20             if rs is not None:
21                 return Contato(rs[0], rs[1], rs[2], rs[3], rs[4])
22             else:
23                 return None
24         except:
25             raise RuntimeError(u'Erro ao pesquisar contato: ' +
traceback.format_exc())
26         finally:
```

```
27              conexao.close()
28
29      def lista(self):
30          sql = 'SELECT codigo, nome, telefone, celular, email FROM
Contatos'
31          conexao = sqlite3.connect(self.banco_dados)
32          try:
33              cursor = conexao.cursor()
34              cursor.execute(sql)
35
36              contatos = []
37              for r in cursor:
38                  contatos.append(Contato(r[0], r[1], r[2], r[3],
r[4]))
39          except:
40              raise RuntimeError(u'Erro ao listar contatos: ' +
traceback.format_exc())
41          finally:
42              conexao.close()
43
44          return contatos
45
46      def insere(self, contato):
47          sql = 'INSERT INTO Contatos (nome, telefone, celular,
email) VALUES (?, ?, ?, ?)'
48          conexao = sqlite3.connect(self.banco_dados)
49          try:
50              cursor = conexao.cursor()
51              cursor.execute(sql, (contato.nome, contato.telefone,
contato.celular, contato.email))
52              conexao.commit()
53          except:
54              raise RuntimeError(u'Erro ao inserir contato: ' +
traceback.format_exc())
55          finally:
56              conexao.close()
57
58      def atualiza(self, contato):
59          sql = 'UPDATE Contatos SET nome = ?, telefone = ?, celular
= ?, email = ? WHERE codigo = ?'
60          conexao = sqlite3.connect(self.banco_dados)
61          try:
62              cursor = conexao.cursor()
63              cursor.execute(sql, (contato.nome, contato.telefone,
contato.celular, contato.email, contato.codigo))
64              conexao.commit()
65          except:
66              raise RuntimeError(u'Erro ao atualizar contato: ' +
traceback.format_exc())
67          finally:
68              conexao.close()
69
70      def existe(self, contato):
71          sql = 'SELECT COUNT(codigo) FROM Contatos WHERE codigo = ?'
72          conexao = sqlite3.connect(self.banco_dados)
73          try:
74              cursor = conexao.cursor()
75              cursor.execute(sql, (contato.codigo,))
76              rs = cursor.fetchone()[0]
77
78              return rs
```

```
79          except:
80              raise RuntimeError(u'Erro ao atualizar contato: ' +
traceback.format_exc())
81          finally:
82              conexao.close()
83
84      def exclui(self, contato):
85          if not self.existe(contato):
86              raise RuntimeError(u'Contato %s não existe.' %
(contato.nome,))
87              return
88
89          sql = 'DELETE FROM Contatos WHERE codigo = ?'
90          conexao = sqlite3.connect(self.banco_dados)
91          try:
92              cursor = conexao.cursor()
93              cursor.execute(sql, (contato.codigo,))
94              conexao.commit()
95          except:
96              raise RuntimeError(u'Erro ao excluir contato: ' +
traceback.format_exc())
97          finally:
98              conexao.close()
```

Na linha 3 importamos o módulo `sqlite3` para poder trabalhar com bancos de dados SQLite. O módulo `traceback` que importamos na linha 4 fornece uma *interface* que nos permite extrair, formatar e imprimir rastros da pilha de exceções. Dentro dos métodos dessa classe, capturamos todas as exceções (apesar de isso não ser uma boa idéia) e as relançamos como `RuntimeException`, passando a mensagem de erro retornada pela função `format_exc` da exceção capturada. Estamos relançando as exceções, a maioria do módulo `sqlite3`, para que não fiquemos acoplados a esse módulo quando estivermos trabalhando em uma camada superior, como a de apresentação.

O primeiro método que definimos é `__init__` que possui o parâmetro `banco_dados`, com um argumento *default*. Dentro do inicializador da classe definimos o atributo `banco_dados` e passamos o parâmetro de mesmo nome a ele. O atributo `banco_dados` é utilizado pelos métodos `connect` dos objetos `Connection`.

Após o inicializador definimos o método `pesquisa`. Esse método é o responsável por pesquisar um contato no banco de dados com base em seu código e retorná-lo, caso nenhum registro seja encontrado, `None` é retornado.

Em seguida definimos `lista`, método reponsável por listar todos os contatos do banco de dados e retorná-los dentro de uma lista. Repare que dentro do método `append` da lista contida nesse método, instanciamos um objeto `Contato`, passando os dados retornados da consulta para inicializador da classe.

Os métodos `insere`, `atualiza` e `existe` são bastante similares com tudo o que vimos até o momento, por isso não nos aprofundaremos neles. O método `exclui`, primeiro

consulta se o contato já existe no banco de dados, através do método `existe`, caso o contato exista, o exclui. Quando trabalhamos com o módulo sqlite3, precisamos fazer essa verificação antes porque quando executamos um comando SQL DELETE, não temos uma forma de constatar se o comando foi executado com sucesso ou não.

Criando a interface gráfica

Por fim, criaremos a interface com o usuário. No diretório raiz de nosso aplicativo, crie um pacote chamado `view`, e dentro dele crie um módulo chamado `frame_contato.py`, contendo o seguinte código:

