



# **Como pensar como um cientista da Computação usando Python**

*Allen Downey*

*Jeffrey Elkner*

*Chris Meyers*

*Tradução: Equipe do projeto <http://pensarpython.incubadora.fapesp.br>*

*Edição: Cárlisson Galdino*

Copyright (c) 2002 Allen Downey, Jeffrey Elkner, and Chris Meyers.  
Edited by Shannon Turlington and Lisa Cutler. Cover design by Rebecca Gimenez.

Printing history:

April 2002: First edition.

Green Tea Press

1 Grove St.

P.O. Box 812901

Wellesley, MA 02482

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being "Foreword," "Preface," and "Contributor List," with no Front-Cover Texts, and with no BackCover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License."

The GNU Free Documentation License is available from [www.gnu.org](http://www.gnu.org) or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

### Como pensar como um cientista da Computação usando Python

Versão atual: **1.0**

Data de Publicação: **16 de janeiro de 2008**

Classificação: **Tutorial**

Autor: **Jeffrey Elkner** (traduzido pelo projeto Pensar Python - <http://pensarpython.incubadora.fapesp.br>)

Publicação: **Cyaneus.net**

## Índice

Foreword.....	5	4.8 A instrução return.....	28
Apresentação.....	6	4.9 Recursividade.....	29
Preface.....	7	4.10 Diagramas de pilha para funções recursivas.....	29
How and why I came to use Python.....	7	4.11 Recursividade infinita.....	30
Finding a textb o ok.....	7	4.12 Entrada pelo teclado.....	30
Intro ducing programming with Python.....	7	4.13 Glossário.....	30
Building a community.....	8	Capítulo 5: Funções frutíferas.....	31
Prefácio.....	9	5.1 Valores de retorno.....	31
Como e porque eu vim a usar Python.....	9	5.2 Desenvolvimento de programas.....	31
Encontrando um livro texto.....	9	5.3 Composição.....	32
Introduzindo programação com Python.....	10	5.4 Funções booleanas.....	32
Construindo uma comunidade.....	10	5.5 Mais recursividade.....	33
Contributor List.....	11	5.6 Voto de confiança (Leap of faith).....	33
Versão Brasileira.....	13	5.7 Mais um exemplo.....	34
Capítulo 1: O caminho do programa.....	14	5.8 Checagem de tipos.....	34
1.1 A linguagem de programação Python.....	14	5.9 Glossário.....	34
1.2 O que é um programa?.....	15	Capítulo 6: Iteração.....	36
1.3 O que é depuração (debugging)?.....	15	6.1 Reatribuições.....	36
1.3.1 Erros de sintaxe.....	15	6.2 O comando while.....	36
1.3.2 Erros em tempo de execução (runtime errors)	15	6.3 Tabelas.....	37
.....	15	6.4 Tabelas de duas dimensões (ou bi-dimensionais). .	38
1.3.3 Erros de semântica.....	15	6.5 Encapsulamento e generalização.....	38
1.3.4 Depuração experimental (Debugging).....	16	6.6 Mais encapsulamento.....	38
1.4 Linguagens naturais e linguagens formais.....	16	6.7 Variáveis locais.....	38
1.5 O primeiro programa.....	17	6.8 Mais generalização.....	39
1.6 Glossário.....	17	6.9 Funções.....	39
Capítulo 2: Variáveis, expressões e comandos.....	18	6.10 Glossário.....	40
2.1 Valores e tipos.....	18	Capítulo 7: Strings.....	41
2.2 Variáveis.....	18	7.1 Um tipo de dado composto.....	41
2.3 Nomes de variáveis e palavras reservadas.....	18	7.2 Comprimento.....	41
2.4 Comandos.....	19	7.3 Travessia e o loop for.....	41
2.5 Avaliando expressões.....	19	7.4 Fatias de strings.....	42
2.6 Operadores e operandos.....	19	7.5 Comparação de strings.....	42
2.7 Ordem dos operadores.....	20	7.6 Strings são imutáveis.....	42
2.8 Operações com strings.....	20	7.7 Uma função find (encontrar).....	42
2.9 Composição.....	20	7.8 Iterando e contando.....	43
2.11 Glossário.....	20	7.9 O módulo string.....	43
Capítulo 3: Funções.....	22	7.10 Classificação de caracteres.....	43
3.1 Chamadas de funções.....	22	7.11 Glossário.....	43
3.2 Conversão entre tipos.....	22	7.11 Glossário2.....	44
3.3 Coerção entre tipos.....	22	Capítulo 8: Listas.....	45
3.4 Funções matemáticas.....	22	8.1 Valores da lista.....	45
3.5 Composição.....	23	8.2 Acessado elementos.....	45
3.6 Adicionando novas funções.....	23	8.3 Comprimento da lista.....	45
3.7 Definições e uso.....	24	8.4 Membros de uma lista.....	46
3.8 Fluxo de execução.....	24	8.5 Listas e laços for.....	46
3.9 Parâmetros e argumentos.....	24	8.6 Operações em listas.....	46
3.10 Variáveis e parâmetros são locais.....	25	8.7 Fatiamento de listas.....	46
3.11 Diagramas da pilha.....	25	8.8 Listas são mutáveis.....	46
3.12 Funções com resultados.....	25	8.9 Remoção em lista.....	47
3.13 Glossário.....	26	8.10 Ojetos e valores.....	47
Capítulo 4: Condicionais e recursividade.....	27	8.11 Apelidos.....	47
4.1 O operador módulo.....	27	8.12 Clonando listas.....	48
4.2 Expressões booleanas.....	27	8.13 Lista como parâmetro.....	48
4.3 Operadores lógicos.....	27	8.14 Lista aninhadas.....	48
4.4 Execução condicional.....	27	8.15 Matrizes.....	48
4.5 Execução alternativa.....	28	8.16 Strings e listas.....	49
4.6 Condicionais encadeados.....	28	8.17 Glossário.....	49
4.7 Condicionais aninhados.....	28	Capítulo 9: Tuplas.....	50

9.1 Mutabilidade e tuplas.....	50	16.1 Herança.....	75
9.2 Atribuições de tupla.....	50	16.2 Uma mão de cartas.....	75
9.3 Tuplas como valores de retorno.....	50	16.3 Dando as cartas.....	75
9.4 Números aleatórios.....	50	16.4 Exibindo a mão.....	76
9.5 Lista de números aleatórios.....	51	16.5 A classe JogoDeCartas.....	76
9.6 Contando.....	51	16.6 Classe MaoDeMico.....	76
9.7 Vários intervalos.....	51	16.7 Classe Mico.....	77
9.8 Uma solução em um só passo.....	52	16.8 Glossário.....	78
9.9 Glossário.....	52	Capítulo 17: Listas encadeadas.....	79
Capítulo 10: Dicionários.....	54	17.1 Referências Embutidas.....	79
10.1 Operações dos Dicionários.....	54	17.2 A classe No (Node).....	79
10.2 Métodos dos Dicionários.....	54	17.3 Listas como Coleções.....	79
10.3 Aliasing (XXX) e Copiar.....	55	17.4 Listas e Recorrência.....	80
10.4 Matrizes Esparsas.....	55	17.5 Listas Infinitas.....	80
10.5 Hint.....	55	17.6 O Teorema da Ambigüidade Fundamental.....	80
10.6 Inteiros Longos.....	56	17.7 Modificando Listas.....	81
10.7 Contando Letras.....	56	17.8 Envoltórios e Ajudadores.....	81
10.8 Glossário.....	56	17.9 A Classe ListaLigada.....	81
Capítulo 11: Arquivos e exceções.....	58	17.10 Invariantes.....	82
Arquivos e exceções.....	58	17.11 Glossário.....	82
11.1 Arquivos texto.....	58	Capítulo 18: Pilhas.....	83
11.2 Gravando variáveis.....	59	18.1 Tipos abstratos de dados.....	83
11.3 Diretórios.....	60	18.2 O TAD Pilha.....	83
11.4 Pickling.....	60	18.3 Implementando pilhas com listas de Python.....	83
11.5 Exceções.....	60	18.4 Empilhando e desempilhando.....	83
11.6 Glossário.....	61	18.5 Usando uma pilha para avaliar expressões pós- fixas.....	84
Capítulo 12: Classes e objetos.....	62	18.6 Análise sintática.....	84
12.1 Tipos compostos definidos pelo usuário.....	62	18.7 Avaliando em pós-fixado.....	84
12.2 Atributos.....	62	18.8 Clientes de fornecedores.....	84
12.3 Instâncias como parâmetros.....	63	18.9 Glossário.....	85
12.4 O significado de "mesmo".....	63	Capítulo 19: Filas.....	86
12.5 Retângulos.....	63	19.1 Um TDA Fila.....	86
12.6 Instancias como valores retornados.....	64	19.2 Fila encadeada.....	86
12.7 Objetos são mutáveis.....	64	19.3 Características de performance.....	86
12.8 Copiando.....	64	19.4 Fila encadeada aprimorada.....	86
12.9 Glossário.....	65	19.5 Fila por prioridade.....	87
Capítulo 13: Classes e funções.....	66	19.6 A classe Golfer.....	88
13.1 Horário.....	66	19.7 Glossário.....	88
13.2 Funções Puras.....	66	Capítulo 20: Árvores.....	89
13.3 Modificadores.....	66	20.1 Construindo árvores.....	89
13.4 O que é melhor ?.....	67	20.2 Percorrendo árvores.....	89
13.5 Desenvolvimento Prototipado versus Desenvolvimento Planejado.....	67	20.3 Árvores de expressões.....	89
13.6 Generalização.....	67	20.4 Percorso de árvores.....	90
13.7 Algoritmos.....	68	20.5 Construindo uma árvore de expressão.....	90
13.8 Glossário.....	68	20.6 Manipulando erros.....	92
Capítulo 14: Classes e métodos.....	69	20.7 A árvore dos animais.....	92
14.1 Características da orientação a objetos.....	69	20.8 Glossário.....	93
14.2 <code>exibeHora</code> ( <code>printTime</code> ).....	69	GNU Free Documentation License.....	94
14.3 Um outro exemplo.....	70	0. PREAMBLE .....	94
14.10 Glossário.....	70	1. APPLICABILITY AND DEFINITIONS .....	94
Capítulo 15: Conjuntos de objetos.....	71	2. VERBATIM COPYING .....	95
15.1 Composição.....	71	3. COPYING IN QUANTITY .....	95
15.2 Objetos Carta.....	71	4. MODIFICATIONS .....	95
15.3 Atributos de classe e o método <code>__str__</code> .....	71	5. COMBINING DOCUMENTS .....	96
15.4 Comparando cartas.....	72	6. COLLECTIONS OF DOCUMENTS .....	96
15.5 Baralhos.....	72	7. AGGREGATION WITH INDEPENDENT WORKS . 96	
15.6 Imprimindo o baralho.....	72	8. TRANSLATION .....	96
15.7 Embaralhando.....	73	9. TERMINATION .....	96
15.8 Removendo e distribuindo cartas.....	73	10. FUTURE REVISIONS OF THIS LICENSE.....	96
15.9 Glossário.....	74	How to use this License for your documents.....	96
Capítulo 16: Herança.....	75		

## Foreword

By David Beazley

As an educator, researcher, and book author, I am delighted to see the completion of this book. Python is a fun and extremely easy-to-use programming language that has steadily gained in popularity over the last few years. Developed over ten years ago by Guido van Rossum, Python's simple syntax and overall feel is largely derived from ABC, a teaching language that was developed in the 1980's. However, Python was also created to solve real problems and it borrows a wide variety of features from programming languages such as C++, Java, Modula-3, and Scheme. Because of this, one of Python's most remarkable features is its broad appeal to professional software developers, scientists, researchers, artists, and educators.

Despite Python's appeal to many different communities, you may still wonder "why Python?" or "why teach programming with Python?" Answering these questions is no simple task—especially when popular opinion is on the side of more masochistic alternatives such as C++ and Java. However, I think the most direct answer is that programming in Python is simply a lot of fun and more productive.

When I teach computer science courses, I want to cover important concepts in addition to making the material interesting and engaging to students. Unfortunately, there is a tendency for introductory programming courses to focus far too much attention on mathematical abstraction and for students to be come frustrated with annoying problems related to low-level details of syntax, compilation, and the enforcement of seemingly arcane rules. Although such abstraction and formalism is important to professional software engineers and students who plan to continue their study of computer science, taking such an approach in an introductory course mostly succeeds in making computer science boring. When I teach a course, I don't want to have a room of uninspired students. I would much rather see them trying to solve interesting problems by exploring different ideas, taking unconventional approaches, breaking the rules, and learning from their mistakes. In doing so, I don't want to waste half of the semester trying to sort out obscure syntax problems, unintelligible compiler error messages, or the several hundred ways that a program might generate a general protection fault.

One of the reasons why I like Python is that it provides a really nice balance between the practical and the conceptual. Since Python is interpreted, beginners can pick up the language and start doing neat things almost immediately without getting lost in the problems of compilation and linking. Furthermore, Python comes with a large library of modules that can be used to do all sorts of tasks ranging from web-programming to graphics. Having such a practical focus is a great way to engage students and it allows them to complete significant projects. However, Python can also serve as an excellent foundation for introducing important computer science concepts. Since Python fully supports procedures and classes, students can be gradually introduced to topics such as procedural abstraction, data structures, and object-oriented programming—all of which are applicable to later courses on Java or C++. Python even borrows a number of features from functional programming languages and can be used to introduce concepts that would be covered in more detail in courses on Scheme and Lisp.

In reading Jeffrey's preface, I am struck by his comments that Python allowed him to see a "higher level of success and a lower level of frustration" and that he was able to "move faster with better results." Although these comments refer to his introductory course, I sometimes use Python for these exact same reasons in advanced graduate level computer science courses at the University of Chicago. In these courses, I am constantly faced with the daunting task of covering a lot of difficult course material in a blistering nine week quarter. Although it is certainly possible for me to inflict a lot of pain and suffering by using a language like C++, I have often found this approach to be counterproductive—especially when the course is about a topic unrelated to just "programming." I find that using Python allows me to better focus on the actual topic at hand while allowing students to complete substantial class projects.

Although Python is still a young and evolving language, I believe that it has a bright future in education. This book is an important step in that direction.

David Beazley

University of Chicago

Author of the Python Essential Reference

### Apresentação

*Tradução do capítulo anterior, que foi mantido por ter sido marcado como seção invariante pelo autor original.*

Como educador, pesquisador e autor de livros, regozija-me ver completo este trabalho. Python é uma linguagem de programação divertida e extremamente fácil de usar que tem ganho forte popularidade nestes últimos poucos anos. Desenvolvida dez anos atrás por Guido van Rossum, a sintaxe simples do Python e seu sentido geral são grandemente derivados do ABC, uma linguagem didática que foi desenvolvida nos anos 80. Entretanto, Python também foi criado para solucionar problemas reais e tomou emprestado uma grande quantidade de características de linguagens de programação como C++, Java, Modula-3 e Scheme. Por causa disso, uma das mais notáveis características do Python é o grande apelo que tem junto a desenvolvedores profissionais de software, cientistas, pesquisadores, artistas e educadores.

A despeito deste apelo do Python junto às mais variadas comunidades, você pode ainda estar pensando "por que Python?" ou "por que ensinar programação com Python?". Responder à estas perguntas não é uma tarefa fácil, especialmente se a opinião pública está do lado de alternativas mais masoquistas como C++ e Java. Entretanto, eu acho que a resposta mais direta é que programar com Python é um bocado divertido e mais produtivo.

Quando ministro cursos de ciências da computação, o que desejo é cobrir conceitos importantes além de tornar a matéria interessante e os alunos participativos. Infelizmente, existe uma tendência entre os cursos introdutórios de programação a focar atenção demais em abstrações matemáticas, e de frustração entre os alunos com problemas enfadonhos e inoportunos relacionados a detalhes de sintaxe em baixo nível, compilação e a imposição de regras que aparentemente só um expert pode compreender. Embora alguma abstração e formalismo sejam importantes para engenheiros profissionais de software e estudantes que planejam continuar seus estudos em ciências da computação, escolher tal abordagem em um curso introdutório faz da ciência da computação algo entediante. Quando ministro um curso, não desejo uma sala cheia de alunos sem inspiração. Em vez disso, preferiria muito mais vê-los tentando solucionar problemas interessantes explorando idéias diferentes, trilhando caminhos não convencionais, quebrando regras, e aprendendo a partir de seus erros. Fazendo assim, não pretendo desperdiçar metade de um semestre tentando explicar problemas obscuros de sintaxe, mensagens ininteligíveis de compiladores ou as várias centenas de maneiras pelas quais um programa pode gerar uma falha geral de proteção.

Uma das razões pelas quais eu gosto de Python

é que ele oferece um equilíbrio realmente bom entre o lado prático e o lado conceitual. Sendo Python interpretado, os iniciantes podem pegar a linguagem e começar a fazer coisas legais quase imediatamente sem se perderem em problemas de compilação e ligação. Além disso, Python vem com uma grande biblioteca de módulos que podem ser utilizados para fazer todo tipo de tarefa, desde a programação para a web até gráficos. Com tal enfoque prático temos uma bela maneira de alcançar o engajamento dos alunos e permitir que eles finalizem projetos significativos. Entretanto, Python também pode servir de excelente embasamento para a introdução de conceitos importantes em ciência da computação. Já que Python suporta plenamente procedimentos (procedures) e classes, os alunos podem ser gradualmente introduzidos a tópicos como abstração procedural, estruturas de dados, e programação orientada a objetos, todos aplicáveis em cursos posteriores de Java ou C++. Python ainda toma emprestado certas características de linguagens de programação funcionais e pode ser usado para introduzir conceitos cujos detalhes poderiam ser aprofundados em cursos de Scheme e Lisp.

Lendo o prefácio de Jeffrey, fiquei impressionado com seu comentário de que Python o fez ver um "maior nível de sucesso e um menor nível de frustração" o que lhe permitiu "progredir mais depressa com resultados melhores". Embora estes comentários refiram-se aos seus cursos introdutórios, eu às vezes uso Python exatamente pelas mesmas razões em cursos avançados de pós-graduação em ciência da computação na Universidade de Chicago. Nestes cursos, enfrento constantemente a assustadora tarefa de cobrir muitos tópicos difíceis em um rapidíssimo trimestre de nove semanas. Embora me seja possível inflingir um bocado de dor e sofrimento pelo uso de uma linguagem como C++, tenho percebido muitas vezes que este enfoque é contraproducente, especialmente quando o curso é sobre um tópico não relacionado apenas com "programar". Acho que usar Python me permite um melhor foco no tópico em questão, enquanto permite que os alunos completem projetos substanciais em classe.