```
1  # -*- coding: latin-1 -*-
2
3  from Tkinter import *
4  import tkMessageBox
5
6  from dao.contato_dao import ContatoDAO
7  from model.contato import Contato
8
9  class FrameContato(Frame):
10     def __init__(self, master=None):
11         Frame.__init__(self, master)
12         self.grid()
13         self._posicao = 0
14         self._dao = ContatoDAO()
15         self._contato = Contato()
16         self._contatos = self.lista_contatos()
17
18         self.__createWidgets()
19         self.primeiro_contato()
20
21     def __createWidgets (self):
22         self.master.title("PyAgenda")
23         self.master.resizable(width=False, height=False)
24
25         self._nome = StringVar()
26         self._telefone = StringVar()
27         self._celular = StringVar()
28         self._email = StringVar()
29
30         # Nome
31         self.lblNome = Label(self, text='Nome', width=10)
32         self.lblNome.grid(row=0, column=0)
33         self.txtNome = Entry(self, textvariable=self._nome)
34         self.txtNome.grid(row=0, column=1, sticky=W + E + N + S,
columnspan=6)
35
36         # Telefone
37         self.lblTelefone = Label(self, text='Telefone')
38         self.lblTelefone.grid(row=1, column=0)
39         self.txtTelefone = Entry(self, textvariable=self._telefone)
40         self.txtTelefone.grid(row=1, column=1, sticky=W + E + N +
S, columnspan=6)
41
42         # Celular
43         self.lblCelular = Label(self, text='Celular')
```

```
44          self.lblCelular.grid(row=2, column=0)
45          self.txtCelular = Entry(self, textvariable=self._celular)
46          self.txtCelular.grid(row=2, column=1, sticky=W + E + N + S,
columnspan=6)
47
48          # E-mail
49          self.lblEmail = Label(self, text='E-mail')
50          self.lblEmail.grid(row=3, column=0)
51          self.txtEmail = Entry(self, textvariable=self._email)
52          self.txtEmail.grid(row=3, column=1, sticky=W + E + N + S,
columnspan=6)
53
54          # Botões
55          self.btnNovo = Button(self, text='Novo')
56          self.btnNovo.grid(row=4, column=0, sticky=W + E + N + S,
padx=2)
57          self.btnNovo["command"] = self.novo_contato
58
59          self.btnSalvar = Button(self, text='Salvar')
60          self.btnSalvar["command"] = self.salva
61          self.btnSalvar.grid(row=4, column=1, sticky=W + E + N + S,
padx=2)
62
63          self.btnExcluir = Button(self, text='Excluir')
64          self.btnExcluir["command"] = self.exclui
65          self.btnExcluir.grid(row=4, column=2, sticky=W + E + N + S,
padx=2)
66
67          self.btnPrimeiro = Button(self, text='|<')
68          self.btnPrimeiro["command"] = self.primeiro_contato
69          self.btnPrimeiro.grid(row=4, column=3)
70
71          self.btnAnterior = Button(self, text='<')
72          self.btnAnterior["command"] = self.retorna_contato
73          self.btnAnterior.grid(row=4, column=4)
74
75          self.btnProximo = Button(self, text='>')
76          self.btnProximo["command"] = self.avanca_contato
77          self.btnProximo.grid(row=4, column=5)
78
79          self.btnUltimo = Button(self, text='>|')
80          self.btnUltimo["command"] = self.ultimo_contato
81          self.btnUltimo.grid(row=4, column=6)
82
83      def lista_contatos(self):
84          return self._dao.lista()
85
86      def novo_contato(self):
87          self._nome.set("")
88          self._telefone.set("")
89          self._celular.set("")
90          self._email.set("")
91          self._contato.limpa()
92          self.txtNome.focus()
93
94      def salva(self):
95          try:
96              # Pega dados do formulário
97              self._contato.nome = self.txtNome.get()
98              self._contato.telefone = self.txtTelefone.get()
99              self._contato.celular = self.txtCelular.get()
```

```
100             self._contato.email = self.txtEmail.get()
101
102             if self._contato.codigo == -1:
103                 self._dao.insere(self._contato)
104                 mensagem = "Contato inserido com sucesso!"
105             else:
106                 self._dao.atualiza(self._contato)
107                 mensagem = "Contato atualizado com sucesso!"
108
109             tkMessageBox.showinfo(title=mensagem, message=mensagem)
110             self._contatos = self.lista_contatos()
111             self.primeiro_contato()
112
113         except ValueError, mensagem:
114             tkMessageBox.showinfo(title="Erro ao inserir contato",
message=mensagem)
115
116     def exclui(self):
117         try:
118             resposta = tkMessageBox.askyesno(title="Confirma
exclusão?", message="Tem certeza que deseja excluir contato?")
119
120             if resposta:
121                 self._dao.exclui(self._contato)
122                 tkMessageBox.showinfo(title="Contato excluído com
sucesso!", message="Contato excluído com sucesso!")
123                 self._contatos = self.lista_contatos()
124                 self.primeiro_contato()
125         except ValueError, mensagem:
126             tkMessageBox.showinfo(title="Erro ao inserir contato",
message=mensagem)
127
128     def atualiza_campos(self):
129         if len(self._contatos):
130             self._contato.codigo =
self._contatos[self._posicao].codigo
131             self._contato.nome = self._contatos[self._posicao].nome
132             self._contato.telefone =
self._contatos[self._posicao].telefone
133             self._contato.celular =
self._contatos[self._posicao].celular
134             self._contato.email =
self._contatos[self._posicao].email
135
136             self._nome.set(self._contato.nome)
137             self._telefone.set(self._contato.telefone)
138             self._celular.set(self._contato.celular)
139             self._email.set(self._contato.email)
140
141     # Botões de navegação
142     def primeiro_contato(self):
143         self._posicao = 0
144         self.atualiza_campos()
145
146     def avanca_contato(self):
147         try:
148             if self._posicao != len(self._contatos) - 1:
149                 self._ posicao += 1
150                 self.atualiza_campos()
151         except IndexError:
152             pass
```