Embora Python seja ainda uma linguagem jovem e em evolução, acredito que tem um futuro brilhante em educação. Este livro é um passo importante nessa direção.

David Beazley

Universidade de Chicago

Autor de Python Essencial Reference



## Preface

By Jeff Elkner

This book owes its existence to the collaboration made possible by the Internet and the free software movement. Its three authors—a college professor, a high school teacher, and a professional programmer—have yet to meet face to face, but we have been able to work closely together and have been aided by many wonderful folks who have donated their time and energy to helping make this book better.

We think this book is a testament to the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

### How and why I came to use Python

In 1999, the College Board's Advanced Placement (AP) Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum at Yorktown High School in Arlington, Virginia, where I teach. Up to this point, Pascal was the language of instruction in both our first-year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first-year course for the 1997-98 school year so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first-year class, I went looking for an alternative to C++.

I needed a language that would run on the machines in our Linux lab as well as on the Windows and Macintosh platforms most students have at home. I wanted it to be free and available electronically, so that students could use it at home regardless of their income. I wanted a language that was used by professional programmers, and one that had an active developer community around it. It had to support both procedural and object-oriented programming. And most importantly, it had to be easy to learn and teach. When I investigated the choices with these goals in mind, Python stood out as the best candidate for the job.

I asked one of Yorktown's talented students, Matt Ahrens, to give Python a try. In two months he not only learned the language but wrote an application called pyTicket that enabled our staff to report technology problems via the Web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment, combined with Matt's positive assessment of Python, suggested that Python was the solution I was looking for.

### Finding a textbook

Having decided to use Python in both of my introductory

computer science classes the following year, the most pressing problem was the lack of an available textbook.

Free content came to the rescue. Earlier in the year, Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational content. Allen had already written a first-year computer science textbook, *How to Think Like a Computer Scientist*. When I read this book, I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming rather than the features of a particular language. Reading it immediately made me a better teacher.

*How to Think Like a Computer Scientist* was not just an excellent book, but it had been released under a GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version of the book into the new language. While I would not have been able to write a textbook on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational content.

Working on this book for the last two years has been rewarding for both my students and me, and my students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for mistakes in the book by giving them a bonus point each time they made a suggestion that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object-oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book sat in an unfinished state for the better part of a year until the free software community once again provided the needed means for its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene, Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our Website at <http://www.ibiblio.org/obp> called Python for Fun and was working with some of my most advanced students as a master teacher, guiding them beyond where I could take them.

### Introducing programming with Python

The process of translating and using *How to Think Like a Computer Scientist* for the past two years has confirmed Python's suitability for teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text illustrates this point. It is the traditional "hello, world" program, which in the

C++ version of the book looks like this:

```
#include <iostream.h>
void main()
{
    cout << "Hello, world." << endl;
}
```

in the Python version it becomes:

```
print "Hello, World!"
```

Even though this is a trivial example, the advantages of Python stand out. Yorktown's Computer Science I course has no prerequisites, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The C++ version has always forced me to choose between two unsatisfying options: either to explain the `#include`, `void main()`, `{`, and `}` statements and risk confusing or intimidating some of the students right at the start, or to tell them, "Just don't worry about all of that stuff now; we will talk about it later," and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to write their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing the explanatory text of the program in each version of the book further illustrates what this means to the beginning student. There are thirteen paragraphs of explanation of "Hello, world!" in the C++ version; in the Python version, there are only two. More importantly, the missing eleven paragraphs do not deal with the "big ideas" in computer programming but with the minutia of C++ syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high-level language like Python allows a teacher to postpone talking about low-level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put "first things first" pedagogically. One of the best examples of this is the way in which Python handles variables. In C++ a variable is a name for a place that holds a thing. Variables have to be declared with types at least in part because the size of the place to which they refer needs to be predetermined. Thus, the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both computer science and algebra). Bytes and addresses do not help the matter. In Python a variable is a name that refers to a thing. This is a far more intuitive concept for beginning students and is much closer to the meaning of "variable" that

they learned in their math courses. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the keyword `def`, so I simply tell my students, "When you define a function, begin with `def`, followed by the name of the function that you are defining; when you call a function, simply call (type) out its name." Parameters go with definitions; arguments go with calls. There are no return types, parameter types, or reference and value parameters to get in the way, so I am now able to teach functions in less than half the time that it previously took me, with better comprehension.

Using Python has improved the effectiveness of our computer science program for all students. I see a higher general level of success and a lower level of frustration than I experienced during the two years I taught C++. I move faster with better results. More students leave the course with the ability to create meaningful programs and with the positive attitude toward the experience of programming that this engenders.

---

### Building a community

---

I have received email from all over the globe from people using this book to learn or to teach programming. A user community has begun to emerge, and many people have been contributing to the project by sending in materials for the companion Website at <http://www.thinkpython.com>.

With the publication of the book in print form, I expect the growth in the user community to continue and accelerate. The emergence of this user community and the possibility it suggests for similar collaboration among educators have been the most exciting parts of working on this project for me. By working together, we can increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you. Please write to the authors at [feedback@thinkpython.com](mailto:feedback@thinkpython.com).

Jeffrey Elkner  
Yorktown High School  
Arlington, Virginia



### Prefácio

*Tradução do capítulo anterior, que foi mantido por ter sido marcado como seção invariante pelo autor original.*

Este livro deve sua existência à colaboração tornada possível pela Internet e pelo movimento do software livre. Seus três autores ? um professor universitário, um secundário e um programador profissional ? ainda não se encontraram pessoalmente, mas temos podido trabalhar bem de perto e temos sido ajudados por muitos colegas maravilhosos que têm dedicado seu tempo e energia a ajudar a fazer deste um livro cada vez melhor.

Achamos que este livro é um testemunho dos benefícios e possibilidades futuras deste tipo de colaboração, cujo modelo tem sido colocado por Richard Stallman e pela Free Software Foundation.

#### Como e porque eu vim a usar Python

Em 1999, o Exame de Colocação Avançada em Ciência da Computação da Comissão de Faculdades (College Board's Advanced Placement (AP) Computer Science) foi aplicado em C++ pela primeira vez. Como em muitas escolas secundárias através do país, a decisão de mudar linguagens teve um impacto direto no currículo de ciência da computação na Yorktown High School em Arlington, Virginia, onde leciono. Até então, Pascal era a linguagem didática para nossos cursos de primeiro ano e avançado. Mantendo a prática corrente de dar aos estudantes dois anos de exposição à mesma linguagem, tomamos a decisão de mudar para C++ no curso de primeiro ano para o ano letivo de 1997-98 de modo que estaríamos em sincronismo com a mudança da Comissão de Faculdades (College Board's) em relação ao curso avançado para o ano seguinte.

Dois anos depois, eu estava convencido que C++ foi uma escolha infeliz para introduzir os alunos em ciência da computação. Ao mesmo tempo em que é certamente uma linguagem de programação muito poderosa, também é uma linguagem extremamente difícil de aprender e de ensinar. Eu me encontrava constantemente lutando com a sintaxe difícil do C++ e as múltiplas maneiras de fazer a mesma coisa, e estava, como resultado, perdendo muitos alunos desnecessariamente. Convencido de que deveria existir uma linguagem melhor para a nossa classe de primeiro ano, fui procurar por uma alternativa ao C++.

Eu precisava de uma linguagem que pudesse rodar nas máquinas em nosso laboratório Linux bem como nas plataformas Windows e Macintosh que a maioria dos alunos tinha em casa. Eu precisava que ela fosse gratuita e disponível eletronicamente, assim os alunos poderiam utilizá-la em casa independentemente de suas rendas. Eu queria uma linguagem que fosse utilizada por programadores profissionais, e que tivesse uma comunidade de desenvolvimento ativa em torno dela. Ela teria que suportar ambas, programação procedural e orientada a objetos. E, mais importante, deveria ser fácil de aprender e de ensinar. Quando considerei as alternativas tendo em mente aquelas metas, Python sobressaiu-se como a melhor candidata para a tarefa.

Pedi para um dos talentosos estudantes de Yorktown, Matt Ahrens, que experimentasse Python. Em dois meses ele não só aprendeu a linguagem como também escreveu uma aplicação chamada pyTicket que possibilitou à nossa equipe reportar problemas de tecnologia pela Web. Eu sabia que Matt não poderia ter finalizado uma aplicação daquele porte em período tão curto em C++, esta realização, combinada com a avaliação positiva do Python dada por Matt, sugeriam que Python era a solução que eu estava procurando.

#### Encontrando um livro texto

Tendo decidido usar Python em ambas as minhas classes introdutórias de ciência da computação do ano seguinte, o problema mais urgente era a falta de um livro texto disponível.

O conteúdo livre veio em socorro. Anteriormente naquele ano, Richard Stallman tinha me apresentado a Allen Downey. Ambos havíamos escrito a Richard expressando interesse em desenvolver conteúdo educacional livre. Allen já tinha escrito um livro texto para o primeiro ano de ciência da computação, *How to Think Like a Computer Scientist*. Quando li este livro, soube imediatamente que queria utilizá-lo nas minhas aulas. Era o texto mais claro e proveitoso em ciência da computação que eu tinha visto. Ele enfatizava o processo de reflexão envolvido em programação em vez de características de uma linguagem em particular. Lê-lo fez de mim imediatamente um melhor professor.

O *How to Think Like a Computer Scientist* era não só um excelente livro, como também fora lançado sob uma licença pública GNU, o que significava que ele poderia ser usado livremente e modificado para atender as necessidades de seu usuário. Uma vez que eu havia decidido usar Python, me ocorreu que eu poderia traduzir a versão original do livro de Allen do Java para a nova linguagem. Apesar de não estar capacitado para escrever eu mesmo um livro texto, tendo o livro de Allen a partir do qual trabalhar tornou possível para mim fazê-lo, ao mesmo tempo demonstrando que o modelo de desenvolvimento cooperativo tão bem utilizado em software poderia também funcionar para conteúdo educacional.

Trabalhar neste livro pelos últimos dois anos tem sido recompensador para mim e meus alunos, e eles tiveram um grande papel neste processo. A partir do momento em que eu podia fazer mudanças instantâneas assim que alguém encontrasse um erro ortográfico ou um trecho difícil, eu os encorajei a procurar por erros no livro, dando a eles pontos de bonificação cada vez que eles fizessem uma sugestão que resultasse em uma mudança no texto. Isto teve o duplo benefício de encorajá-los a ler o texto mais cuidadosamente e de ter o texto totalmente revisado por seus críticos mais importantes: alunos utilizando-o para aprender ciência da computação.

Para a segunda metade do livro, sobre programação orientada a objetos, eu sabia que seria preciso alguém com uma maior experiência do que a minha em programação real para fazê-lo corretamente. O livro esteve em estado inacabado por quase um ano até que a comunidade de software livre providenciasse mais uma vez os meios necessários para sua conclusão.

Eu recebi um e-mail de Chris Meyers mostrando interesse no livro. Chris é um programador profissional que começou a dar um curso de programação no ano anterior usando Python no Lane Community College em Eugene, Oregon. A perspectiva de dar aquele curso ligou Chris ao livro, e ele começou a ajudar o trabalho imediatamente. Pelo final do ano letivo ele tinha criado um projeto colaborativo em nosso Website em <http://www.ibiblio.org/obp> chamado Python for Fun e estava trabalhando com alguns dos meus alunos mais avançados como um guru, guiando-os além de onde eu poderia levá-los.

### Introduzindo programação com Python

O processo de traduzir e utilizar How to Think Like a Computer Scientist pelos últimos dois anos tem confirmado a conveniência de Python no ensino de alunos iniciantes. Python simplifica tremendamente os programas exemplo e torna idéias importantes de programação mais fáceis de ensinar.

O primeiro exemplo do texto ilustra este ponto. É o tradicional programa "Alô mundo?", do qual na versão C++ do livro se parece com isto:

```
#include <iostream.h>

void main()
{
    cout << "Alô, mundo." << endl;
}
```

Na versão Python, ele se transforma em:

```
print "Alô, Mundo!"
```

Mesmo sendo um exemplo trivial, as vantagens do Python saltam aos olhos. O curso de Ciência da Computação I que ministro em Yorktown não tem pré-requisitos, assim, muitos dos alunos que veem esse exemplo estão olhando para o seu primeiro programa. Alguns deles estão indubitavelmente nervosos, por já terem ouvido falar que programação de computadores é difícil de aprender. A versão C++ tem sempre me forçado a escolher entre duas opções insatisfatórias: ou explicar os comandos #include, void main(), {, e } e arriscar confundir ou intimidar alguns dos alunos logo assim que iniciam, ou dizer a eles "Não se preocupem com todas estas coisas agora; falaremos sobre elas mais tarde", e correr o mesmo risco. O objetivo educacional neste ponto do curso é introduzir os alunos à idéia de comando em programação e vê-los escrever seu primeiro programa, deste modo introduzindo-os ao ambiente de programação. O programa em Python tem exatamente o que é necessário para conseguir isto, e nada mais.

Comparar o texto explicativo do programa em cada versão do livro ilustra ainda mais o que significa para o aluno iniciante. Existem treze parágrafos de explicação do "Alô, mundo!" na versão C++; na versão Python existem apenas dois. Mais importante, os onze parágrafos perdidos não se ocupam das "idéias chave" da programação de computadores, mas com a minúcia da sintaxe C++. Vejo a mesma coisa acontecendo através de todo o livro. Parágrafos inteiros simplesmente desaparecem da versão do texto para Python porque a sintaxe muito mais clara do Python os torna desnecessários.

Utilizar uma linguagem de tão alto nível como Python, permite ao professor deixar para falar mais tarde sobre os níveis mais baixos, próximos à máquina, quando os alunos já terão a experiência necessária para ver com mais sentido os detalhes. Desta maneira podemos "por em primeiro lugar as primeiras coisas?", pedagogicamente. Um dos melhores exemplos disto é a maneira com que Python lida com variáveis. Em C++ uma variável é um nome para um lugar que guarda uma coisa. Variáveis têm de ser declaradas com seu tipo pelo menos em parte por que o tamanho do lugar a que se referem precisa ser predeterminado. Assim, a idéia de variável fica amarrada ao hardware da máquina. O conceito

poderoso e fundamental de variável já é bastante difícil para o aluno iniciante (em ambas, ciência da computação e álgebra). Bytes e endereços não ajudam neste caso. Em Python uma variável é um nome que se refere a uma coisa. Este é um conceito muito mais intuitivo para alunos iniciantes e está muito mais próximo do significado de "variável" que eles aprenderam em seus cursos de matemática. Eu tive muito menos dificuldade em ensinar variáveis este ano do que tive no passado, e gastei menos tempo ajudando aos alunos com problemas no uso delas.

Um outro exemplo de como Python ajuda no ensino e aprendizagem de programação é em sua sintaxe para função. Meus alunos têm sempre tido grande dificuldade na compreensão de funções. O problema principal gira em torno da diferença entre a definição de uma função e a chamada de uma função, e a distinção relacionada entre um parâmetro e um argumento. Python vem em auxílio com uma sintaxe não apenas curta quanto bela. As definições de função começam com def, então eu simplesmente digo aos meus alunos "Quando você define uma função, comece com def, seguido do nome da função que você está definindo; quando você chama uma função, simplesmente chame-a digitando o nome dela?". Parâmetros ficam nas definições; argumentos vão com as chamadas. Não existem tipos de retorno, tipos de parâmetro ou passagem de parâmetros por valor ou por referência no meio do caminho, permitindo-me ensinar funções em menos da metade do tempo que isto me tomava anteriormente, com uma melhor compreensão.

A utilização do Python tem melhorado a efetividade de nosso programa em ciência da computação para todos os estudantes. Eu vejo um nível geral de sucesso muito mais alto e um nível mais baixo de frustração do que experimentei durante os dois anos em que ensinei C++. Eu avanço mais rápido com melhores resultados. Mais alunos deixam o curso com a habilidade de criar programas significativos e com uma atitude positiva em relação a experiência de programação que isso traz.

### Construindo uma comunidade

Tenho recebido e-mails de todo o planeta de pessoas utilizando este livro para aprender ou ensinar programação. Uma comunidade de usuários tem começado a emergir e muitas pessoas têm contribuído com o projeto enviando seus materiais para o Website cooperativo em:

<http://www.thinkpython.com>

Com a publicação do livro em formato impresso, minha expectativa quanto ao crescimento da comunidade de usuários é que ela seja contínua e acelerada. O surgimento desta comunidade de usuários e a possibilidade que sugere de colaboração semelhante entre educadores tem sido para mim a parte mais excitante do trabalho neste projeto. Trabalhando juntos, podemos aumentar a qualidade do material disponível para o nosso uso e poupar tempo valioso. Eu convido você a se juntar a nossa comunidade e espero ouvir algo de você. Por favor, escreva para os autores em [feedback@thinkpython.com](mailto:feedback@thinkpython.com).

Jeffrey Elkner

Yorktown High School

Arlington, Virginia

## Contributor List

To paraphrase the philosophy of the Free Software Foundation, this book is free like free speech, but not necessarily free like free pizza. It came about because of a collaboration that would not have been possible without the GNU Free Documentation License. So we thank the Free Software Foundation for developing this license and, of course, making it available to us.

We also thank the more than 100 sharp-eyed and thoughtful readers who have sent us suggestions and corrections over the past few years. In the spirit of free software, we decided to express our gratitude in the form of a contributor list. Unfortunately, this list is not complete, but we are doing our best to keep it up to date.

If you have a chance to look through the list, you should realize that each person here has spared you and all subsequent readers from the confusion of a technical error or a less-than-transparent explanation, just by sending us a note.

Impossible as it may seem after so many corrections, there may still be errors in this book. If you should stumble across one, please check the online version of the book at <http://thinkpython.com>, which is the most up-to-date version. If the error has not been corrected, please take a minute to send us email at [feedback@thinkpython.com](mailto:feedback@thinkpython.com). If we make a change due to your suggestion, you will appear in the next version of the contributor list (unless you ask to be omitted). Thank you!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the `Makefile` so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word “unconsciously” in Chapter 1 needed to be changed to “subconsciously”.
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13.
- John Ouzts corrected the definition of “return value” in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the `printTime` function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen’s Perlscript that generates HTML from LaTeX.
- Craig T. Snyder is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between `gleich` and `selbe`.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezny created the

illustrations in Chapter 1 and improved many of the other illustrations.

- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and is helping us prepare to update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that “a error” is an error.
- Abel David and Alexis Dinno reminded us that the plural of “matrix” is “matrices”, not “matrixes”. This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of “argument” and “parameter”.
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.

## Versão Brasileira

A versão traduzida para Português Brasileiro foi feita pela equipe do site <http://pensarpython.incubadora.fapesp.br>, abaixo relacionada:

- [Adrovane Kade \(adrovane\)](#)
- [Alex Augusto da Luz dos Santos \(nowayx\)](#)
- [Claudio Fernando Berrondo Soares \(cl-audio\)](#)
- [Daniel Rosa Franzini \(danielt3\)](#)
- [Douglas Soares de Andrade \(dsa\)](#)
- [Fabio Rizzo Matos \(fabrizmat\)](#)
- [Imre Simon \(imres\)](#)
- [Joao Paulo Liberato \(jpliberato\)](#)
- [João Paulo Gomes Vanzueta \(taken\)](#)
- [Julio Monteiro \(jmonteiro\)](#)
- [Luciano Ramalho \(luciano\)](#)
- [Marcus Pereira \(mymendes\)](#)
- [Mario O. de Menezes \(modemene\)](#)
- [Paulo J. S. Silva \(pjssilva\)](#)
- [Victor Rafael da Paixão Lopes \(reije\)](#)
- [marta mello \(martamello\)](#)
- [vitor.gurgel \(vitor\\_gurgel\)](#)

Esta diagramação foi feita por Cárliston Galdino <[bardo@castelodotempo.com](mailto:bardo@castelodotempo.com)>, com algumas modificações visando a facilitar a publicação em duas colunas. Alguns comentários de depuração feitos pela equipe de tradução foram omitidos.



## Capítulo 1: O caminho do programa

O objetivo deste livro é ensinar o leitor a pensar como um cientista da computação. Essa maneira de pensar combina algumas das melhores características da matemática, da engenharia e das ciências naturais. Como os matemáticos, os cientistas da computação usam linguagens formais para representar idéias (especificamente, computações). Como os engenheiros, eles projetam coisas, montando sistemas a partir de componentes e avaliando as vantagens e desvantagens de diferentes alternativas. Como os cientistas naturais, eles observam o comportamento de sistemas complexos, formulam hipóteses e testam previsões.

A habilidade mais importante de um cientista da computação é a **solução de problemas**. Solução de problemas é a habilidade de formular questões, pensar criativamente sobre soluções possíveis e expressar uma solução de forma clara e precisa. Ocorre que aprender a programar é uma excelente oportunidade de praticar a habilidade da solução de problemas. É por isso que este capítulo se chama "O caminho do programa".

Em certo nível, você estará aprendendo a programar, habilidade que é útil em si mesma. Em outro nível, você usará a programação como um meio para atingir um objetivo. A medida que você for avançando na leitura, esse objetivo ficará mais claro.

### 1.1 A linguagem de programação Python

Python é a linguagem de programação que você vai estudar neste livro. Python é um exemplo de **linguagem de programação de alto nível**: outras linguagens de alto nível de que você já pode ter ouvido falar são C, C++, Perl e Java.

Como você pode deduzir a partir da expressão "linguagem de alto nível", também existem as "linguagens de baixo nível", às vezes chamadas de "linguagens de máquina" ou "linguagem assembly" ("linguagens de montagem"). Dito de maneira simples, o computador só consegue executar programas escritos em linguagens de baixo nível. Deste modo, programas escritos em linguagens de alto nível precisam ser processados antes que possam rodar. Esse processamento extra toma algum tempo, o que é uma pequena desvantagem em relação às linguagens de alto nível.

Mas as vantagens são enormes. Primeiro, é muito mais fácil programar em uma linguagem de alto nível. É mais rápido escrever programas em uma linguagem de alto nível; eles são mais curtos e mais fáceis de ler e é mais provável que estejam corretos. Segundo, as linguagens de alto nível são **portáveis**, o que significa que podem rodar em diferentes tipos de computador, com pouca ou nenhuma modificação. Programas em baixo nível só podem rodar em um único tipo de computador e precisam ser re-escritos para rodar em outro tipo.

Devido a essas vantagens, quase todos os programas são escritos em linguagens de alto nível. As de baixo nível são utilizadas somente para umas poucas aplicações especializadas.

Dois tipos de programas processam linguagens de alto nível, traduzindo-as em linguagens de baixo nível: interpretadores e compiladores. O interpretador lê um programa escrito em linguagem de alto nível e o executa, ou seja, faz o que o programa diz. Ele processa o programa um pouco de cada vez, alternadamente: ora lendo algumas linhas, ora realizando computações.



O compilador lê o programa e o traduz completamente antes que o programa comece a rodar. Neste caso, o programa escrito em linguagem de alto nível é chamado de código fonte, e o programa traduzido é chamado de código objeto ou executável. Uma vez que um programa é compilado, você pode executá-lo repetidamente, sem que precise de nova tradução.



Python é considerada uma linguagem interpretada, porque os programas em Python são executados por um interpretador. Existem duas maneiras de usar o interpretador: no modo de linha de comando e no modo de script. No modo de linha de comando, você digita programas em Python e o interpretador mostra o resultado:

```

$ python
Python 2.4.3 (#2, Oct 6 2006, 07:49:22)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> print 1 + 1
2
  
```

A primeira linha deste exemplo é o comando que inicia o interpretador Python. As três linhas seguintes são mensagens do interpretador. A quarta linha começa com >>>, que é o sinal usado pelo interpretador para indicar que ele está pronto. No exemplo anterior, digitamos `print 1 + 1` e o interpretador respondeu 2.

Você também pode escrever um programa em um arquivo e usar o interpretador para executar o conteúdo desse arquivo. Um arquivo como este é chamado de **script**. Por exemplo, usamos um editor de texto para criar um arquivo chamado `leticia.py` com o seguinte conteúdo:

```
print 1 + 1
```

Por convenção, arquivos que contenham programas em Python têm nomes que terminam com `.py`.

Para executar o programa, temos de dizer ao interpretador o nome do script:

```
$ python leticia.py
2
```

Em outros ambientes de desenvolvimento, os detalhes da execução de programas podem ser diferentes. Além disso, a maioria dos programas são mais interessantes do que esse...

A maioria dos exemplos neste livro são executados a partir da linha de comando. Trabalhar com a linha de comando é conveniente no desenvolvimento e testagem de programas, porque você pode digitar os programas e executá-los imediatamente. Uma vez que você

tem um programa que funciona, deve guardá-lo em um script, de forma a poder executá-lo ou modificá-lo no futuro.

### 1.2 O que é um programa?

Um **programa** é uma seqüência de instruções que especificam como executar uma computação. A computação pode ser algo matemático, como solucionar um sistema de equações ou encontrar as raízes de um polinômio, mas também pode ser uma computação simbólica, como buscar e substituir uma palavra em um documento ou (estranhamente) compilar um programa.

Os detalhes são diferentes em diferentes linguagens, mas algumas instruções básicas aparecem em praticamente todas as linguagens:

<b>entrar</b>	Pegar dados do teclado, de um arquivo ou de algum outro dispositivo.
<b>sair</b>	Mostrar dados na tela ou enviar dados para um arquivo ou outro dispositivo.
<b>calcular</b>	Executar operações matemáticas básicas, como adição e multiplicação.
<b>executar condicionalmente</b>	Checar certas condições e executar a seqüência apropriada de instruções.
<b>repetir</b>	Executar alguma ação repetidamente, normalmente com alguma variação.

Acredite se quiser: isso é praticamente tudo. Todos os programas que você já usou, não importa quão complicados, são feitos de instruções mais ou menos parecidas com essas. Assim, poderíamos definir programação como o processo de dividir uma tarefa grande e complexa em sub-tarefas cada vez menores, até que as sub-tarefas sejam simples o suficiente para serem executadas com uma dessas instruções básicas.

Isso pode parecer um pouco vago, mas vamos voltar a esse tópico mais adiante, quando falarmos sobre **algoritmos**.

### 1.3 O que é depuração (debugging)?

Programar é um processo complicado e, como é feito por seres humanos, freqüentemente conduz a erros. Por mero capricho, erros em programas são chamados de **bugs** e o processo de encontrá-los e corrigi-los é chamado de **depuração** (debugging).

Três tipos de erro podem acontecer em um programa: erros de sintaxe, erros em tempo de execução (runtime errors) e erros de semântica. Distinguir os três tipos ajuda a localizá-los mais rápido:

#### 1.3.1 Erros de sintaxe

Python só executa um programa se ele estiver sintaticamente correto; caso contrário, o processo falha e retorna uma mensagem de erro. **Sintaxe** se refere à estrutura de um programa e às regras sobre esta estrutura. Por exemplo, em português, uma frase deve começar com uma letra maiúscula e terminar com um ponto.

esta frase contém um **erro de sintaxe**. Assim como esta

Para a maioria dos leitores, uns errinhos de sintaxe não chegam a ser um problema significativo e é por isso que conseguimos ler a poesia moderna de e. e. cummings sem cuspir mensagens de erro. Python não é tão indulgente. Se o seu programa tiver um único erro de sintaxe em algum

lugar, o interpretador Python vai exibir uma mensagem de erro e vai terminar - e o programa não vai rodar. Durante as primeiras semanas da sua carreira como programador, você provavelmente perderá um bocado de tempo procurando erros de sintaxe. Conforme for ganhando experiência, entretanto, cometerá menos erros e os localizará mais rápido.

#### 1.3.2 Erros em tempo de execução (runtime errors)

O segundo tipo de erro é o erro de runtime, ou erro em tempo de execução, assim chamado porque só aparece quando você roda o programa. Esses erros são também conhecidos como **exceções**, porque normalmente indicam que alguma coisa excepcional (e ruim) aconteceu.

Erros de runtime são raros nos programas simples que você vai ver nos primeiros capítulos - então, vai demorar um pouco até você se deparar com um erro desse tipo.

#### 1.3.3 Erros de semântica

O terceiro tipo de erro é o **erro de semântica** (mais comumente chamado erro de lógica). Mesmo que o seu programa tenha um erro de semântica, ele vai rodar com sucesso, no sentido de que o computador não vai gerar nenhuma mensagem de erro. Só que o programa não vai fazer a coisa certa, vai fazer alguma outra coisa. Especificamente, aquilo que você tiver dito para ele fazer.

O problema é que o programa que você escreveu não é aquele que você queria escrever. O significado do programa (sua semântica ou lógica) está errado. Identificar erros semânticos pode ser complicado, porque requer que você trabalhe de trás para frente, olhando a saída do programa e tentando imaginar o que ele está fazendo.

#### 1.3.4 Depuração experimental (Debugging)

Uma das habilidades mais importantes que você vai adquirir é a de depurar. Embora possa ser frustrante, depurar é uma das partes intelectualmente mais ricas, desafiadoras e interessantes da programação.

De certa maneira, a depuração é como um trabalho de detetive. Você se depara com pistas, e tem que deduzir os processos e eventos que levaram aos resultados que aparecem.

Depurar também é como uma ciência experimental. Uma vez que você tem uma idéia do que está errado, você modifica o seu programa e tenta de novo. Se a sua hipótese estava correta, então você consegue prever o resultado da modificação e fica um passo mais perto de um programa que funciona. Se a sua hipótese estava errada, você tem que tentar uma nova. Como Sherlock Holmes mostrou, "Quando você tiver eliminado o impossível, aquilo que restou, ainda que improvável, deve ser a verdade." (Arthur Conan Doyle, O signo dos quatro).

Para algumas pessoas, programação e depuração são a mesma coisa. Ou seja, programar é o processo de gradualmente depurar um programa, até que ele faça o que você quer. A idéia é começar com um programa que faça alguma coisa e ir fazendo pequenas modificações, depurando-as conforme avança, de modo que você tenha sempre um programa que funciona.

Por exemplo, o Linux é um sistema operacional que contém milhares de linhas de código, mas começou como um programa simples, que Linus Torvalds usou para explorar o chip Intel 80386. De acordo com Larry Greenfield, "Um dos primeiros projetos de Linus Torvalds foi um programa que deveria alternar entre imprimir AAAA e BBBB. Isso depois evoluiu até o Linux". (The Linux User's Guide Versão Beta 1)

Capítulos posteriores farão mais sugestões

sobre depuração e outras práticas de programação.

### 1.4 Linguagens naturais e linguagens formais

**Linguagens naturais** são as linguagens que as pessoas falam, como o português, o inglês e o espanhol. Elas não foram projetadas pelas pessoas (muito embora as pessoas tentem colocar alguma ordem nelas); elas evoluíram naturalmente.

**Linguagens formais** são linguagens que foram projetadas por pessoas, para aplicações específicas. Por exemplo, a notação que os matemáticos usam é uma linguagem formal, que é particularmente boa em denotar relações entre números e símbolos. Os químicos usam uma linguagem formal para representar a estrutura química das moléculas. E, mais importante:

Linguagens de programação são linguagens formais que foram desenvolvidas para expressar computações.

As linguagens formais tendem a ter regras estritas quanto à sintaxe. Por exemplo,  $3 + 3 = 6$  é uma expressão matemática sintaticamente correta, mas  $3=+6$  não é.  $H_2O$  é um nome químico sintaticamente correto, mas  $2Zz$  não é.

As regras de sintaxe são de dois tipos, um relacionado aos tokens, outro à estrutura. "Tokens" são os elementos básicos da linguagem, como as palavras, números, e elementos químicos. Um dos problemas com  $3=+6$  é que  $=$  não é um token válido em linguagem matemática (pelo menos até onde sabemos). Do mesmo modo,  $2Zz$  é inválida porque não existe nenhum elemento cuja abreviatura seja  $Zz$ .

O segundo tipo de erro de sintaxe está relacionado à estrutura de uma expressão -- quer dizer, ao modo como os tokens estão arrumados. A expressão  $3=+6$  é estruturalmente inválida, porque você não pode colocar um sinal de "mais" imediatamente após um sinal de "igual". Do mesmo modo, fórmulas moleculares devem ter índices subscritos colocados depois do nome do elemento, não antes.

Faça este exercício: crie o que pareça ser uma frase bem estruturada em português com "tokens" irreconhecíveis dentro dela. Depois escreva outra frase com todos os "tokens" válidos, mas com uma estrutura inválida.

Quando você lê uma frase em português ou uma expressão em uma linguagem formal, você tem de imaginar como é a estrutura da frase (embora, em uma linguagem natural, você faça isso inconscientemente). Este processo é chamado **parsing** (análise sintática).

Por exemplo, quando você ouve a frase, "Caiu a ficha", entende que "a ficha" é o sujeito e "caiu" é o verbo. Uma vez que você analisou a frase, consegue entender o seu significado, ou a semântica da frase. Assumindo que você saiba o que é uma ficha e o que significa cair, você entenderá o sentido geral dessa frase.

Muito embora as linguagens formais e as naturais tenham muitas características em comum -- tokens, estrutura, sintaxe e semântica -- existem muitas diferenças:

<b>ambigüidade</b>	As linguagens naturais estão cheias de ambigüidades, que as pessoas contornam usando pistas contextuais e outras informações. Já as linguagens formais são desenvolvidas para serem quase ou totalmente desprovidas de ambigüidade, o que significa que qualquer expressão tem precisamente só um sentido, independentemente do contexto.
<b>redundância</b>	Para compensar a ambigüidade e reduzir mal-entendidos, emprega-se muita

	redundância nas linguagens naturais, o que freqüentemente as torna prolixas. As linguagens formais são menos redundantes e mais concisas.
<b>literalidade</b>	As linguagens naturais estão cheias de expressões idiomáticas e metáforas. Se eu digo "Caiu a ficha", é possível que não exista ficha nenhuma, nem nada que tenha caído. Nas linguagens formais, não há sentido ambíguo.

Pessoas que crescem falando uma linguagem natural -- ou seja, todo mundo -- muitas vezes têm dificuldade de se acostumar com uma linguagem formal. De certa maneira, a diferença entre linguagens formais e naturais é como a diferença entre poesia e prosa, porém mais acentuada:

<b>poesia</b>	As palavras são usadas pela sua sonoridade, além de seus sentidos, e o poema como um todo cria um efeito ou uma reação emocional. A ambigüidade não é apenas freqüente, mas na maioria das vezes, proposital.
<b>prosa</b>	O sentido literal das palavras é mais importante, e a estrutura contribui mais para o significado. A prosa é mais fácil de analisar do que a poesia, mas ainda é muitas vezes ambígua.
<b>programas</b>	O significado de um programa de computador é exato e literal, e pode ser inteiramente entendido pela análise de seus tokens e de sua estrutura.

Aqui vão algumas sugestões para a leitura de programas (e de outras linguagens formais). Primeiro, lembre-se de que linguagens formais são muito mais densas do que linguagens naturais, por isso, é mais demorado lê-las. A estrutura, também, é muito importante, logo, geralmente não é uma boa idéia ler de cima para baixo, da esquerda para a direita. Em vez disso, aprenda a analisar o programa na sua cabeça, identificando os tokens e interpretando a estrutura. Finalmente, os detalhes são importantes. Pequenas coisas, como, erros ortográficos e má pontuação, com as quais você pode se safar nas linguagens naturais, podem fazer uma grande diferença em uma linguagem formal.

### 1.5 O primeiro programa

Tradicionalmente, o primeiro programa escrito em uma nova linguagem de programação é chamado de "Alô, Mundo!" porque tudo que ele faz é apresentar as palavras "Alô, Mundo!". Em Python, ele é assim:

```
print "Alô, Mundo!"
```

Isso é um exemplo de um comando print, que, na realidade, não "imprime" nada em papel. Ele apresenta o valor na tela. Neste caso, o resultado são as palavras:

```
Alô, Mundo!
```

As aspas no programa marcam o começo e o fim do valor; elas não aparecem no resultado final.

Algumas pessoas julgam a qualidade de uma linguagem de programação pela simplicidade do programa "Alô, Mundo!". Por esse padrão, Python se sai tão bem quanto possível.

### 1.6 Glossário

<b>solução de problemas</b>	O processo de formular um problema, encontrar uma solução e expressar esta
-----------------------------	--

## Como pensar como um cientista da Computação usando Python

<b>(problem solving)</b>	solução.
<b>linguagem de alto nível (high-level language)</b>	Uma linguagem de programação como Python: projetada para ser fácil para os seres humanos a utilizarem.
<b>linguagem de baixo nível (low-level language)</b>	Uma linguagem de programação que é concebida para ser fácil para um computador, tal como a linguagem de máquina ou a linguagem montagem (assembly language)
<b>portabilidade (portability)</b>	Propriedade que um programa tem, de rodar em mais de um tipo de computador.
<b>interpretar (interpret)</b>	Executar um programa escrito em uma linguagem de alto nível, traduzindo-o uma linha de cada vez.
<b>compilar (compile)</b>	Traduzir todo um programa escrito em uma linguagem de alto nível para uma de baixo nível de um só vez, em preparação para uma execução posterior.
<b>código fonte (source code)</b>	Um programa em uma linguagem de alto nível, antes de ter sido compilado.
<b>código objeto (object code)</b>	A saída do compilador, depois que ele traduziu o programa.
<b>executável (executable)</b>	Um outro nome para código objeto que está pronto para ser executado.
<b>script</b>	Um programa guardado em um arquivo (normalmente um que será interpretado).
<b>programa (program)</b>	Conjunto de instruções que especifica uma computação.

<b>(algorithm)</b>	categoria de problemas.
<b>bug</b>	Erro em um programa.
<b>depuração (debugging)</b>	O processo de encontrar e remover qualquer um dos três tipos de erros de programação.
<b>sintaxe (syntax)</b>	A estrutura de um programa.
<b>erro de sintaxe (syntax error)</b>	Erro em um programa, que torna impossível a análise sintática (logo, também impossível a interpretação).
<b>erro em tempo de execução (runtime error)</b>	Erro que não ocorre até que o programa seja executado, mas que impede que o programa continue.
<b>exceção (exception)</b>	Um outro nome para um erro em tempo de execução ou erro de runtime.
<b>erro de semântica (semantic error)</b>	Erro em um programa, que o leva a fazer algo diferente do que pretendia o programador.
<b>semântica (semantics)</b>	O significado de um programa.
<b>linguagem natural (natural language)</b>	Qualquer língua falada pelos seres humanos que tenha evoluído naturalmente.
<b>linguagem formal (formal language)</b>	Qualquer linguagem desenvolvida pelas pessoas para propósitos específicos, tais como, a representação de idéias matemáticas ou programas de computadores; todas as linguagens de programação são linguagens formais.
<b>átomo (token)</b>	Um elemento básico da estrutura sintática de um programa, análogo a uma palavra em uma linguagem natural.
<b>análise sintática (parse)</b>	Examinar um programa e analisar sua estrutura sintática.
<b>comando print</b>	Instrução que leva o interpretador

`( print`  
statement)`

Python a apresentar um valor na tela.



## Capítulo 2: Variáveis, expressões e comandos

### 2.1 Valores e tipos

O **valor** (por exemplo, letras e números) é uma das coisas fundamentais que um programa manipula. Os valores que já vimos até agora foram o 2 (como resultado, quando adicionamos 1 + 1) e "Alô, Mundo!".

Esses valores pertencem a **tipos** diferentes: 2 é um inteiro, e "Alô, Mundo!" é uma **string**, assim chamada porque "string", em inglês, quer dizer seqüência, série, cadeia (de caracteres), ou neste caso, "série de letras". Você (e o interpretador) consegue identificar strings porque elas aparecem entre aspas.

O comando print também funciona com inteiros:

```
>>> print 4
4
```

Se você estiver em dúvida sobre qual é o tipo de um determinado valor, o interpretador pode revelar:

```
>>> type("Alô, Mundo!")
<type 'string'>
>>> type(17)
<type 'int'>
```

Nenhuma surpresa: strings pertencem ao tipo string e inteiros pertencem ao tipo int. Menos obviamente, números com um ponto decimal pertencem a um tipo chamado float, porque estes números são representados em um formato chamado **ponto flutuante**!

```
>>> type(3.2)
<type 'float'>
```

O que dizer de valores como "17" e "3.2"? Eles parecem números, mas estão entre aspas, como strings:

```
>>> type("17")
<type 'string'>
>>> type("3.2")
<type 'string'>
```

Eles são strings.

Ao digitar um número grande, é tentador usar pontos entre grupos de três dígitos, assim: 1.000.000. Isso não funciona porque Python usa o ponto como separador decimal. Usar a vírgula, como se faz em inglês, resulta numa expressão válida, mas não no número que queríamos representar:

```
>>> print 1,000,000
1 0 0
```

Não é nada do que se esperava! Python interpreta 1,000,000 como uma tupla, algo que veremos no Capítulo 9. Por hora, lembre-se apenas de não colocar vírgulas nos números.

### 2.2 Variáveis

Uma das características mais poderosas de uma linguagem de programação é a habilidade de manipular variáveis. Uma

1 N.T.: Observe o uso de ponto no lugar da vírgula para separar a parte inteira da parte fracionária.

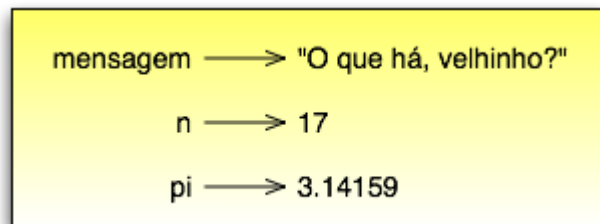
variável é um nome que se refere a um valor.

O comando de atribuição cria novas variáveis e dá a elas valores:

```
>>> mensagem = "E aí, Doutor?"
>>> n = 17
>>> pi = 3.14159
```

Este exemplo faz três atribuições. A primeira atribui a string "E aí, Doutor?" a uma nova variável chamada mensagem. A segunda dá o valor inteiro 17 a n, e a terceira atribui o número de ponto flutuante 3.14159 à variável chamada pi.

Uma maneira comum de representar variáveis no papel é escrever o nome delas com uma seta apontando para o valor da variável. Esse tipo de figura é chamado de diagrama de estado porque mostra em que estado cada variável está (pense nisso como o estado de espírito da variável). O diagrama a seguir mostra o resultado das instruções de atribuição:



O comando print também funciona com variáveis:

```
>>> print mensagem
E aí, Doutor?
>>> print n
17
>>> print pi
3.14159
```

Em cada um dos casos, o resultado é o valor da variável. Variáveis também têm tipo; novamente, podemos perguntar ao interpretador quais são eles:

```
>>> type(mensagem)
<type 'string'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

O tipo de uma variável é o tipo do valor ao qual ela se refere.

### 2.3 Nomes de variáveis e palavras reservadas

Os programadores geralmente escolhem nomes significativos para suas variáveis -- eles documentam para o quê a variável é usada.

Nomes de variáveis podem ser arbitrariamente longos. Eles podem conter tanto letras quanto números, mas

## Como pensar como um cientista da Computação usando Python

têm de começar com uma letra. Embora seja válida a utilização de letras maiúsculas, por convenção, não usamos. Se você o fizer, lembre-se de que maiúsculas e minúsculas são diferentes. Bruno e bruno são variáveis diferentes.

O caractere para sublinhado ( `_` ) pode aparecer em um nome. Ele é muito utilizado em nomes com múltiplas palavras, tal como em `meu_nome` ou `preco_do_cha_na_china`.

Se você der a uma variável um nome inválido, causará um erro de sintaxe:

```
>>> 76trombones = "grande parada"
SyntaxError: invalid syntax
>>> muito$ = 1000000
SyntaxError: invalid syntax
>>> class = "Ciencias da Computacao 101"
SyntaxError: invalid syntax
```

`76trombones` é inválida porque não começa com uma letra. `muito$` é inválida porque contém um caractere ilegal, o cifrão. Mas o que está errado com `class`?

Ocorre que `class` é uma das **palavras reservadas** em Python. Palavras reservadas definem as regras e a estrutura da linguagem e não podem ser usadas como nomes de variáveis.

Python tem 29 palavras reservadas:

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

Pode ser útil ter essa lista à mão. Se o interpretador acusar erro sobre um de seus nomes de variável e você não souber porquê, veja se o nome está na lista.

### 2.4 Comandos

Um comando é uma instrução que o interpretador Python pode executar. Vimos até agora dois tipos de comandos: de exibição (`print`) e de atribuição.

Quando você digita um comando na linha de comando, o Python o executa e mostra o resultado, se houver um. O resultado de um comando `print` é a exibição de um valor. Comandos de atribuição não produzem um resultado visível.

Um script normalmente contém uma seqüência de comandos. Se houver mais de um comando, os resultados aparecerão um de cada vez, conforme cada comando seja executado.

Por exemplo, o "script":

```
print 1
x = 2
print 2
```

produz a saída:

```
1
2
```

Novamente, o comando de atribuição não produz saída.

### 2.5 Avaliando expressões

Uma expressão é uma combinação de valores, variáveis e operadores. Se você digitar uma expressão na linha de

comando, o interpretador avalia e exhibe o resultado:

```
>>> 1 + 1
2
```

Embora expressões contêmam valores, variáveis e operadores, nem toda expressão contém todos estes elementos. Um valor por si só é considerado uma expressão, do mesmo modo que uma variável:

```
>>> 17
17
>>> x
2
```

Avaliar uma expressão não é exatamente a mesma coisa que imprimir um valor:

```
>>> mensagem = "E aí, Doutor?"
>>> mensagem
'E aí, Doutor?'
>>> print mensagem
E aí, Doutor?
```

Quando Python exhibe o valor de uma expressão, usa o mesmo formato que você usaria para entrar com o valor. No caso de strings, isso significa que as aspas são incluídas [#]\_. Mas o comando `print` imprime o valor da expressão, que, neste caso, é o conteúdo da string.

Num script, uma expressão sozinha é um comando válido, porém sem efeito. O script:

```
17
3.2
"Alô, Mundo!"
1 + 1
```

não produz qualquer saída. Como você mudaria o "script" para exibir os valores destas quatro expressões?

### 2.6 Operadores e operandos

Operadores são símbolos especiais que representam computações como adição e multiplicação. Os valores que o operador usa são chamados operandos.

Todas as expressões seguintes são válidas em Python e seus significados são mais ou menos claros:

```
20+32      hora-1      hora*60+minuto      minuto/60      5**2 \
(5+9)*(15-7)
```

Em Python, os símbolos `+`, `-`, `/` e o uso de parênteses para agrupamento têm o mesmo significado que em matemática. O asterisco (`*`) é o símbolo para multiplicação e `**` é o símbolo para potenciação.

Quando um nome de variável aparece no lugar de um operando, ele é substituído pelo valor da variável, antes da operação ser executada.

Adição, subtração, multiplicação e potenciação fazem o que se espera, mas você pode ficar surpreso com a divisão. A operação seguinte tem um resultado inesperado:

```
>>> minuto = 59
>>> minuto/60
0
```

O valor de `minuto` é 59 e, em aritmética convencional, 59 dividido por 60 é 0,98333, não 0. A razão para a discrepância é que Python está realizando uma **divisão inteira**.

Quando ambos os operandos são inteiros, o resultado tem de ser também um inteiro e, por convenção, a divisão inteira sempre arredonda para baixo, mesmo em casos

como este, em que o inteiro seguinte está muito próximo:

```
>>> minuto*100/60
98
```

De novo, o resultado é arredondado para baixo, mas agora pelo menos a resposta é aproximadamente correta. A alternativa é usar a divisão em ponto flutuante, o que veremos no capítulo 3.

### 2.7 Ordem dos operadores

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das **regras de precedência**. Python segue as mesmas regras de precedência para seus operadores matemáticos que a matemática. O acrônimo **PEMDAS** é uma maneira prática de lembrar a ordem das operações:

- Parênteses têm a mais alta precedência e podem ser usados para forçar uma expressão a ser avaliada na ordem que você quiser. Já que expressões entre parênteses são avaliadas primeiro,  $2 * (3-1)$  é 4, e  $(1+1)**(5-2)$  é 8. Você também pode usar parênteses para tornar uma expressão mais fácil de ler, como em  $(minuto * 100) / 60$ , ainda que isso não altere o resultado.
- Exponenciação ou potenciação tem a próxima precedência mais alta, assim  $2**1+1$  é 3 e não 4, e  $3*1**3$  é 3 e não 27.
- Multiplicação e Divisão têm a mesma precedência, que é mais alta do que a da Adição e da Subtração, que também têm a mesma precedência. Assim  $2*3-1$  dá 5 em vez de 4, e  $2/3-1$  é -1, não 1 (lembre-se de que na divisão inteira,  $2/3=0$ ).
- Operadores com a mesma precedência são avaliados da esquerda para a direita. Assim, na expressão  $minuto*100/60$ , a multiplicação acontece primeiro, resultando em  $5900/60$ , o que se transforma produzindo 98. Se as operações tivessem sido avaliadas da direita para a esquerda, o resultado poderia ter sido  $59*1$ , que é 59, que está errado.

### 2.8 Operações com strings

De maneira geral, você não pode executar operações matemáticas em strings, ainda que as strings se pareçam com números. O que segue é inválido (assumindo que mensagem é do tipo string):

```
mensagem-1 "Alô"/123 mensagem*"Alô" "15"+2
```

Interessante é o operador +, que funciona com strings, embora ele não faça exatamente o que você poderia esperar. Para strings, o operador + representa concatenação, que significa juntar os dois operandos ligando-os pelos extremos. Por exemplo:

```
fruta = "banana"
assada = " com canela"
print fruta + assada
```

A saída deste programa é banana com canela. O espaço antes da palavra com é parte da string e é necessário para produzir o espaço entre as strings concatenadas.

O operador \* também funciona com strings; ele realiza repetição. Por exemplo,  $"Legal"*3$  é "LegalLegalLegal". Um dos operadores tem que ser uma string; o outro tem que ser um inteiro.

Por um lado, esta interpretação de + e \* faz sentido pela analogia entre adição e multiplicação. Assim

como  $4*3$  equivale a  $4+4+4$ , não é de estranhar que  $"Legal"*3$  seja o mesmo que  $"Legal"+"Legal"+"Legal"$ . Por outro lado, uma diferença significativa separa concatenação e repetição de adição e multiplicação. Você saberia mencionar uma propriedade da adição e da multiplicação que não ocorre na concatenação e na repetição?

### 2.9 Composição

Até agora, vimos os elementos de um programa -- variáveis, expressões, e instruções ou comandos -- isoladamente, sem mencionar como combiná-los.

Uma das características mais práticas das linguagens de programação é a possibilidade de pegar pequenos blocos e combiná-los numa **composição**. Por exemplo, nós sabemos como somar números e sabemos como exibi-los; acontece que podemos fazer as duas coisas ao mesmo tempo:

```
>>> print 17 + 3
20
```

Na realidade, a soma tem que acontecer antes da impressão, assim, as ações não estão na realidade acontecendo ao mesmo tempo. O ponto é que qualquer expressão envolvendo números, strings, e variáveis pode ser usada dentro de um comando print. Você já tinha visto um exemplo disto:

```
print "Número de minutos desde a meia-noite: ", \
hora*60+minuto
```

Esta possibilidade pode não parecer muito impressionante agora, mas você verá outros exemplos em que a composição torna possível expressar computações complexas de modo limpo e conciso.

Atenção: Existem limites quanto ao lugar onde você pode usar certos tipos de expressão. Por exemplo, o lado esquerdo de um comando de atribuição tem que ser um nome de variável, e não uma expressão. Assim, o seguinte não é válido:  $minuto+1 = hora$ .

### 2.11 Glossário

<b>valor (value)</b>	Um número ou string (ou outra coisa que ainda vamos conhecer) que pode ser atribuída a uma variável ou computada em uma expressão.
<b>tipo (type)</b>	Um conjunto de valores. O tipo de um valor determina como ele pode ser usado em expressões. Até agora, os tipos vistos são: inteiros (tipo int), números em ponto-flutuante (tipo float) e strings (tipo string).
<b>ponto-flutuante (floating-point)</b>	Formato para representar números que possuem partes fracionárias.
<b>variável (variable)</b>	Nome que se refere a um valor.
<b>comando (statement)</b>	Trecho de código que representa uma instrução ou ação. Até agora, os comandos vistos foram de atribuição e exibição.
<b>atribuição (assignment)</b>	Comando que atribui um valor a uma variável.
<b>diagrama de estado (state diagram)</b>	Representação gráfica de um conjunto de variáveis e os valores aos quais elas se referem.

<b>palavra-chave (keyword)</b>	Palavra reservada usada pelo compilador para analisar o programa; você não pode usar palavras-chave como if, def, e while como nomes de variáveis.
<b>operador (operator)</b>	Símbolo especial que representa uma computação simples, como adição, multiplicação ou concatenação de strings.
<b>operando (operand)</b>	Um dos valores sobre o qual o operador opera.
<b>expressão (expression)</b>	Combinação de variáveis, operadores e valores, que representa um resultado único.
<b>avaliar (evaluate)</b>	Simplificar uma expressão através da realização de operações, para produzir um valor único.
<b>divisão inteira (integer division)</b>	Operação que divide um inteiro por outro e resulta em um inteiro. A divisão inteira resulta no número de vezes que o numerador é divisível pelo denominador e descarta qualquer resto.
<b>regras de precedência (rules of precedence)</b>	O conjunto de regras que governa a ordem em que expressões envolvendo múltiplos operadores e operandos são avaliadas.
<b>concatenar (concatenate)</b>	Juntar dois operandos lado a lado.
<b>composição (composition)</b>	Habilidade de combinar expressões e comandos simples em expressões e comandos compostos, de forma a representar computações complexas de forma concisa.
<b>comentário (comment)</b>	Informação em um programa dirigida a outros programadores (ou qualquer pessoa que esteja lendo o código fonte) e que não tem efeito na execução do programa.

## Capítulo 3: Funções

### 3.1 Chamadas de funções

Você já viu um exemplo de uma chamada de função:

```
>>> type('32')
<type 'str'>
```

O nome da função é `type` e ela exibe o tipo de um valor ou variável. O valor ou variável, que é chamado de **argumento** da função, tem que vir entre parênteses. É comum se dizer que uma função 'recebe' um valor e 'retorna' um resultado. O resultado é chamado de **valor de retorno**.

Em vez de imprimir um valor de retorno, podemos atribuí-lo a uma variável:

```
>>> bia = type('32')
>>> print bia
<type 'str'>
```

Como outro exemplo, a função `id` recebe um valor ou uma variável e retorna um inteiro, que atua como um identificador único para aquele valor:

```
>>> id(3)
134882108
>>> bia = 3
>>> bia(beth)
134882108
```

Todo valor tem um `id`, que é um número único relacionado ao local onde ele está guardado na memória do computador. O `id` de uma variável é o `id` do valor a qual ela se refere.

### 3.2 Conversão entre tipos

Python provê uma coleção de funções nativas que convertem valores de um tipo em outro. A função `int` recebe um valor e o converte para inteiro, se possível, ou, se não, reclama:

```
>>> int('32')
32
>>> int('Alô')
ValueError: invalid literal for int() : Alô
```

`int` também pode converter valores em ponto flutuante para inteiro, mas lembre que isso trunca a parte fracionária:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

A função `float` converte inteiros e strings em números em ponto flutuante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, a função `str` converte para o tipo `string`:

```
>>> str(32)
```

```
'32'
>>> str(3.14149)
'3.14149'
```

Pode parecer curioso que Python faça distinção entre o valor inteiro 1 e o valor em ponto flutuante 1.0. Eles podem representar o mesmo número, mas pertencem a tipos diferentes. A razão é que eles são representados de modo diferente dentro do computador.

### 3.3 Coerção entre tipos

Agora que podemos converter entre tipos, temos outra maneira de lidar com a divisão inteira. Voltando ao exemplo do capítulo anterior, suponha que queiramos calcular a fração de hora que já passou. A expressão mais óbvia, `minuto / 60`, faz aritmética inteira, assim, o resultado é sempre 0, mesmo aos 59 minutos passados da hora.

Uma solução é converter minuto para ponto flutuante e fazer a divisão em ponto flutuante:

```
>>> minuto = 59
>>> float(minuto) / 60
0.983333333333
```

Opcionalmente, podemos tirar vantagem das regras de conversão automática entre tipos, chamada de **coerção de tipos**. Para os operadores matemáticos, se qualquer operando for um `float`, o outro é automaticamente convertido para `float`:

```
>>> minuto = 59
>>> minuto / 60.0
0.983333333333
```

Fazendo o denominador um `float`, forçamos o Python a fazer a divisão em ponto flutuante.

### 3.4 Funções matemáticas

Em matemática, você provavelmente já viu funções como seno (`sin`) e log, e aprendeu a resolver expressões como  $\sin(\pi/2)$  e  $\log(1/x)$ . Primeiro você resolve a expressão entre parênteses (o argumento). Por exemplo,  $\pi/2$  é aproximadamente 1,571, e  $1/x$  é 0.1 (se  $x$  for 10,0).

Aí você avalia a função propriamente dita, seja procurando numa tabela ou realizando vários cálculos. O  $\sin$  de 1,571 é 1 e o  $\log$  de 0,1 é -1 (assumindo que  $\log$  indica o logaritmo na base 10).

Este processo pode ser aplicado repetidamente para avaliar expressões mais complicadas, como  $\log(1/\sin(\pi/2))$ . Primeiro você avalia o argumento na função mais interna, depois avalia a função e assim por diante.

Python tem um módulo matemático que provê a maioria das funções matemáticas mais familiares. Um **módulo** é um arquivo que contém uma coleção de funções relacionadas agrupadas juntas.

Antes de podermos usar as funções contidas em um módulo, temos de importá-lo:

```
>>> import math
```



Para chamar uma das funções, temos que especificar o nome do módulo e o nome da função, separados por um ponto. Esse formato é chamado de **notação de ponto**:

```
>>> decibel = math.log10(17.0)
>>> angulo = 1.5
>>> altura = math.sin(angulo)
```

A primeira instrução atribui a decibel o logaritmo de 17 na base 10. Existe também uma função chamada log, que pega o logaritmo na base e.

A terceira instrução encontra o seno do valor da variável angulo. sin e as outras funções trigonométricas (cós, tan, etc.) recebem argumentos em radianos. Para converter de graus em radianos, divida por 360 e multiplique por  $2\pi$ . Por exemplo, para encontrar o seno de 45 graus, primeiro calcule o ângulo em radianos e depois ache o seno:

```
>>> graus = 45
>>> angulo = graus * 2 * math.pi / 360.0
>>> math.sin(angulo)
0.707106781187
```

A constante pi também é parte do módulo math. Se você sabe geometria, pode checar o resultado anterior comparando-o com a raiz quadrada de dois dividido por dois:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

### 3.5 Composição

Do mesmo modo como nas funções matemáticas, as funções do Python podem ser compostas, o que significa que você pode usar uma expressão como parte de outra. Por exemplo, você pode usar qualquer expressão como um argumento para uma função:

```
>>> x = math.cos(angulo + pi/2)
```

Esta instrução toma o valor de pi, divide-o por 2, e soma o resultado ao valor de angulo. A soma é então passada como um argumento para a função cos.

Você também pode pegar o resultado de uma função e passá-lo como um argumento para outra:

```
>>> x = math.exp(math.log(10.0))
```

Esta instrução encontra o logaritmo base e de 10 e então eleva a àquela potência. O resultado é atribuído a x.

### 3.6 Adicionando novas funções

Até aqui, temos utilizado somente as funções que vêm com Python, mas também é possível adicionar novas funções. Criar novas funções para resolver seus próprios problemas é uma das coisas mais úteis de uma linguagem de programação de propósito geral.

No contexto de programação, **função** é uma seqüência nomeada de instruções ou comandos, que realizam uma operação desejada. Esta operação é especificada numa **definição de função**. Até agora, as funções que usamos neste livro são pré-definidas e suas definições não foram apresentadas. Isso demonstra que podemos usar funções sem ter que nos preocupar com os detalhes de suas definições.

A sintaxe para uma definição de função é:

```
def NOME( LISTA DE PARAMETROS ) :
    COMANDOS
```

Você pode usar o nome que quiser para as funções que criar, exceto as palavras reservadas do Python. A

lista de parâmetros especifica que informação, se houver alguma, você tem que fornecer para poder usar a nova função.

Uma função pode ter quantos comandos forem necessários, mas eles precisam ser endentados a partir da margem esquerda. Nos exemplos deste livro, usaremos uma endentação de dois espaços.

As primeiras funções que vamos mostrar não terão parâmetros, então, a sintaxe terá esta aparência:

```
def novaLinha():
    print
```

Esta função é chamada de novaLinha. Os parênteses vazios indicam que ela não tem parâmetros. Contém apenas um único comando, que gera como saída um caractere de nova linha (isso é o que acontece quando você usa um comando print sem qualquer argumento).

A sintaxe para a chamada desta nova função é a mesma sintaxe para as funções nativas:

```
print 'Primeira Linha.'
novaLinha()
print 'Segunda Linha.'
```

A saída deste programa é:

```
Primeira Linha.
Segunda Linha.
```

Observe o espaço extra entre as duas linhas. E se quiséssemos mais espaço entre as linhas? Poderíamos chamar a mesma função repetidamente:

```
print 'Primeira Linha.'
novaLinha()
novaLinha()
novaLinha()
print 'Segunda Linha.'
```

Ou poderíamos escrever uma nova função chamada tresLinhas, que produzisse três novas linhas:

```
def tresLinhas() :
    novaLinha()
    novaLinha()
    novaLinha()

print 'Primeira Linha.'
tresLinhas()
print 'Segunda Linha.'
```

Esta função contém três comandos, todos com recuo de dois espaços a partir da margem esquerda. Já que o próximo comando não está endentado, Python reconhece que ele não faz parte da função.

Algumas coisas que devem ser observadas sobre este programa:

1. Você pode chamar o mesmo procedimento repetidamente. Isso é muito comum, além de útil.
2. Você pode ter uma função chamando outra função; neste caso tresLinhas chama novaLinha.

Pode não estar claro, até agora, de que vale o esforço de criar novas funções - existem várias razões, mas este exemplo demonstra duas delas:

- Criar uma nova função permite que você coloque nome em um grupo de comandos. As funções podem simplificar um programa ao ocultar uma computação complexa por trás de um simples comando cujo nome pode ser uma palavra em português, em vez de algum código misterioso.

- Criar uma nova função pode tornar o programa menor, por eliminar código repetido. Por exemplo, um atalho para 'imprimir' nove novas linhas consecutivas é chamar `tresLinhas` três vezes.

Como exercício, escreva uma função chamada `noveLinhas` que use `tresLinhas` para imprimir nove linhas em branco. Como você poderia imprimir vinte e sete novas linhas?

### 3.7 Definições e uso

Reunindo os fragmentos de código da Seção 3.6, o programa completo fica assim:

```
def novaLinha() :
    print

def tresLinhas() :
    novaLinha()
    novaLinha()
    novaLinha()

print 'Primeira Linha.'
tresLinhas()
print 'Segunda Linha.'
```

Esse programa contém duas definições de funções: `novaLinha` e `tresLinhas`. Definições de funções são executadas como quaisquer outros comandos, mas o efeito é criar a nova função. Os comandos dentro da definição da função não são executados até que a função seja chamada, logo, a definição da função não gera nenhuma saída.

Como você já deve ter imaginado, é preciso criar uma função antes de poder executá-la. Em outras palavras, a definição da função tem que ser executada antes que ela seja chamada pela primeira vez.

Como exercício, mova as últimas três linhas deste programa para o topo, de modo que a chamada da função apareça antes das definições. Rode o programa e veja que mensagem de erro você terá.

Também a título de exercício, comece com a versão que funciona do programa e mova a definição de `novaLinha` para depois da definição de `tresLinhas`. O que acontece quando você roda este programa?

### 3.8 Fluxo de execução

Para assegurar que uma função esteja definida antes do seu primeiro uso, é preciso saber em que ordem os comandos são executados, o que é chamado de **fluxo de execução**.

A execução sempre começa com o primeiro comando do programa. Os comandos são executados um de cada vez, pela ordem, de cima para baixo.

As definições de função não alteram o fluxo de execução do programa, mas lembre-se que comandos dentro da função não são executados até a função ser chamada. Embora não seja comum, você pode definir uma função dentro de outra. Neste caso, a definição mais interna não é executada até que a função mais externa seja chamada.

Chamadas de função são como um desvio no fluxo de execução. Em vez de ir para o próximo comando, o fluxo salta para a primeira linha da função chamada, executa todos os comandos lá e então volta atrás para retomar de onde havia deixado.

Parece muito simples, até a hora em que você

lembra que uma função pode chamar outra. Enquanto estiver no meio de uma função, o programa poderia ter de executar os comandos em uma outra função. Mas enquanto estivesse executando esta nova função, o programa poderia ter de executar ainda outra função!

Felizmente, Python é adepto de monitorar a posição onde está, assim, cada vez que uma função se completa, o programa retoma de onde tinha parado na função que a chamou. Quando chega ao fim do programa, ele termina.

Qual a moral dessa história sórdida? Quando você ler um programa, não o leia de cima para baixo. Em vez disso, siga o fluxo de execução.

### 3.9 Parâmetros e argumentos

Algumas das funções nativas que você já usou requerem argumentos, aqueles valores que controlam como a função faz seu trabalho. Por exemplo, se você quer achar o seno de um número, você tem que indicar qual número é. Deste modo, sin recebe um valor numérico como um argumento.

Algumas funções recebem mais de um argumento. Por exemplo, `pow` recebe dois argumentos, a base e o expoente. Dentro da função, os valores que lhe são passados são atribuídos a variáveis chamadas **parâmetros**.

Veja um exemplo de uma função definida pelo usuário, que recebe um parâmetro:

```
def imprimeDobrado(bruno):
    print bruno, bruno
```

Esta função recebe um único argumento e o atribui a um parâmetro chamado `bruno`. O valor do parâmetro (a essa altura, não sabemos qual será) é impresso duas vezes, seguido de uma nova linha. Estamos usando `bruno` para mostrar que o nome do parâmetro é decisão sua, mas claro que é melhor escolher um nome que seja mais ilustrativo.

A função `imprimeDobrado` funciona para qualquer tipo que possa ser impresso:

```
>>> imprimeDobrado('Spam')
Spam Spam
>>> imprimeDobrado(5)
5 5
>>> imprimeDobrado(3.14159)
3.14159 3.14159
```

Na primeira chamada da função, o argumento é uma string. Na segunda, é um inteiro. Na terceira é um float.

As mesmas regras de composição que se aplicam a funções nativas também se aplicam às funções definidas pelo usuário, assim, podemos usar qualquer tipo de expressão como um argumento para `imprimeDobrado`:

```
>>> imprimeDobrado('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> imprimeDobrado(math.cos(math.pi))
-1.0 -1.0
```

Como acontece normalmente, a expressão é avaliada antes da execução da função, assim `imprimeDobrado` imprime `SpamSpamSpamSpam SpamSpamSpamSpam` em vez de `'Spam'*4 'Spam'*4`.

Como exercício, escreva um chamada a `imprimeDobrado` que imprima `'Spam'*4 'Spam'*4`. Dica: strings podem ser colocadas tanto entre aspas simples quanto duplas e o tipo de aspas que não for usado para envolver a string pode ser usado dentro da string, como parte dela.

Também podemos usar uma variável como argumento:

```
>>> miguel = 'Eric, the half a bee.'
>>> imprimeDobrado(miguel)
Eric, the half a bee. Eric, the half a bee.
```

N.T.: "Eric, the half a bee" é uma música do grupo humorístico britânico Monty Python. A linguagem Python foi batizada em homenagem ao grupo e, por isso, os programadores gostam de citar piadas deles em seus exemplos.

Repare numa coisa importante: o nome da variável que passamos como um argumento (miguel) não tem nada a ver com o nome do parâmetro (bruno). Não importa de que modo o valor foi chamado de onde veio (do 'chamador'); aqui, em `imprimeDobrado`, chamamos a todo mundo de bruno.

### 3.10 Variáveis e parâmetros são locais

Quando você cria uma variável local dentro de uma função, ela só existe dentro da função e você não pode usá-la fora de lá. Por exemplo:

```
def concatDupla(partel1, parte2)
    concat = partel1 + parte2
    imprimeDobrado(concat)
```

Esta função recebe dois argumentos, concatena-os, e então imprime o resultado duas vezes. Podemos chamar a função com duas strings:

```
>>> canto1 = 'Pie Jesu domine, '
>>> canto2 = 'dona eis requiem. '
>>> concatDupla(canto1, canto2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, \
Dona eis requiem.
```

Quando a função `concatDupla` termina, a variável `concat` é destruída. Se tentarmos imprimi-la, teremos um erro:

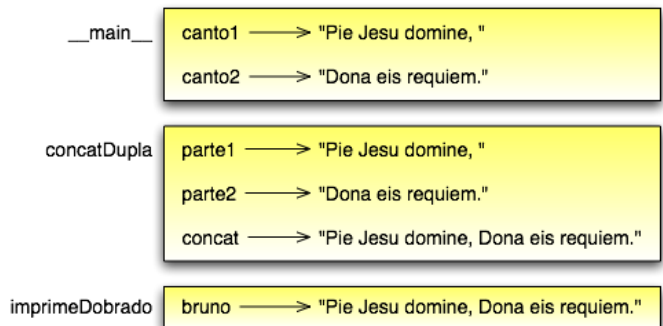
```
>>> print concat
NameError: concat
```

Parâmetros são sempre locais. Por exemplo, fora da função `imprimeDobrado`, não existe alguma coisa chamada `bruno`. Se você tentar utilizá-la, Python vai reclamar.

### 3.11 Diagramas da pilha

Para entender que variáveis podem ser usadas aonde, às vezes é útil desenhar um **diagrama da pilha**. Como os diagramas de estado, diagramas da pilha mostram o valor de cada variável, mas também a função à qual cada variável pertence.

Cada função é representada por um **frame** (quadro). Um frame é uma caixa com o nome de uma função ao lado dela e os parâmetros e variáveis da função dentro dela. O diagrama de pilha para o exemplo anterior tem a seguinte aparência:



A ordem da pilha mostra o fluxo de execução. `imprimeDobrado` foi chamado por `concatDupla`, e `concatDupla` foi chamado por `__main__` (principal), que é um nome especial para a função mais no topo. Quando você cria uma variável fora de qualquer função, ela pertence à `__main__`.

Cada parâmetro se refere ao mesmo valor que o seu argumento correspondente. Assim, `partel1` tem o mesmo valor de `canto1`, `parte2` tem o mesmo valor de `canto2` e `bruno` tem o mesmo valor de `concat`.

Se um erro acontece durante uma chamada de função, Python imprime o nome da função, e o nome da função que a chamou, e o nome da função que chamou a que chamou, percorrendo todo o caminho de volta a `__main__`.

Por exemplo, se tentássemos acessar `concat` de dentro de `imprimeDobrado`, teríamos um `NameError`:

```
Traceback (innermost last):
  File "teste.py", line 13, in __main__
    concatDupla(canto1, canto2)
  File "teste.py", line 5, in concatDupla
    imprimeDobrado(concat)
  File "teste.py", line 9, in imprimeDobrado
    print concat
NameError: concat
```

Esta lista de funções é chamada de **traceback**. Ela mostra em qual arquivo de programa o erro ocorreu, em que linha, e quais funções estavam sendo executadas naquele momento. Mostra também a linha de código que causou o erro.

Note a similaridade entre o traceback e o diagrama da pilha; não é coincidência.

### 3.12 Funções com resultados

A essa altura, você deve ter percebido que algumas das funções que estamos usando, tais como as funções matemáticas, produzem resultados. Outras funções, como `novaLinha`, executam uma ação, mas não retornam um valor. O que levanta algumas questões:

1. O que acontece se você chama uma função e não faz nada com o resultado (por exemplo, não atribui o resultado a uma variável ou o usa como parte de uma expressão maior)?
2. O que acontece se você usa uma função que não produz resultado em uma expressão tal como `novaLinha() + 7`?
3. Você pode escrever funções que produzem resultados, ou está preso a funções como `novaLinha` e `imprimeDobrado`?

A resposta para a terceira questão é afirmativa e nós vamos fazer isso no Capítulo 5.

A título de exercício, responda as outras duas questões testando-as. Se tiver dúvida sobre o que é válido ou inválido em Python, tente buscar a resposta perguntando ao interpretador.

### 3.13 Glossário

<b>chamada de função (function call)</b>	Comando que executa uma função. Consiste do nome da função seguido de uma lista de argumentos entre parênteses.
<b>argumento (argument)</b>	Valor fornecido a uma função quando ela é chamada. Este valor é atribuído ao parâmetro correspondente na função.
<b>valor de retorno (return value)</b>	O resultado da função. Se uma chamada de função é usada como expressão, o valor de retorno é o valor da expressão.
<b>conversão de tipo (type conversion)</b>	Comando explícito que pega um valor de um tipo e devolve o valor correspondente em outro tipo.
<b>coercividade de tipo (type coercion)</b>	Uma conversão de tipo que ocorre automaticamente, de acordo com as regras de coercividade do Python.
<b>módulo (module)</b>	Arquivo que contém uma coleção de funções e classes relacionadas entre si.
<b>notação de ponto (dot notation)</b>	A sintaxe para chamar uma função que está em outro módulo, especificando o nome do módulo, seguido por um ponto (.) e o nome da função.
<b>função (function)</b>	Seqüência de comandos nomeada, que realiza alguma tarefa útil. As funções podem ou não receber parâmetros e podem ou não retornar valores.
<b>definição de função (function definition)</b>	Comando que cria uma nova função, especificando seu nome, parâmetros e comandos que ela executa.
<b>fluxo de execução (flow of execution)</b>	A ordem na qual os comandos são executados durante a execução do programa.
<b>parâmetro (parameter)</b>	Nome usado numa função para referir-se ao valor passado como argumento.
<b>variável local (local variable)</b>	Variável definida dentro da função. Uma variável local só pode ser usada dentro da função onde foi definida.
<b>diagrama da pilha (stack diagram)</b>	Representação gráfica da pilha de funções, suas variáveis e os valores aos quais elas se referem.
<b>frame</b>	Retângulo no diagrama da pilha que representa uma chamada de função. Contém as variáveis locais e os parâmetros da função.
<b>traceback</b>	Lista de funções que estão em execução, impressa quando um erro de execução ocorre.

## Capítulo 4: Condicionais e recursividade

### 4.1 O operador módulo

O operador **módulo trabalha** com inteiros (e expressões que têm inteiros como resultado) e produz o resto da divisão do primeiro pelo segundo. Em Python, o operador módulo é um símbolo de porcentagem (%). A sintaxe é a mesma que a de outros operadores:

```
>>> quociente = 7 / 3
>>> print quociente
2
>>> resto = 7 % 3
>>> print resto
1
```

Então, 7 dividido por 3 é 2 e o resto é 1.

O operador módulo se revela surpreendentemente útil. Por exemplo, você pode checar se um número é divisível por outro - se  $x \% y$  dá zero, então  $x$  é divisível por  $y$ .

Você também pode extrair o algarismo ou algarismos mais à direita de um número. Por exemplo,  $x \% 10$  resulta o algarismo mais à direita de  $x$  (na base 10). Similarmente,  $x \% 100$  resulta nos dois dígitos mais à direita.

### 4.2 Expressões booleanas

Uma expressão booleana é uma expressão que é verdadeira (true) ou é falsa (false). Em Python, uma expressão que é verdadeira tem o valor 1, e uma expressão que é falsa tem o valor 0.

O operador `==` compara dois valores e produz uma expressão booleana:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

No primeiro comando, os dois operadores são iguais, então a expressão avalia como True (verdadeiro); no segundo comando, 5 não é igual a 6, então temos False (falso).

O operador `==` é um dos **operadores de comparação**; os outros são:

```
x != y      # x é diferente de y
x > y       # x é maior que y
x < y       # x é menor que y
x >= y      # x é maior ou igual a y
x <= y      # x é menor ou igual a y
```

Embora esses operadores provavelmente sejam familiares a você, os símbolos em Python são diferentes dos símbolos da matemática. Um erro comum é usar um sinal de igual sozinho (=) em vez de um duplo (==). Lembre-se de que = é um operador de atribuição e == é um operador de comparação. Também não existem coisas como =< ou =>.

### 4.3 Operadores lógicos

Existem três operadores lógicos: and, or, not (e, ou, não). A semântica (significado) destes operadores é similar aos seus significados em inglês (ou português). Por exemplo,  $x > 0$  and  $x < 10$  é verdadeiro somente se  $x$  for maior que 0 e menor que 10.

$n \% 2 == 0$  or  $n \% 3 == 0$  é verdadeiro se qualquer das condições for verdadeira, quer dizer, se o número  $n$  for divisível por 2 ou por 3.

Finalmente, o operador lógico not nega uma expressão booleana, assim,  $\text{not}(x > y)$  é verdadeiro se  $(x > y)$  for falso, quer dizer, se  $x$  for menor ou igual a  $y$ .

A rigor, os operandos de operadores lógicos deveriam ser expressões booleanas, mas Python não é muito rigoroso. Qualquer número diferente de zero é interpretado como verdadeiro (True):

```
>>> x = 5
>>> x and 1
1
>>> y = 0
>>> y and 1
0
```

Em geral, esse tipo de coisa não é considerado de bom estilo. Se você precisa comparar um valor com zero, deve fazê-lo explicitamente.

### 4.4 Execução condicional

Para poder escrever programas úteis, quase sempre precisamos da habilidade de checar condições e mudar o comportamento do programa de acordo com elas. As **instruções condicionais** nos dão essa habilidade. A forma mais simples é a instrução if (se):

```
if x > 0
    print "x é positivo"
```

A expressão booleana depois da instrução if é chamada de **condição**. Se ela é verdadeira (true), então a instrução endentada é executada. Se não, nada acontece.

Assim como outras instruções compostas, a instrução if é constituída de um cabeçalho e de um bloco de instruções:

```
CABECALHO:
    PRIMEIRO COMANDO
    ...
    ULTIMO COMANDO
```

O cabeçalho começa com uma nova linha e termina com dois pontos (:). Os comandos ou instruções endentados que seguem são chamados de **bloco**. A primeira instrução não endentada marca o fim do bloco. Um bloco de comandos dentro de um comando composto ou instrução composta é chamado de **corpo** do comando.

Não existe limite para o número de instruções que podem aparecer no corpo de uma instrução if, mas tem que haver pelo menos uma. Ocasionalmente, é útil ter um



corpo sem nenhuma instrução (usualmente, como um delimitador de espaço para código que você ainda não escreveu). Nesse caso, você pode usar o comando `pass`, que não faz nada.

### 4.5 Execução alternativa

Um segundo formato da instrução `if` é a execução alternativa, na qual existem duas possibilidades e a condição determina qual delas será executada. A sintaxe se parece com:

```
if x % 2 == 0:
    print x, "é par"
else:
    print x, "é impar"
```

Se o resto da divisão de `x` por 2 for 0, então sabemos que `x` é par, e o programa exibe a mensagem para esta condição. Se a condição é falsa, o segundo grupo de instruções é executado. Desde que a condição deva ser verdadeira (`true`) ou falsa (`false`), precisamente uma das alternativas vai ser executada. As alternativas são chamadas **ramos** (`branches`), porque existem ramificações no fluxo de execução.

Por sinal, se você precisa checar a paridade de números com frequência, pode colocar este código dentro de uma função:

```
def imprimeParidade(x):
    if x % 2 == 0:
        print x, "é par"
    else:
        print x, "é impar"
```

Para qualquer valor de `x`, `imprimeParidade` exibe uma mensagem apropriada. Quando você a chama, pode fornecer uma expressão de resultado inteiro como um argumento:

```
>>> imprimeParidade(17)
>>> imprimeParidade(y+1)
```

### 4.6 Condicionais encadeados

Às vezes existem mais de duas possibilidades e precisamos de mais que dois ramos. Uma **condicional encadeada** é uma maneira de expressar uma computação dessas:

```
if x < y:
    print x, "é menor que", y
elif x > y:
    print x, "é maior que", y
else:
    print x, "e", y, "são iguais"
```

`elif` é uma abreviação de "else if" ("então se"). De novo, precisamente um ramo será executado. Não existe limite para o número de instruções `elif`, mas se existir uma instrução `else` ela tem que vir por último:

```
if escolha == 'A':
    funcaoA()
elif escolha == 'B':
    funcaoB()
elif escolha == 'C':
    funcaoC()
else:
    print "Escolha inválida."
```

Cada condição é checada na ordem. Se a primeira é falsa, a próxima é checada, e assim por diante. Se

uma delas é verdadeira, o ramo correspondente é executado, e a instrução termina. Mesmo que mais de uma condição seja verdadeira, apenas o primeiro ramo verdadeiro executa.

Como exercício, coloque os exemplos acima em funções chamadas `comparar(x, y)` e `executar(escolha)`.

### 4.7 Condicionais aninhados

Um condicional também pode ser aninhado dentro de outra. Poderíamos ter escrito o exemplo tricotômico (dividido em três) como segue:

```
if x == y:
    print x, "e", y, "são iguais"
else:
    if x < y:
        print x, "é menor que", y
    else:
        print x, "é maior que", y
```

O condicional mais externo tem dois ramos. O primeiro ramo contém uma única instrução de saída. O segundo ramo contém outra instrução `if`, que por sua vez tem dois ramos. Os dois ramos são ambos instruções de saída, embora pudessem conter instruções condicionais também.

Embora a endentação das instruções torne a estrutura aparente, condicionais aninhados tornam-se difíceis de ler rapidamente. Em geral, é uma boa idéia evitar o aninhamento quando for possível.

Operadores lógicos freqüentemente fornecem uma maneira de simplificar instruções condicionais aninhadas. Por exemplo, podemos reescrever o código a seguir usando uma única condicional:

```
if 0 < x:
    if x < 10:
        print "x é um número positivo de um só
algarismo."
```

A instrução `print` é executada somente se a fizermos passar por ambos os condicionais, então, podemos usar um operador `and`:

```
if 0 < x and x < 10:
    print "x é um número positivo de um só algarismo."
```

Esses tipos de condição são comuns, assim, Python provê uma sintaxe alternativa que é similar à notação matemática:

```
if 0 < x < 10:
    print "x é um número positivo de um só algarismo."
```

### 4.8 A instrução `return`

O comando `return` permite terminar a execução de uma função antes que ela alcance seu fim. Uma razão para usá-lo é se você detectar uma condição de erro:

```
import math

def imprimeLogaritmo(x):
    if x <= 0:
        print "Somente números positivos, por favor."
        return

    resultado = math.log(x)
    print "O log de x é ", resultado
```

A função `imprimeLogaritmo` recebe um

parâmetro de nome `x`. A primeira coisa que ela faz é checar se `x` é menor ou igual a 0, neste caso ela exibe uma mensagem de erro e então usa `return` para sair da função. O fluxo de execução imediatamente retorna ao ponto chamador, quer dizer, de onde a função foi chamada, e as linhas restantes da função não são executadas.

Lembre-se que para usar uma função do módulo de matemática, `math`, você tem de importá-lo.

### 4.9 Recursividade

Já mencionamos que é válido uma função chamar outra função, e você viu vários exemplos disso. Mas ainda não tínhamos dito que também é válido uma função chamar a si mesma. Talvez não seja óbvio porque isso é bom, mas trata-se de uma das coisas mais mágicas e interessantes que um programa pode fazer. Por exemplo, dê uma olhada na seguinte função:

```
def contagemRegressiva(n):
    if n == 0:
        print "Fogo!"
    else:
        print n
        contagemRegressiva(n-1)
```

`contagemRegressiva` espera que o parâmetro, `n`, seja um inteiro positivo. Se `n` for 0, ela produz como saída a palavra "Fogo!". De outro modo, ela produz como saída `n` e então chama uma função de nome `contagemRegressiva` -- ela mesma -- passando `n-1` como argumento.

O que acontece se chamarmos essa função da seguinte maneira:

```
>>> contagemRegressiva(3)
```

A execução de `contagemRegressiva` começa com `n=3`, e desde que `n` não é 0, produz como saída o valor 3, e então chama a si mesma...

A execução de `contagemRegressiva` começa com `n=2`, e desde que `n` não é 0, produz como saída o valor 2, e então chama a si mesma...

A execução de `contagemRegressiva` começa com `n=1`, e desde que `n` não é 0, produz como saída o valor 1, e então chama a si mesma...

A execução de `contagemRegressiva` começa com `n=0`, e desde que `n` é 0, produz como saída a palavra "Fogo!" e então retorna.

A `contagemRegressiva` que tem `n=1` retorna.

A `contagemRegressiva` que tem `n=2` retorna.

A `contagemRegressiva` que tem `n=1` retorna.

E então estamos de volta em `__main__` (que viagem!). Assim, a saída completa se parece com:

```
3
2
1
Fogo!
```

Como um segundo exemplo, dê uma olhada novamente nas funções `novaLinha` e `tresLinhas`:

```
def novaLinha():
    print

def tresLinhas():
    novaLinha()
    novaLinha()
```

```
novaLinha()
```

Muito embora isso funcione, não seria muito útil se precisássemos gerar como saída 2 novas linhas, ou 106. Uma alternativa melhor seria esta:

```
def nLinhas(n):
    if n > 0:
        print
        nLinhas(n-1)
```

Esse programa é similar a `contagemRegressiva`; sempre que `n` for maior que 0, ele gera como saída uma nova linha e então chama a si mesmo para gerar como saída `n-1` linhas adicionais. Deste modo, o número total de novas linhas é  $1 + (n-1)$  que, se você estudou álgebra direitinho, vem a ser o próprio `n`.

O processo de uma função chamando a si mesma é chamado de **recursividade**, e tais funções são ditas recursivas.

### 4.10 Diagramas de pilha para funções recursivas

Na Seção 3.11, usamos um diagrama de pilha para representar o estado de um programa durante uma chamada de função. O mesmo tipo de diagrama pode ajudar a interpretar uma função recursiva.

Toda vez que uma função é chamada, Python cria um novo quadro (frame) para a função, que contém as variáveis locais e parâmetros da função. Para uma função recursiva, terá que existir mais de um quadro na pilha ao mesmo tempo.

Esta figura mostra um diagrama de pilha para `contagemRegressiva`, chamada com `n = 3`:



Como de costume, no topo da pilha está o quadro para `__main__`. Ele está vazio porque nem criamos qualquer variável em `__main__` nem passamos qualquer valor para ele.

Os quatro quadros `contagemRegressiva` têm valores diferentes para o parâmetro `n`. A parte mais em baixo na pilha, onde `n=0`, é chamada de **caso base**. Ele não faz uma chamada recursiva, então não há mais quadros.

Como exercício, desenhe um diagrama de pilha para `nLinhas` chamada com `n=4`.

### 4.11 Recursividade infinita

Se uma recursividade nunca chega ao caso base, ela prossegue fazendo chamadas recursivas para sempre, e o programa nunca termina. Isto é conhecido como recursividade infinita, e geralmente não é considerada uma boa idéia. Aqui está um programa mínimo com uma recursividade infinita:

```
def recursiva():
    recursiva()
```

Na maioria dos ambientes de programação, um programa com recursividade infinita na verdade não roda para sempre. Python reporta uma mensagem de erro quando a profundidade máxima de recursividade é alcançada:

```
File "<stdin>", line 2, in recursiva
(98 repetitions omitted)
File "<stdin>", line 2, in recursiva
RuntimeError: Maximum recursion depth exceeded
```

Este traceback é um pouco maior do que aquele que vimos no capítulo anterior. Quando o erro ocorre, existem 100 quadros recursiva na pilha!

Como exercício, escreva uma função com recursividade infinita e rode-a no interpretador Python.

### 4.12 Entrada pelo teclado

Os programas que temos escrito até agora são um pouco crus, no sentido de não aceitarem dados entrados pelo usuário. Eles simplesmente fazem a mesma coisa todas as vezes.

Python fornece funções nativas que pegam entradas pelo teclado. A mais simples é chamada `raw_input`. Quando esta função é chamada, o programa pára e espera que o usuário digite alguma coisa. Quando o usuário aperta a tecla Enter ou Return, o programa prossegue e a função `raw_input` retorna o que o usuário digitou como uma string:

```
>>> entrada = raw_input()
0 que você está esperando?
>>> print entrada
0 que você está esperando?
```

Antes de chamar `raw_input`, é uma boa idéia exibir uma mensagem dizendo ao usuário o que ele deve entrar. Esta mensagem é uma `deixa` (prompt). Podemos suprir uma `deixa` como um argumento para `raw_input`:

```
>>> nome = raw_input("Qual... é o seu nome? ")
Qual... é o seu nome? Arthur, Rei dos Bretões!
>>> print nome
Arthur, Rei dos Bretões!
```

Se esperamos que a entrada seja um inteiro, podemos usar a função `input`:

```
deixa = "Qual... é a velocidade de vôo de uma \
andorinha?\n"
velocidade = input(deixa)
```

Se o usuário digita uma string de números, ela é convertida para um inteiro e atribuída a `velocidade`. Infelizmente, se o usuário digitar um caractere que não seja um número, o programa trava:

```
>>> velocidade = input(deixa)
Qual... é a velocidade de vôo de uma andorinha?
De qual você fala, uma andorinha Africana ou uma \
Européia?
SyntaxError: invalid syntax
```

Para evitar esse tipo de erro, geralmente é bom usar `raw_input` para pegar uma string e, então, usar funções de conversão para converter para outros tipos.

### 4.13 Glossário

<b>operador módulo (modulus operator)</b>	Operador denotado por um símbolo de porcentagem (%), que trabalha com inteiros e retorna o resto da divisão de um número por outro.
<b>expressão booleana (boolean expression)</b>	Uma expressão que é verdadeira ou falsa.
<b>operador de comparação (comparison operator)</b>	Um dos operadores que compara dois valores: ==, !=, >, <, >=, e <=.
<b>operador lógico (logical operator)</b>	Um dos operadores que combina expressões booleanas: and, or, e not.
<b>comando condicional (conditional statement)</b>	Comando que controla o fluxo de execução dependendo de alguma condição.
<b>condição (condition)</b>	A expressão booleana que determina qual bloco será executado num comando condicional.
<b>comando composto (compound statement)</b>	Comando que consiste de um cabeçalho e um corpo. O cabeçalho termina com um dois-pontos (:). O corpo é endentado em relação ao cabeçalho.
<b>bloco (block)</b>	Grupo de comandos consecutivos com a mesma endentação.
<b>corpo (body)</b>	O bloco que se segue ao cabeçalho em um comando composto.
<b>aninhamento (nesting)</b>	Estrutura de programa dentro da outra, como um comando condicional dentro de um bloco de outro comando condicional.
<b>recursividade (recursion)</b>	O processo de chamar a própria função que está sendo executada.
<b>caso base (base case)</b>	Bloco de comando condicional numa função recursiva que não resulta em uma chamada recursiva.
<b>recursão infinita (infinite recursion)</b>	Função que chama a si mesma recursivamente sem nunca chegar ao caso base. Após algum tempo, uma recursão infinita causa um erro de execução.
<b>deixa (prompt)</b>	Indicação visual que diz ao usuário que o programa está esperando uma entrada de dados.

## Capítulo 5: Funções frutíferas

### 5.1 Valores de retorno

Algumas das funções nativas do Python que temos usado, como as funções matemáticas, produziram resultados. Chamar a função gerou um novo valor, o qual geralmente atribuímos à uma variável ou usamos como parte de uma expressão:

```
e = math.exp(1.0)
altura = raio * math.sin(angulo)
```

Mas até agora, nenhuma das funções que nós mesmos escrevemos retornou um valor.

Neste capítulo, iremos escrever funções que retornam valores, as quais chamaremos de **funções frutíferas**, ou funções que dão frutos, na falta de um nome melhor. O primeiro exemplo é `area`, que retorna a área de um círculo dado o seu raio:

```
import math

def area(raio):
    temp = math.pi * raio**2
    return temp
```

Já vimos a instrução `return` antes, mas em uma função frutífera a instrução `return` inclui um **valor de retorno**. Esta instrução significa: "Retorne imediatamente desta função e use a expressão em seguida como um valor de retorno". A expressão fornecida pode ser arbitrariamente complicada, de modo que poderíamos ter escrito esta função de maneira mais concisa:

```
def area(raio):
    return math.pi * raio**2
```

Por outro lado, variáveis temporárias como `temp` muitas vezes tornam a depuração mais fácil.

Às vezes é útil ter múltiplos comandos `return`, um em cada ramo de uma condicional:

```
def valorAbsoluto(x):
    if x < 0:
        return -x
    else:
        return x
```

Já que estes comandos `return` estão em ramos alternativos da condicional, apenas um será executado. Tão logo um seja executado, a função termina sem executar qualquer instrução ou comando subsequente.

O código que aparece depois de uma instrução `return`, ou em qualquer outro lugar que o fluxo de execução jamais alcance, é chamado código morto (dead code).

Em uma função frutífera, é uma boa idéia assegurar que todo caminho possível dentro do programa encontre uma instrução `return`. Por exemplo:

```
def valorAbsoluto(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

Este programa não está correto porque se `x` for 0, nenhuma das condições será verdadeira, e a função

terminará sem encontrar um comando `return`. Neste caso, o valor de retorno será um valor especial chamado `None`:

```
>>> print valorAbsoluto(0)
None
```

Como exercício, escreva uma função compare que retorne 1 se `x > y`, 0 se `x == y` e -1 se `x < y`.