```
153
154     def retorna_contato(self):
155         try:
156             if self._posicao != 0:
157                 self._posicao -= 1
158                 self.atualiza_campos()
159         except IndexError:
160             self._posicao += 1
161
162     def ultimo_contato(self):
163         try:
164             self._posicao = len(self._contatos) - 1
165             self.atualiza_campos()
166         except IndexError:
167             pass
```

Nas primeiras linhas importamos as classes `Contato`, `ContatoDAO`, `Tkinter` e `tkMessageBox`. O módulo `Tkinter` contém os objetos necessários para a criação da *interface* gráfica, como o formulário, os botões e caixas de entrada, e o `tkMessageBox`, as funções de diálogo que utilizaremos para interagir com o usuário.

A seguir, definimos a classe `FrameContato`, que herda da classe `Frame`, presente no módulo `Tkinter`. No seu método `__init__` chamamos o construtor da classe base, e em seguida chamamos o método `grid` herdado de `Frame`, que torna a aplicação visível na tela.

Na linha 16 listamos todos os contatos do banco de dados através do método `lista_contatos`, definido na linha 83, que simplesmente retorna os registros retornados pelo método `lista` do objeto `_dao`. O método `__createWidgets` instancia os *widgets* (controles gráficos), os posiciona no *frame* e configura algumas propriedades, como na linha 22, que altera o título do frame do padrão ("tk") para "PyAgenda", e na 23 que torna o *frame* não-redimensionável.

Da linha 25 a 28 criamos objetos do tipo `StringVar` que utilizaremos para passar os valores para as caixas de texto. Veremos isso nas linhas seguintes.

Nas próximas linhas instanciamos objetos das classes `Label` – os rótulos das caixas de texto, `Entry` – as caixas de texto, e `Button`, os botões que utilizaremos para criar, salvar, excluir e navegar entre os contatos. Tomemos como exemplo as seguintes linhas:

```
31         self.lblNome = Label(self, text='Nome', width=10)
32         self.lblNome.grid(row=0, column=0)
```

A primeira linha instancia um objeto `Label`, passando como texto a string "Nome" e tamanho 10. A segunda posiciona o rótulo em uma "tabela", na linha 0 e coluna 0.

```
33         self.txtNome = Entry(self, textvariable=self._nome)
```

Na linha 33 instanciamos um objeto `Entry`, passando o objeto `StringVar` que criamos na linha 25 para controlar seu valor.

```
55          self.btnNovo = Button(self, text='Novo')
56          self.btnNovo.grid(row=4, column=0, sticky=W + E + N + S,
padx=2)
57          self.btnNovo["command"] = self.novo_contato
```

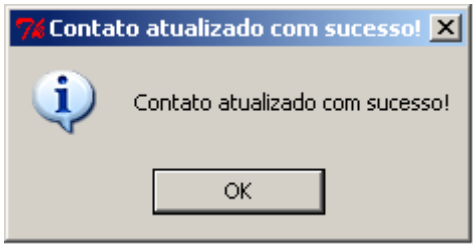
Da linha 55 e 57 instanciamos um botão, passando o valor “Novo” para seu texto, posicionamos o objeto recém criado na tabela, linha 4, coluna 0, tornando o controle redimensionável a N (Norte), S (Sul), E (Leste) e W (Oeste) e tornando sua margem igual a 2. Na próxima linha passamos o método `novo_contato` para a propriedade `command` do botão, assim, quando o usuário clicar no botão, ela será chamada. Note que não inserimos os parênteses após o nome da função, e fazemos isso porque estamos referenciando o objeto função, não o valor que ele retorna.

Nossa janela deve se parecer com isso:

Certo, agora temos os rótulos, caixas de texto e botões, precisamos adicionar as funcionalidades.

O método `novo_contato`, que é chamado quando o botão *Novo* é clicado, chama o método `set` dos objetos `StringVar`, passando uma string vazia. Quando esse método é chamado, as caixas de texto são limpas e a caixa de texto `txtNome` ganha foco pelo método `focus` do objeto `Entry`.

O método `salva`, que é chamado quando o botão *Salva* é clicado, “popula” o objeto `_contato` instanciado no método `__init__` com os valores digitados pelo usuário nas caixas de texto. Os valores são recebidos pelo método `get` dos objetos `Entry`. Em seguida, o método `salva` do objeto `_dao` é executado e a mensagem é configurada com base no código do contato. A função `showinfo` do módulo `tkMessageBox` é chamada, passando a mensagem configurada anteriormente no título e mensagem a serem exibidos, algo como:



Em seguida, atualiza o objeto `_contatos`, do tipo `list`, que armazena os contatos armazenados no banco de dados e o primeiro contato da lista é selecionado. O método `primeiro_contato` atribui 0 (zero) para o objeto `_posicao` e chama o método `atualiza_campos`, que popula o objeto `_contato` com o os dados da lista `_contatos`, no índice especificado pelo objeto `_posicao`. O método `atualiza_campos` também altera os textos dos objetos `StringVar` para as propriedades do objeto `_contato`, assim, as caixas de texto tem seus valores de acordo com o contato corrente.

O método `avanca_contato` verifica se não é o último contato que está selecionado e incrementa em 1 o objeto `_posicao`. Em seguida, atualiza o objeto `_contato` e os campos de texto do *frame*. Os métodos seguintes são muito parecidos com esse, portanto, não há necessidade de explicá-los.

Agora que temos as camadas de apresentação, acesso a dados e modelo, vamos criar o *starter* do aplicativo – o módulo responsável exibir o *frame*. No diretório raiz do aplicativo crie um módulo chamado `main.py` e insira o seguinte código:

```
1  # -*- coding: latin-1 -*-
2
3  from view.frame_contato import FrameContato
4
5  if __name__ == "__main__":
6      frame = FrameContato()
7      frame.mainloop()
```

A comparação `if __name__ == "__main__"` faz com que o conteúdo dentro do bloco `if` só seja executado caso o módulo seja executado por alguém, não quando importado por outro módulo. Isso porque quando um módulo é importado, seu atributo `__name__` é igual ao nome do módulo sem sua extensão, e quando executado, `__name__` é igual a `__main__`.

Em seguida, instanciamos `FrameContato` e armazenamos o objeto na variável `frame`. O método `mainloop` chamado na linha 7 faz com que o *frame* aguarde por eventos do *mouse* e teclado.

Nosso aplicativo está pronto para uso, mas ainda podemos facilitar sua distribuição para usuários de Windows através do py2exe, uma extensão do Python Distutils que nos

permite converter *scripts* em programas executáveis, sem mesmo necessitar do Python instalado no sistema.

Para podermos utilizar o *py2exe*, primeiro precisamos baixá-lo e instalá-lo no nosso sistema. Para isso, vá em http://sourceforge.net/projects/py2exe/files/ e baixe a última versão do aplicativo para seu sistema, no momento, a 0.6.9. É muito importante que você baixe a versão correta do *py2exe* para seu sistema, atentando para a arquitetura do sistema operacional (32bit ou 64bit) e a versão do Python instalado (2.6, 2.7, etc.). Caso você instale a versão incorreta do aplicativo para seu sistema podem ocorrer erros no processo de conversão dos *scripts* Python para executável.

A instalação é bem simples e rápida, no início do processo será solicitado que você selecione o diretório onde o Python foi instalado, faça como o solicitado e o *py2exe* será instalado sem problemas.

Uma vez tendo o *py2exe* instalado no sistema, devemos criar um arquivo de instalação, geralmente com o nome `setup.py`, no diretório raiz do nosso aplicativo. Dentro desse arquivo, insira o seguinte código:

```
1  # -*- coding: latin-1 -*-
2
3  from distutils.core import setup
4  import py2exe
5
6  setup(windows=['main.py'],
7        data_files = (("contatos.sqlite"),),
8  )
```

Os dois primeiros `import`s são necessários para o funcionamento do *py2exe* e *Distutils*. Em seguida chamamos a função `setup`, passando uma série de argumentos. Na linha 6, passamos o nome do módulo que dá início ao aplicativo, `main.py` e no argumento `data_files` especificamos que o nosso arquivo de banco de dados, `contatos.sqlite`, deverá ser copiado para a raiz do aplicativo.

Agora só precisamos chamar o *script* `setup.py` passando o argumento *install* na linha de comando. Através do *prompt* de comandos, vá até o diretório raiz do aplicativo, onde se localiza o módulo `setup.py` e dispare o comando:

```
python setup.py py2exe
```

Se você não recebeu nenhuma mensagem de erro durante o processo, nosso aplicativo foi convertido para executável com sucesso. Dentro do diretório raiz deve constar dois novos diretórios, `dist` – que contém os arquivos necessários para a distribuição do aplicativo, bem

como seu executável – e `build`, utilizado pelo *py2exe* como um espaço de trabalho enquanto o aplicativo está sendo enpacotado.

Entre no diretório `dist` e execute o arquivo `main.exe` – se tudo ocorreu bem, nosso aplicativo deverá ser exibido, e dessa vez sem o *prompt* de comando do Windows, presente quando executamos o módulo `main.py`.

Atenção

A exceção `UnicodeEncodeError` é acionada quando o *script* se encontra em um diretório cuja uma das partes do caminho possua um caractere não-*ascii*, como em `C:\Users\Usuário`.

```
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe1' in
position 99: ordinal not in range(128)
```

Programação Funcional

A Programação Funcional é um estilo de programação que enfatiza a avaliação de expressões ao invés da execução de comandos. O estilo funcional se contrasta com os paradigmas Imperativos por evitar alterações de variáveis e execução de comandos, conceitos no qual os paradigmas Imperativos se baseiam.

O conceito de *função* do Paradigma Funcional e dos paradigmas Imperativos se difere. As funções dos paradigmas Imperativos podem conter efeitos-colaterais, o que significa que eles podem utilizar e alterar variáveis globais (de fora da função). Por dependerem de variáveis externas, as funções dos paradigmas Imperativos não possuem transparência referencial, ou seja, dados dois valores de entrada, uma expressão *pode* resultar valores diferentes em momentos diferentes.

Funções do Paradigma Funcional, chamadas também de expressões, são puras - elas recebem valores de entrada e retornam um valor processado – funções puras não podem referenciar variáveis externas, o resultado delas depende somente dos seus valores de entrada, por isso, sempre retornam o mesmo valor dados os mesmos valores de entrada.

A eliminação dos efeitos-colaterais pode tornar muito mais fácil de entender e prever o comportamento de um programa e é uma das principais motivações do desenvolvimento do Paradigma Funcional. Muitos autores afirmam que as características da Programação Funcional tornam o desenvolvimento mais rápido e menos suscetível a erros, além de mais fácil de *debuggar* e testar, já que os programas são ‘quebrados’ em pequenas partes utilizando funções.

Apesar de todas as qualidades do Paradigma Funcional, é difícil evitar todos os efeitos colaterais. Linguagens de programação puramente funcionais - como Haskell, por exemplo - não possuem declarações de atribuição como a=3 ou c=a+b. Imprimir um texto na tela com a função `print` ou escrever um arquivo em disco, por exemplo, são efeitos colaterais e são tarefas comuns no desenvolvimento de programas.

Programas escritos em Python no estilo Funcional geralmente não vão ao extremo de evitar todas as operações de I/O ou atribuições, invés disso, eles fornecem uma interface que se parece funcional, mas que internamente utiliza recursos não-funcionais.

Linguagens Funcionais ou que suportam o Paradigma Funcional possuem algumas dessas características:

- Funções são objetos de primeira-classe, em outras palavras, qualquer coisa que pode ser feita com dados pode ser também feito com funções, como por exemplo, receber uma função como argumento, e retornar uma função;
- A recursividade é utilizada como a estrutura de controle primária. Em algumas linguagens, nenhuma outra estrutura de repetição existe;
- Há um foco no processamento de listas, que são frequentemente utilizadas com recursão e sub-listas como um substituto de *loops*;
- Há uma preocupação em *o que* será computado, ao invés de *como será* computado;

Python possui muitas das características acima. Os elementos básicos da Programação Funcional em Python são as funções embutidas `map`, `reduce`, e `filter`, bem como o operador `lambda`, utilizado para criar as funções *lambda*, que veremos adiante. A principal utilidade das funções serem objetos de primeira-classe é passá-las como argumentos para funções como `map`, `reduce` e `filter` que recebem uma função como primeiro argumento.

Funções anônimas ou *lambda*

As funções anônimas são aquelas que, como o seu nome já diz, não possuem nome e podem ser criadas e executadas a qualquer momento da execução do programa sem a necessidade de serem definidas. São funções que apenas retornam o valor de uma expressão sem precisar da cláusula `return`.

As funções *lambda* são puras, e mais próximas de funções matemáticas do que das funções normais, que podem referenciar variáveis globais, chamar outras funções e comandos. Ao contrário dessas, as funções *lambda* só podem referenciar seus parâmetros de entrada. Funções *lambda* são definidas com o operador `lambda` na forma:

```
lambda param1, param2: expressao
```

Exemplo:

```
1  inc = lambda x: x + 1
2  print inc(10)              # Imprime 11
```

Outro exemplo:

```
1  ampl = lambda x, y, z: (x ** 2 + y ** 2 + z ** 2) ** 0.5
2  print ampl(1, 4, 7)         # Imprime 8.12403840464
3  print ampl(5, 2, 4)         # Imprime 6.7082039325
```

A função `ampl` declarada acima recebe três parâmetros – `x`, `y` e `z` – e retorna o valor do cálculo da amplitude de um vetor 3D - `(x ** 2 + y ** 2 + z ** 2) ** 0.5`. As funções *lambda* consomem menos recursos computacionais, portanto, tendem a ser mais rápidas que as funções convencionais, vistas nos primeiros capítulos da apostila.

Mapeamento

O mapeamento consiste em aplicar uma função a todos os elementos de uma sequencia de dados, como uma lista, gerando outra lista do mesmo tamanho da lista inicial, contendo os resultados do mapeamento. O mapeamento em Python é realizado com a função embutida `map` que pode substituir diretamente o uso de um laço de repetição `for`. Exemplo:

Usando um laço for

```
1  lista = [1, 2, 3, 4, 5]
2  lista_2 = []
3  for i in lista:
4      lista_2.append(i ** 2)
5  print lista_2              # Imprime [1, 4, 9, 16, 25]
```

Usando a função map