### 5.2 Desenvolvimento de programas

Neste ponto, você deve estar apto a olhar para funções completas e dizer o que elas fazem. Também, se você vem fazendo os exercícios, você escreveu algumas pequenas funções. Conforme escrever funções maiores, você pode começar a ter mais dificuldade, especialmente com erros em tempo de execução (erros de runtime) ou erros semânticos.

Para lidar com programas de crescente complexidade, vamos sugerir uma técnica chamada desenvolvimento incremental. A meta do desenvolvimento incremental é evitar seções de depuração (debugging) muito longas pela adição e teste de somente uma pequena quantidade de código de cada vez.

Como exemplo, suponha que você queira encontrar a distância entre dois pontos, dados pelas coordenadas  $(x_1, y_1)$  e  $(x_2, y_2)$ . Pelo teorema de Pitágoras, a distância é:

$$distancia = \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)}$$

O primeiro passo é considerar como deveria ser uma função `distancia` em Python. Em outras palavras, quais são as entradas (parâmetros) e qual é a saída (valor de retorno)?

Neste caso, os dois pontos são as entradas, os quais podemos representar usando quatro parâmetros. O valor de retorno é a distância, que é um valor em ponto flutuante.

Já podemos escrever um esboço da função:

```
def distancia(x1, y1, x2, y2):
    return 0.0
```

Obviamente, esta versão da função não computa distâncias; ela sempre retorna zero. Mas ela está sintaticamente correta, e vai rodar, o que significa que podemos testá-la antes de torná-la mais complicada.

Para testar a nova função, vamos chamá-la com valores hipotéticos:

```
>>> distancia(1, 2, 4, 6)
0.0
```

Escolhemos estes valores de modo que a distância horizontal seja igual a 3 e a distância vertical seja igual a 4; deste modo, o resultado é 5 (a hipotenusa de um triângulo 3-4-5). Quando testamos uma função, é útil sabermos qual o resultado correto.

Neste ponto, já confirmamos que a função está sintaticamente correta, e podemos começar a adicionar linhas de código. Depois de cada mudança adicionada, testamos a função de novo. Se um erro ocorre em qualquer ponto, sabemos aonde ele deve estar: nas linhas adicionadas mais recentemente.



Um primeiro passo lógico nesta computação é encontrar as diferenças  $x_2 - x_1$  e  $y_2 - y_1$ . Nós iremos guardar estes valores em variáveis temporárias chamadas  $dx$  e  $dy$  e imprimi-las:

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print "dx vale", dx
    print "dy vale", dy
    return 0.0
```

Se a função estiver funcionando, as saídas deverão ser 3 e 4. Se é assim, sabemos que a função está recebendo os parâmetros corretos e realizando a primeira computação corretamente. Se não, existem poucas linhas para checar.

Em seguida, computaremos a soma dos quadrados de  $dx$  e  $dy$ :

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dquadrado = dx**2 + dy**2
    print "dquadrado vale: ", dquadrado
    return 0.0
```

Note que removemos os comandos `print` que havíamos escrito no passo anterior. Código como este é chamado de **andaime** porque ajuda a escrever o programa, mas não é parte do produto final.

De novo, nós vamos rodar o programa neste estágio e checar a saída (que deveria ser 25).

Finalmente, se nós tínhamos importado o módulo matemático `math`, podemos usar a função `sqrt` para computar e retornar o resultado:

```
def distancia(x1, x2, y1, y2):
    dx = x2 - x1
    dy = y2 - y1
    dquadrado = dx**2 + dy**2
    resultado = math.sqrt(dquadrado)
    return resultado
```

Se isto funcionar corretamente, você conseguiu. Caso contrário, talvez fosse preciso imprimir (exibir) o valor de resultado antes da instrução `return`.

Enquanto for iniciante, você deve acrescentar apenas uma ou duas linhas de código de cada vez. Conforme ganhar mais experiência, você se verá escrevendo e depurando pedaços maiores. De qualquer modo, o processo de desenvolvimento incremental pode poupar um bocadinho de tempo de depuração.

Os aspectos chave do processo são:

1. Comece com um programa que funciona e faça pequenas mudanças incrementais. Em qualquer ponto do processo, se houver um erro, você saberá exatamente onde ele está.
2. Use variáveis temporárias para manter valores intermediários de modo que você possa exibi-los e checá-los.
3. Uma vez que o programa funcione, você pode querer remover algum código muleta, ou andaime (scaffolding) ou consolidar múltiplos comandos dentro de expressões compostas, mas somente se isto não tornar o programa difícil de ler.

Como um exercício, use o desenvolvimento incremental para escrever uma função chamada `hipotenusa` que retorna a medida da hipotenusa de um triângulo retângulo dadas as medidas dos dois catetos como

parâmetros. Registre cada estágio do desenvolvimento incremental conforme você avança.

### 5.3 Composição

Conforme você poderia esperar agora, você pode chamar uma função de dentro de outra. Esta habilidade é chamada de **composição**.

Como um exemplo, vamos escrever uma função que recebe dois pontos, o centro de um círculo e um ponto em seu perímetro, e calcula a área do círculo.

Assuma que o ponto do centro está guardado nas variáveis  $x_c$  e  $y_c$ , e que o ponto do perímetro está nas variáveis  $x_p$  e  $y_p$ . O primeiro passo é encontrar o raio do círculo, o qual é a distância entre os dois pontos. Felizmente, temos uma função, `distancia`, que faz isto:

```
Raio = distancia(xc, yc, xp, yp)
```

O segundo passo é encontrar a área de um círculo com o raio dado e retorná-la:

```
resultado = area(raio)
return resultado
```

Juntando tudo numa função, temos:

```
def area2(xc, yc, xp, yp):
    raio = distancia(xc, yc, xp, yp)
    resultado = area(raio)
    return resultado
```

Chamamos à esta função de `area2` para distinguir da função `area`, definida anteriormente. Só pode existir uma única função com um determinado nome em um determinado módulo.

As variáveis temporárias `raio` e `resultado` são úteis para o desenvolvimento e para depuração (debugging), mas uma vez que o programa esteja funcionando, podemos torná-lo mais conciso através da composição das chamadas de função:

```
def area2(xc, yc, xp, yp):
    return area(distancia(xc, yc, xp, yp))
```

Como exercício, escreva uma função `inclinacao(x1, y1, x2, y2)` que retorne a inclinação (ou coeficiente angular?) de uma linha dados os pontos  $(x_1, y_1)$  e  $(x_2, y_2)$ . Depois use esta função em uma função chamada `cortaY(x1, y1, x2, y2)` que retorne a interseção da linha com o eixo  $y$ , dados os pontos  $(x_1, y_1)$  e  $(x_2, y_2)$ .

### 5.4 Funções booleanas

Funções podem retornar valores booleanos, o que muitas vezes é conveniente por ocultar testes complicados dentro de funções. Por exemplo:

```
def ehDivisivel(x, y):
    If x % y == 0:
        return True # é verdadeiro (true), é divisível
    else:
        return False # é falso (false), não é divisível
```

O nome desta função é `ehDivisivel` ("é divisível"). É comum dar a uma função booleana nomes que soem como perguntas sim/não. `ehDivisivel` retorna ou `True` ou `False` para indicar se  $x$  é ou não é divisível por  $y$ .

Podemos tornar a função mais concisa se tirarmos vantagem do fato de a condição da instrução `if` ser ela mesma uma expressão booleana. Podemos retorná-la diretamente, evitando totalmente o `if`:



```
def ehDivisivel(x, y):
    return x % y == 0
```

Esta sessão mostra a nova função em ação:

```
>>> ehDivisivel(6, 4)
False
>>> ehDivisivel(6, 3)
True
```

Funções booleanas são freqüentemente usadas em comandos condicionais:

```
if ehDivisivel(x, y):
    print "x é divisível por y"
else:
    print "x não é divisível por y"
```

Mas a comparação extra é desnecessária.

Como exercício, escreva uma função estaEntre(x, y, z) que retorne True se  $y < x < z$  ou False se não.

### 5.5 Mais recursividade

Até aqui, você aprendeu apenas um pequeno subconjunto da linguagem Python, mas pode ser que te interesse saber que este pequeno subconjunto é uma linguagem de programação completa, o que significa que qualquer coisa que possa ser computada pode ser expressa nesta linguagem. Qualquer programa já escrito pode ser reescrito usando somente os aspectos da linguagem que você aprendeu até agora (usualmente, você precisaria de uns poucos comandos para controlar dispositivos como o teclado, mouse, discos, etc., mas isto é tudo).

Provar esta afirmação é um exercício nada trivial, que foi alcançado pela primeira vez por Alan Turing, um dos primeiros cientistas da computação (alguém poderia dizer que ele foi um matemático, mas muitos dos primeiros cientistas da computação começaram como matemáticos). Por isso, ficou conhecido como Tese de Turing. Se você fizer um curso em Teoria da Computação, você terá chance de ver a prova.

Para te dar uma idéia do que você pode fazer com as ferramentas que aprendeu a usar até agora, vamos avaliar algumas funções matemáticas recursivamente definidas. Uma definição recursiva é similar à uma definição circular, no sentido de que a definição faz referência à coisa que está sendo definida. Uma verdadeira definição circular não é muito útil:

**vorpal:** adjetivo usado para descrever algo que é vorpal.

Se você visse esta definição em um dicionário, ficaria confuso. Por outro lado, se você procurasse pela definição da função matemática fatorial, você encontraria algo assim:

```
0! = 1
n! = n.(n-1)!
```

Esta definição diz que o fatorial de 0 é 1, e que o fatorial de qualquer outro valor, n, é n multiplicado pelo fatorial de n-1.

Assim, 3! (lê-se "3 fatorial" ou "fatorial de 3") é 3 vezes 2!, o qual é 2 vezes 1!, o qual é 1 vezes 0!. Colocando tudo isso junto, 3! igual 3 vezes 2 vezes 1 vezes 1, o que é 6.

Se você pode escrever uma definição recursiva de alguma coisa, você geralmente pode escrever um programa em Python para executá-la. O primeiro passo é decidir quais são os parâmetros para esta função. Com pouco esforço, você deverá concluir que fatorial recebe um único parâmetro:

```
def fatorial(n):
```

Se acontece de o argumento ser 0, tudo o que temos de fazer é retornar 1:

```
def fatorial(n):
    if n == 0:
        return 1
```

Por outro lado, e esta é a parte interessante, temos que fazer uma chamada recursiva para encontrar o fatorial de n-1 e então multiplicá-lo por n:

```
def fatorial(n):
    if n == 0:
        return 1
    else:
        recursivo = fatorial(n-1)
        resultado = n * recursivo
        return resultado
```

O fluxo de execução para este programa é similar ao fluxo de contagemRegressiva na Seção 4.9. Se chamarmos fatorial com o valor 3:

Já que 3 não é 0, tomamos o segundo ramo e calculamos o fatorial de n-1 ...

Já que 2 não é 0, tomamos o segundo ramo e calculamos o fatorial de n-1 ...

Já que 1 não é 0, tomamos o segundo ramo e calculamos o fatorial de n-1 ...

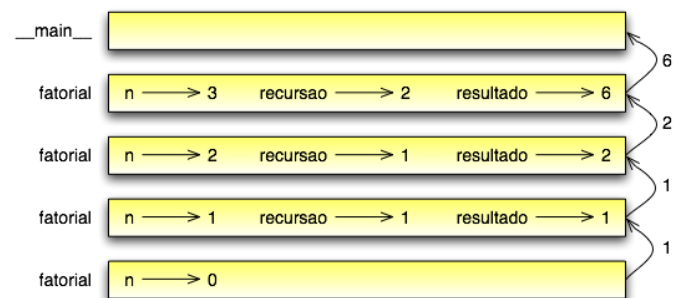
Já que 0 é 0, tomamos o primeiro ramo e retornamos 1 sem fazer mais qualquer chamada recursiva.

O valor retornado (1) é multiplicado por n, que é 1, e o resultado é retornado.

O valor retornado (1) é multiplicado por n, que é 2, e o resultado é retornado.

O valor retornado (2) é multiplicado por n, que é 1, e o resultado, 6, se torna o valor de retorno da chamada de função que iniciou todo o processo.

Eis o diagrama de pilha para esta seqüência de chamadas de função:



Os valores de retorno são mostrados sendo passados de volta para cima da pilha. Em cada quadro, o valor de retorno é o valor de resultado, o qual é o produto de n por recursivo.

### 5.6 Voto de confiança (Leap of faith)

Seguir o fluxo de execução é uma maneira de ler programas, mas que pode rapidamente se transformar em um labirinto. Uma alternativa é o que chamamos de "voto de confiança". Quando você tem uma chamada de função, em vez de seguir o fluxo de execução, você assume que a função funciona corretamente e retorna o valor apropriado.

De fato, você está agora mesmo praticando este voto de confiança ao usar as funções nativas. Quando você chama `math.cos` ou `math.exp`, você não examina a implementação destas funções. Você apenas assume que elas funcionam porque as pessoas que escreveram as bibliotecas nativas eram bons programadores.

O mesmo também é verdade quando você chama uma de suas próprias funções. Por exemplo, na Seção 5.4, escrevemos a função chamada `ehDivisivel` que determina se um número é divisível por outro. Uma vez que nos convencemos que esta função está correta -- ao testar e examinar o código -- podemos usar a função sem examinar o código novamente.

O mesmo também é verdadeiro para programas recursivos. Quando você tem uma chamada recursiva, em vez de seguir o fluxo de execução, você poderia assumir que a chamada recursiva funciona (produz o resultado correto) e então perguntar-se, "Assumindo que eu possa encontrar o fatorial de  $n-1$ , posso calcular o fatorial de  $n$ ?" Neste caso, é claro que você pode, multiplicando por  $n$ .

Naturalmente, é um pouco estranho que uma função funcione corretamente se você ainda nem terminou de escrevê-la, mas é por isso que se chama voto de confiança!

### 5.7 Mais um exemplo

No exemplo anterior, usamos variáveis temporárias para explicitar (spell out XXX) os passos e tornar o código mais fácil de depurar, mas poderíamos ter economizado algumas linhas:

```
def fatorial(n):
    if n == 0:
        return 1
    else:
        return n * fatorial(n-1)
```

De agora em diante, tenderemos a utilizar um formato mais conciso, mas recomendamos que você use a versão mais explícita enquanto estiver desenvolvendo código. Quando ele estiver funcionando, você pode enxugá-lo se estiver se sentindo inspirado.