```
1  quad = lambda x: x ** 2
2  lista = [1, 2, 3, 4, 5]
3  lista_2 = map(quad, lista)
4  print lista_2              # Imprime [1, 4, 9, 16, 25]
```

O primeiro argumento da função `map` é a função *lambda* `quad`, que eleva à segunda potência o número passado por argumento. O segundo argumento é a lista na qual os elementos serão aplicados à função `quad`. A lista gerada pela função `map` é armazenada como `lista_2`, e então exibida na linha 4. Como as funções em Python são objetos de primeira-classe, podemos fazer isso:

```
1  fx = lambda f: f()
2
3  f1 = lambda: 'f1() executada'
4  f2 = lambda: 'f2() executada'
5  f3 = lambda: 'f3() executada'
```

```
6  f4 = lambda: 'f4() executada'
7
8  fn = [f1, f2, f3, f4]
9
10 print map(fx, fn) # Imprime ['f1() executada', 'f2() executada',
'f3() executada', 'f4() executada']
```

A função `fx` recebe como argumento uma função e a executa. Da linha 3 à linha 6 definimos 4 funções *lambda* que retornam uma string, e na linha 8 as adicionamos na lista `fn`. A função `map` aplica a função `fx` aos elementos da lista `fn` – em outras palavras, executamos as funções `f1` a `f4`, gerando uma lista que é impressa com a função `print` na linha 10.

Filtragem

A filtragem consiste em aplicar uma função a todos os elementos de uma sequencia para gerar uma sequencia filtrada. Se a função aplicada retornar `True` para o elemento ele fará parte da nova sequencia. Ao contrário da função `map`, a lista resultante não terá necessariamente o mesmo tamanho da lista original. Exemplo:

```
1  e_par = lambda x: (x % 2) == 0
2  lista = [1, 2, 3, 4, 5, 6]
3  lista_pares = filter(e_par, lista)
4
5  print lista_pares    # Imprime [2, 4, 6]
```

A função *lambda* `e_par` retorna `True` quando o número passado como argumento é par e `False` quando é impar. Quando o elemento da lista aplicado à função `e_par` retorna `True`, é armazenado em `lista_pares`, que é impressa na linha 4. O mesmo código pode ser escrito ao estilo imperativo da seguinte forma:

```
1  lista = [1, 2, 3, 4, 5, 6]
2  lista_pares = []
3
4  for num in lista:
5     if (num % 2) == 0:
6        lista_pares.append(num)
7
8  print lista_pares    # Imprime [2, 4, 6]
```

Redução

Redução, conhecida também no mundo da Programação Funcional como *Fold*, consiste em aplicar uma função que recebe dois parâmetros nos dois primeiros elementos de uma sequencia, aplicar novamente essa função usando como parâmetros o resultado do primeiro par e o terceiro elemento, e assim sucessivamente. Ao contrário das funções `map` e `filter` vistas anteriormente, o resultado da função `reduce` é apenas um elemento e não uma lista. O exemplo a seguir ajudará a entender melhor o uso da função `reduce`:

```
1  def psoma(x, y):
2     print '%d + %d = %d' % (x, y, x + y)
3     return x + y
4
5  lista = [1, 2, 3, 4, 5]
6
7  print reduce(psoma, lista)
```

O código anterior imprime:

```
1 + 2 = 3
3 + 3 = 6
6 + 4 = 10
10 + 5 = 15
15
```

O exemplo anterior utilizando uma função que imprime o resultado parcial de forma legível serviu para demonstrar como a função `reduce` funciona internamente – ela possui um acumulador, que guarda o resultado da operação anterior utilizando-o na próxima operação. Os dois exemplos de códigos a seguir são equivalentes, o primeiro ao estilo imperativo - com um laço de repetição `for`, e o segundo utilizando uma função *lambda* e a função `reduce`:

Imperativo

```
1  tupla = (1, 2, 3, 4, 5)
2  soma = 0
3  for num in tupla:
4     soma += num
5
6  print soma                          # Imprime 15
```

Funcional

```
1  tupla = (1, 2, 3, 4, 5)
2  print reduce(lambda x, y: x + y, tupla)     # Imprime 15
```

Iteradores

Um iterador é um objeto que representa um fluxo de dados, retornando um elemento do fluxo de cada vez. Iteradores devem suportar um método chamado `next` que não recebe argumentos e sempre retorna o próximo elemento do fluxo de dados, acionando a exceção `StopIteration` quando o iterador chegou ao fim. A função *built-in* `iter` é utilizada para criar um iterador - esta função recebe um objeto como argumento e tenta retornar um iterador que irá retornar os conteúdos dos objetos, ou elementos. Há diversos tipos de dados que suportam iteração, os mais comuns sendo sequencias, como listas, tuplas e dicionários. O exemplo abaixo demonstra a criação de um iterador a partir de uma lista no interpretador interativo:

```
1  >>> lista = [1, 2, 3]
```

```
>>> iterador = iter(lista)
>>> iterador.next()
1
>>> iterador.next()
2
>>> iterador.next()
3
>>> iterador.next()
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

Como podemos ver no exemplo acima, a cada vez que chamamos o método `next` do objeto iterador, ele retorna um elemento do objeto sequencialmente, até que a exceção `StopIteration` é acionada quando não há mais elementos no objeto. Note que com iteradores só é possível avançar sobre seus elementos, não há uma maneira de retornar um elemento anterior, fazer uma cópia ou *resetar* um iterador. Python espera objetos iteráveis em diversos contextos, o mais importante sendo a estrutura de repetição `for`. Por exemplo, na expressão `"for x in Y"`, o `Y` deve ser um iterador ou algum objeto onde a função `iter` possa criar um iterador. As duas seqüências a seguir são idênticas:

```
for i in iter(obj):
    print i

for i in obj:
    print i
```

Iteradores podem ser materializados em listas e tuplas utilizando, respectivamente, os construtores `list` e `tuple`. Exemplos:

```
>>> L = [1,2,3]
>>> iterador = iter(L)
>>> tupla = tuple(iterador)
>>> print tupla                 # Imprime (1, 2, 3)
```

Generator expressions e *Lists comprehensions*

As duas operações mais comuns da saída de um iterador são: 1) Realizar alguma operação em todos os elementos e; 2) Selecionar um subconjunto de elementos que satisfazem uma condição. Por exemplo, dada uma lista de strings, você pode querer retirar todos os espaços finais de cada linha ou extrair todas as strings que contenham uma dada substring.

List comprehensions e *generator expressions*, conhecidas também como "*listcomps*" e "*genexps*", são notações concisas para tais operações e foram pegas emprestadas da linguagem de programação funcional Haskell. *Generator expressions* são envolvidas entre

parênteses, enquanto *list comprehensions* são envolvidas por colchetes ([ ]). *Generator expressions* são definidos na forma:

```
( expressao for expr1 in sequencia1
            if condicao1
            for expr2 in sequencia2
            if condicao2
            for expr3 in sequencia3
            if condicao3
            for exprN in sequenciaN
            if condicaoN )
```

Como a única diferença na definição de *list comprehensions* e *generator expressions* são os parênteses e colchetes, para criar uma *listcomps*, basta substituir os parênteses por colchetes:

```
[ expressao for expr1 in sequencia1
            if condicao1
            for expr2 in sequencia2
            if condicao2
            for expr3 in sequencia3
            if condicao3
            for exprN in sequenciaN
            if condicaoN ]
```

Os elementos gerados, tanto de *listcomps* quanto de *genexps*, serão valores sucessivos de `expressao`. A clausula `if` de ambos é opcional e se estiver presente, `expressao` só será avaliada e adicionada ao resultado caso a condição seja avaliada como `True`.

Para exemplificar, com as *listcomps* e *genexps* você pode retirar todos os espaços em branco a partir de um fluxo de strings da seguinte forma:

```
1  lista_strings = ['  Linha 1\n', 'Linha 2  \n', 'Linha 3     ']
2
3  # Com generator expressions
4  stripped_iter = (linha.strip() for linha in lista_strings)
5
6  # Com list comprehensions
7  stripped_list = [linha.strip() for linha in lista_strings]
8
9  print stripped_list    # Imprime ['Linha 1', 'Linha 2', 'Linha 3']
10
11 for linha in stripped_iter:
12    print linha
```

Adicionando um `if` você também pode selecionar certos elementos da lista. Exemplo:

```
stripped_list = [linha.strip() for linha in lista_strings
             if linha != ""]
```

Como se pode perceber, uma *list comprehension* retorna uma lista, que imprimimos na linha 9 do primeiro exemplo. *Generator expressions* retornam um iterador que calcula os

valores conforme o necessário, não necessitando de se materializar todos os valores de uma só vez. Utilizar *list comprehensions* quando se trabalha com iteradores que retornam um fluxo infinito ou muito grande de dados não é muito adequado, sendo preferível a utilização de *generator expressions* nesses casos.

Devido aos *generator expressions* calcularem os valores conforme a necessidade, ele tende a utilizar menos recursos que *list comprehensions*, principalmente memória.

Generators

Generators são uma classe especial de funções que simplificam a tarefa de escrever iteradores. Como vimos anteriormente, funções normais calculam um valor, o retornam e encerram. *Generators* funcionam de maneira similar, porém retornam um iterador que retorna um fluxo de valores e assim lembram o estado de processamento entre as chamadas, permanecendo em memória e retornando o próximo elemento quando chamado.

Generators geralmente são evocados através de um laço `for`. A sintaxe de *generators* é semelhante a das funções normais, porém ao invés da cláusula `return`, há a cláusula `yield`. A cada nova iteração, `yield` retorna o próximo valor calculado. Exemplo de uma função *geradora*:

```
1  def generate_ints(N):
2      for i in range(N):
3          yield i
```

A função geradora `generate_ints` gera uma seqüência de números até o número passado por argumento. Um *generator* retorna um iterador, então para extrair os valores de `generate_ints` podemos fazer:

```
1  for n in generate_ints(3):
2      print n
```

Ou ainda:

```
1  gen = generate_ints(3)
2
3  print gen.next()          # Imprime 1
4  print gen.next()          # Imprime 2
5  print gen.next()          # Imprime 3
```

Generators e *generator expressions* utilizam o conceito de *lazy evaluation*, conhecida também como avaliação não-estrita ou preguiçosa, utilizada como padrão na maioria das linguagens Funcionais. *Lazy evaluation* é a técnica de atrasar um cálculo até que seu resultado seja necessário, e um de seus maiores benefícios é o aumento de desempenho e economia de memória. O conceito de *lazy evaluation* vai de encontro ao de *eager evaluation*, conhecida

também como avaliação estrita ou ansiosa, que é utilizada como padrão na maioria das linguagens imperativas. Na *eager evaluation* uma expressão é avaliada assim que ela é vinculada a uma variável. Os exemplos a seguir demonstram a diferença entre os dois tipos de avaliação:

Usando um generator (Programação Funcional)

```
1  def gen_fibonacci(limite):
2      x, y = 1, 1
3      i = 0
4      while i < limite:
5          yield x
6          x, y = y, x+y
7          i += 1
8
9  for num in gen_fibonacci (10):
10     print num
```

Usando uma função normal (Programação Imperativa)

```
1  def fibonacci(limite):
2      x, y, i = 1, 0, 0
3      num = 1
4      result = []
5      while i < limite:
6          i += 1
7          num = x + y
8          x = y
9          y = num
10         result.append(num)
11     return result
12
13 for num in fibonacci(10):
14     print num
```

No primeiro exemplo, ao estilo funcional, a cada iteração do laço `for` a função geradora `gen_fibonacci` é executada e retorna, com `yield`, um elemento da seqüência *fibonacci*, que então é impresso na linha 10.

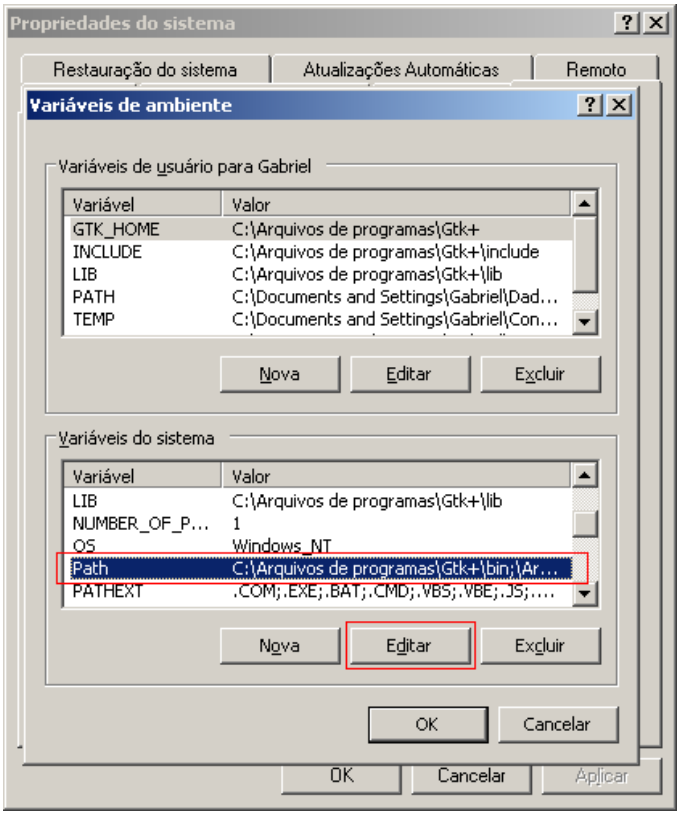
Já no segundo exemplo, ao estilo Imperativo, a função `fibonacci` é executada por inteiro no laço `for` e retorna uma lista contendo todos os elementos já calculados da sequência *fibonacci*. Essa lista é então transformada de maneira transparente em um iterador, e tem seus elementos impressos na linha 14.

Apêndice

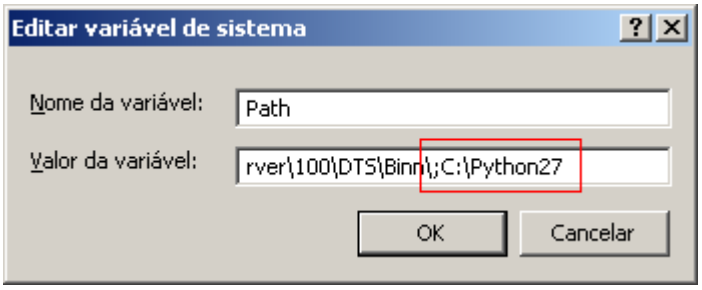
A – Configurando o PATH do Windows

Adicionar o caminho do interpretador do Python no PATH do Windows torna mais fácil tanto executar o interpretador interativo, quanto executar programas escritos em Python na linha de comando.

Primeiro, pressione simultaneamente as teclas WINDOWS e PAUSE/BREAK para abrir a janela *Propriedades do sistema*. Na aba Avançado, clique em Variáveis de ambiente, selecione a variável *Path* em *Variáveis de sistema* e clique em *Editar*.



Em seguida, adicione o caminho onde o Python foi instalado – geralmente `C:\PythonXX`, onde XX corresponde sua versão (sem ponto) – no final do texto já existente na caixa *Valor da variável*, separando os por um ponto-e-vírgula, como mostra a imagem a seguir:



Pressione *OK* e feche as janelas anteriores. As alterações feitas só terão efeito após o computador ser reiniciado. Para verificar se as alterações funcionaram, abra o *Prompt* de comando (CMD no Windows 2000/XP/Vista/7) e digite `python`.