Depois de `fatorial`, o exemplo mais comum de uma função matemática definida recursivamente é `fibonacci`, a qual tem a seguinte definição:

```
fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
```

Traduzido em Python, parecerá assim:

```
def fibonacci(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Se você tentar seguir o fluxo de execução aqui, mesmo para valores bem pequenos de  $n$ , sua cabeça explodirá. Mas, de acordo com o voto de confiança, se você assume que as duas chamadas recursivas funcionam corretamente, então é claro que você terá o resultado correto ao juntá-las.

### 5.8 Checagem de tipos

O que acontece se chamamos `fatorial` e damos a ela 1.5 como argumento?:

```
>>> fatorial(1.5)
```

```
RuntimeError: Maximum recursion depth exceeded
```

Parece um caso de recursividade infinita. Mas o que será que é de fato? Existe um caso base -- quando  $n == 0$ . O problema é que o valor de  $n$  nunca encontra o caso base.

Na primeira chamada recursiva, o valor de  $n$  é 0.5. Na próxima, ele é igual a -0.5. Daí em diante, ele se torna cada vez menor, mas jamais será 0.

Temos então duas alternativas. Podemos tentar generalizar a função fatorial para que funcione com números em ponto flutuante, ou fazemos fatorial realizar a checagem de tipo de seus parâmetros. A primeira é chamada função `gamma` e está um pouco além do escopo deste livro. Sendo assim, ficaremos com a segunda.

Podemos usar `type` para comparar o tipo do parâmetro com o tipo de um valor inteiro conhecido (como 1). Ao mesmo tempo em que fazemos isto, podemos nos certificar também de que o parâmetro seja positivo:

```
def fatorial(n):
    if type(n) != type(1):
        print "Fatorial somente é definido para inteiros."
        return -1
    elif n < 0:
        print "Fatorial somente é definido para inteiros\ positivos."
        return -1
    elif n == 0:
        return 1
    else:
        return n * fatorial(n-1)
```

Agora temos três casos base. O primeiro pega os não-inteiros. O segundo pega os inteiros negativos. Em ambos os casos, o programa exibe uma mensagem de erro e retorna um valor especial, -1, para indicar que alguma coisa saiu errada:

```
>>> fatorial("Fred")
Fatorial somente é definido para inteiros.
-1
>>> fatorial(-2)
Fatorial somente é definido para inteiros positivos.
-1
```

Se passarmos pelas duas checagens, então saberemos que  $n$  é um inteiro positivo, e poderemos provar que a recursividade encontra seu término.

Este programa demonstra um padrão (pattern) chamado às vezes de **guardião**. As duas primeiras condicionais atuam como guardiãs, protegendo o código que vem em seguida de valores que poderiam causar um erro. Os guardiões tornam possível garantir a correção do código.

### 5.9 Glossário

<b>função frutífera (fruitful function)</b>	Uma função que produz um valor de retorno.
<b>valor de retorno (return value)</b>	O valor entregue como resultado de uma chamada de função.
<b>variável temporária (temporary variable)</b>	Uma variável usada para guardar um valor intermediário em um cálculo complexo.
<b>código morto (dead code)</b>	Parte de um programa que nunca pode ser executada, muitas vezes por que ela aparece depois de uma instrução <code>return</code> .

## Como pensar como um cientista da Computação usando Python

---

<b>None</b>	Um valor especial em Python, retornado por funções que não possuem uma instrução return ou têm uma instrução return sem argumento.
<b>desenvolvimento incremental (incremental development)</b>	Uma estratégia de desenvolvimento de programas que evita a depuração ao adicionar e testar somente uma pequena quantidade de código de cada vez.
<b>andaime (scaffolding)</b>	Código usado durante o desenvolvimento do programa, mas que não faz parte do produto final.
<b>guardião (guardian)</b>	Uma condição que checa e manipula circunstâncias que poderiam causar um erro.

## Capítulo 6: Iteração

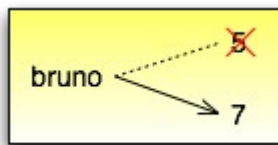
### 6.1 Reatribuições

Como você talvez já tenha descoberto, é permitido fazer mais de uma atribuição à mesma variável. Uma nova atribuição faz uma variável existente referir-se a um novo valor (sem se referir mais ao antigo):

```
bruno = 5
print bruno,
bruno = 7
print bruno
```

A saída deste programa é 5 7, porque na primeira vez que bruno é impresso, seu valor é 5 e na segunda vez, seu valor é 7. A vírgula no final do primeiro comando `print` suprime a nova linha no final da saída, que é o motivo pelo qual as duas saídas aparecem na mesma linha.

Veja uma **reatribuição** em um diagrama de estado:



Com a reatribuição torna-se ainda mais importante distinguir entre uma operação de atribuição e um comando de igualdade. Como Python usa o sinal de igual (=) para atribuição, existe a tendência de lermos um comando como `a = b` como um comando de igualdade. Mas não é!

Em primeiro lugar, igualdade é comutativa e atribuição não é. Por exemplo, em matemática, se  $a = 7$  então  $7 = a$ . Mas em Python, o comando `a = 7` é permitido e `7 = a` não é.

Além disso, em matemática, uma expressão de igualdade é sempre verdadeira. Se  $a = b$  agora, então,  $a$  será sempre igual a  $b$ . Em Python, um comando de atribuição pode tornar duas variáveis iguais, mas elas não têm que permanecer assim:

```
a = 5
b = a # a e b agora são iguais
b = 3 # a e b não são mais iguais
```

A terceira linha muda o valor de `a` mas não muda o valor de `b`, então, elas não são mais iguais. (Em algumas linguagens de programação, um símbolo diferente é usado para atribuição, como `<-` ou `:=`, para evitar confusão.)

Embora a reatribuição seja freqüentemente útil, você deve usá-la com cautela. Se o valor das variáveis muda freqüentemente, isto pode fazer o código difícil de ler e de depurar.

### 6.2 O comando while

Os computadores são muito utilizados para automatizar tarefas repetitivas. Repetir tarefas idênticas ou similares sem cometer erros é uma coisa que os computadores fazem bem e que as pessoas fazem poorly.

Vimos dois programas, `nLinhas` e `contagemRegressiva`, que usam recursividade (recursão) para fazer a repetição, que também é chamada **iteração**. Porque a iteração é muito comum, Python tem várias características para torná-la mais fácil. A primeira delas em que vamos dar uma olhada é o comando `while`.

Aqui está como fica `contagemRegressiva` com um comando `while`:

```
def contagemRegressiva(n):
    while n > 0:
        print n
        n = n-1
    print "Fogo!"
```

Desde que removemos a chamada recursiva, esta função não é recursiva.

Você quase pode ler o comando `while` como se fosse Inglês. Ele significa, "Enquanto (`while`) `n` for maior do que 0, siga exibindo o valor de `n` e diminuindo 1 do valor de `n`. Quando chegar a 0, exiba a palavra Fogo!"

Mais formalmente, aqui está o fluxo de execução para um comando `while`:

1. Teste a condição, resultando 0 ou 1.
2. Se a condição for falsa (0), saia do comando `while` e continue a execução a partir do próximo comando.
3. Se a condição for verdadeira (1), execute cada um dos comandos dentro do corpo e volte ao passo 1.

O corpo consiste de todos os comandos abaixo do cabeçalho, com a mesma endentação.

Este tipo de fluxo é chamado de um **loop** (ou laço) porque o terceiro passo cria um "loop" ou um laço de volta ao topo. Note que se a condição for falsa na primeira vez que entrarmos no loop, os comandos dentro do loop jamais serão executados.

O corpo do loop poderia alterar o valor de uma ou mais variáveis de modo que eventualmente a condição se torne falsa e o loop termine. Se não for assim, o loop se repetirá para sempre, o que é chamado de um **loop infinito**. Uma fonte de diversão sem fim para os cientistas da computação é a observação de que as instruções da embalagem de shampoo, "Lave, enxágüe, repita" é um loop infinito.

No caso de `contagemRegressiva`, podemos provar que o loop terminará porque sabemos que o valor de `n` é finito, e podemos ver que o valor de `n` diminui dentro de cada repetição (iteração) do loop, então, eventualmente chegaremos ao 0. Em outros casos, isto não é tão simples de afirmar:

```
def sequencia(n):
    while n != 1:
        print n,
        if n%2 == 0:                # n é par
            n = n/2
        else:                        # n é ímpar
            n = n*3+1
```

A condição para este loop é  $n \neq 1$ , então o loop vai continuar até que  $n$  seja 1, o que tornará a condição falsa.

Dentro de cada repetição (iteração) do loop, o programa gera o valor de  $n$  e então checka se ele é par ou impar. Se ele for par, o valor de  $n$  é dividido por 2. Se ele for impar, o valor é substituído por  $n*3+1$ . Por exemplo, se o valor inicial (o argumento passado para seqüência) for 3, a seqüência resultante será 3, 10, 5, 16, 8, 4, 2, 1.

Já que  $n$  às vezes aumenta e às vezes diminui, não existe uma prova óbvia de que  $n$  jamais venha a alcançar 1, ou de que o programa termine. Para alguns valores particulares de  $n$ , podemos provar o término. Por exemplo, se o valor inicial for uma potência de dois, então o valor de  $n$  será par dentro de cada repetição (iteração) do loop até que alcance 1. O exemplo anterior termina com uma dessas seqüências começando em 16.

Valores específicos à parte, A questão interessante é se há como provarmos que este programa termina para todos os valores de  $n$ . Até hoje, ninguém foi capaz de provar que sim ou que não!

Como um exercício, reescreva a função `nLinhas` da seção 4.9 usando iteração em vez de recursão.

```
1.0 0.0
2.0 0.69314718056
3.0 1.09861228867
4.0 1.38629436112
5.0 1.60943791243
6.0 1.79175946923
7.0 1.94591014906
8.0 2.07944154168
9.0 2.19722457734
```

Se estes valores parecem odd, lembre-se que a função `log` usa a base  $e$ . Já que potências de dois são tão importantes em ciência da computação, nós frequentemente temos que achar logaritmos referentes à base 2. Para fazermos isso, podemos usar a seguinte fórmula:

$$\log_2 x = \frac{\log_e x}{\log_e 2}$$

Alterando o comando de saída para:

```
print x, '\t', math.log(x)/math.log(2.0)
```

o que resultará em:

```
1.0 0.0
2.0 1.0
3.0 1.58496250072
4.0 2.0
5.0 2.32192809489
6.0 2.58496250072
7.0 2.80735492206
8.0 3.0
9.0 3.16992500144
```

Podemos ver que 1, 2, 4 e 8 são potências de dois porque seus logaritmos na base 2 são números redondos. Se precisássemos encontrar os logaritmos de outras potências de dois, poderíamos modificar o programa deste modo:

```
x = 1.0
while x < 100.0:
    print x, '\t', math.log(x)/math.log(2.0)
    x = x * 2.0
```

Agora, em vez de somar algo a  $x$  a cada iteração do loop, o que resulta numa seqüência aritmética, nós multiplicamos  $x$  por algo, resultando numa seqüência geométrica. O resultado é:

```
1.0 0.0
2.0 1.0
4.0 2.0
8.0 3.0
16.0 4.0
32.0 5.0
64.0 6.0
```

Por causa do caractere de tabulação entre as colunas, a posição da segunda coluna não depende do número de dígitos na primeira coluna.

Tabelas de logaritmos podem não ser mais úteis, mas para cientistas da computação, conhecer as potências de dois é!

Como um exercício, modifique este programa de modo que ele produza as potências de dois acima de 65.535 (ou seja, 2<sup>16</sup>). Imprima e memorize-as.

O caractere de barra invertida em `\t` indica o início de uma seqüência de escape. Seqüências de escape são usadas para representar caracteres invisíveis como de tabulação e de nova linha. A seqüência `\n` representa uma nova linha.

### 6.3 Tabelas

Uma das coisas para qual os loops são bons é para gerar dados tabulares. Antes que os computadores estivessem readily disponíveis, as pessoas tinham que calcular logaritmos, senos, cossenos e outras funções matemáticas à mão. Para tornar isto mais fácil, os livros de matemática continham longas tabelas listando os valores destas funções. Criar as tabelas era demorado e entediante, e elas tendiam a ser cheias de erros.

Quando os computadores entraram em cena, uma das reações iniciais foi "Isto é ótimo! Podemos usar computadores para gerar as tabelas, assim não haverá erros." Isto veio a se tornar verdade (na maioria das vezes) mas shortsighted. Rapidamente, porém, computadores e calculadoras tornaram-se tão pervasivos que as tabelas ficaram obsoletas.

Bem, quase. Para algumas operações, os computadores usam tabelas de valores para conseguir uma resposta aproximada e então realizar cálculos para melhorar a aproximação. Em alguns casos, têm havido erros nas tabelas underlying, o caso mais famoso sendo o da tabela usada pelo processador Pentium da Intel para executar a divisão em ponto-flutuante.

Embora uma tabela de logaritmos não seja mais tão útil quanto já foi um dia, ela ainda dá um bom exemplo de iteração. O seguinte programa gera uma seqüência de valores na coluna da esquerda e seus respectivos logaritmos na coluna da direita:

```
x = 1.0
while x < 10.0:
    print x, '\t', math.log(x)
    x = x + 1.0
```

A string `\t` representa um caracter de tabulação.

Conforme caracteres e strings vão sendo mostrados na tela, um ponteiro invisível chamado **cursor** marca aonde aparecerá o próximo caractere. Depois de um comando `print`, o cursor normalmente vai para o início de uma nova linha.

O caractere de tabulação desloca o cursor para a direita até que ele encontre uma das marcas de tabulação. Tabulação é útil para fazer colunas de texto line up, como na saída do programa anterior:



Uma seqüência de escape pode aparecer em qualquer lugar em uma string; no exemplo, a seqüência de escape de tabulação é a única coisa dentro da string.

Como você acha que se representa uma barra invertida em uma string?

Como um exercício, escreva um única string que

```
produza
esta
saída.
```

### 6.4 Tabelas de duas dimensões (ou bi-dimensionais)

Uma tabela de duas dimensões é uma tabela em que você lê o valor na interseção entre uma linha e uma coluna. Uma tabela de multiplicação é um bom exemplo. Digamos que você queira imprimir uma tabela de multiplicação de 1 a 6.

Uma boa maneira de começar é escrever um loop que imprima os múltiplos de 2, todos em uma linha:

```
i = 1
while i <= 6:
    print 2*i, ' ',
    i = i + 1
print
```

A primeira linha inicializa a variável chamada *i*, a qual age como um contador ou **variável de controle do loop**. Conforme o loop é executado, o valor de *i* é incrementado de 1 a 6. Quando *i* for 7, o loop termina. A cada repetição (iteração) do loop, é mostrado o valor de  $2*i$ , seguido de três espaços.

De novo, a vírgula no comando print suprime a nova linha. Depois que o loop se completa, o segundo comando print inicia uma nova linha.

A saída do programa é:

```
2    4    6    8    10   12
```

Até aqui, tudo bem. O próximo passo é **encapsular** e **generalizar**.

### 6.5 Encapsulamento e generalização

Encapsulamento é o processo de wrapping um pedaço de código em uma função, permitindo que você tire vantagem de todas as coisas para as quais as funções são boas. Você já viu dois exemplos de encapsulamento: `imprimeParidade` na seção 4.5; e `eDivisivel` na seção 5.4

Generalização significa tomar algo que é específico, tal como imprimir os múltiplos de 2, e torná-lo mais geral, tal como imprimir os múltiplos de qualquer inteiro.

Esta função encapsula o loop anterior e generaliza-o para imprimir múltiplos de *n*:

```
def imprimeMultiplos(n):
    i = 1
    while i <= 6:
        print n*i, '\t ',
        i = i + 1
    print
```

Para encapsular, tudo o que tivemos que fazer foi adicionar a primeira linha, que declara o nome de uma função e sua lista de parâmetros. Para generalizar, tudo o que tivemos que fazer foi substituir o valor 2 pelo parâmetro *n*.

Se chamarmos esta função com o argumento 2, teremos a mesma saída que antes. Com o argumento 3, a saída é:

```
3    6    9    12   15   18
```

Com o argumento 4, a saída é:

```
4    8    12   16   20   24
```

Agora você provavelmente pode adivinhar como imprimir uma tabela de multiplicação - chamando `imprimeMultiplos` repetidamente com argumentos diferentes. De fato, podemos usar um outro loop:

```
i = 1
while i <= 6:
    imprimeMultiplos(i)
    i = i + 1
```

Note o quanto este loop é parecido com aquele dentro de `imprimeMultiplos`. Tudo o que fiz foi substituir o comando `print` pela chamada à função.

A saída deste programa é uma tabela de multiplicação:

```
1    2    3    4    5    6
2    4    6    8    10   12
3    6    9    12   15   18
4    8    12   16   20   24
5    10   15   20   25   30
6    12   18   24   30   36
```

### 6.6 Mais encapsulamento

Para demonstrar de novo o encapsulamento, vamos pegar o código do final da seção 6.5 e acondicioná-lo, envolvê-lo em uma função:

```
def imprimeTabMult():
    i = 1
    while i <= 6:
        imprimeMultiplos(i)
        i = i + 1
```

Este processo é um **plano de desenvolvimento** comum. Nós desenvolvemos código escrevendo linhas de código fora de qualquer função, ou digitando-as no interpretador. Quando temos o código funcionando, extraímos ele e o embalamos em uma função.

Este plano de desenvolvimento é particularmente útil se você não sabe, quando você começa a escrever, como dividir o programa em funções. Esta técnica permite a você projetar enquanto desenvolve.

### 6.7 Variáveis locais

Você pode estar pensando como podemos utilizar a mesma variável, *i*, em ambos, `imprimeMultiplos` e `imprimeTabMult`. Isto não causaria problemas quando uma das funções mudasse o valor da variável?

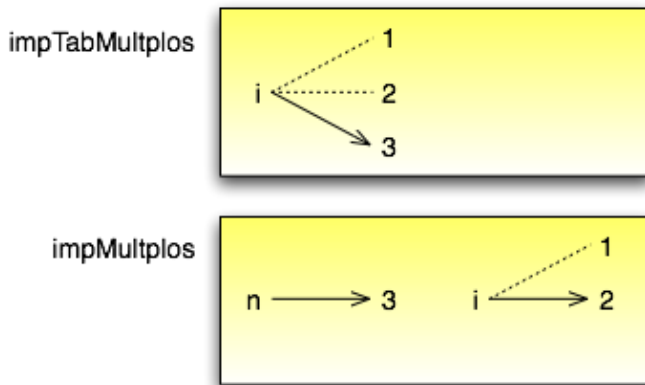
A resposta é não, porque o *i* em `imprimeMultiplos` e o *i* em `imprimeTabMult` não são a mesma variável.

Variáveis criadas dentro de uma definição de função são locais; você não pode acessar uma variável local de fora da função em que ela "mora". Isto significa que você é livre para ter múltiplas variáveis com o mesmo nome, desde que elas não estejam dentro da mesma função.

O diagrama de pilha para este programa mostra

## Como pensar como um cientista da Computação usando Python

que duas variáveis chamadas *i* não são a mesma variável. Elas podem se referir a valores diferentes e alterar o valor de uma não afeta à outra.



O valor de *i* em `imprimeTabMult` vai de 1 a 6. No diagrama, *i* agora é 3. Na próxima iteração do loop *i* será 4. A cada iteração do loop, `imprimeTabMult` chama `imprimeMultiplos` com o valor corrente de *i* como argumento. O valor é atribuído ao parâmetro *n*.

Dentro de `imprimeMultiplos`, o valor de *i* vai de 1 a 6. No diagrama, *i* agora é 2. Mudar esta variável não tem efeito sobre o valor de *i* em `imprimeTabMult`.

É comum e perfeitamente legal ter variáveis locais diferentes com o mesmo nome. Em particular, nomes como *i* e *j* são muito usados para variáveis de controle de loop. Se você evitar utilizá-los em uma função só porque você já os usou em outro lugar, você provavelmente tornará seu programa mais difícil de ler.

### 6.8 Mais generalização

Como um outro exemplo de generalização, imagine que você precise de um programa que possa imprimir uma tabela de multiplicação de qualquer tamanho, não apenas uma tabela de seis por seis. Você poderia adicionar um parâmetro a `imprimeTabMult`:

```
def imprimeTabMult(altura):
    i = 1
    while i <= altura:
        imprimeMultiplos(i)
        i = i + 1
```

Nós substituímos o valor 6 pelo parâmetro *altura*. Se chamarmos `imprimeTabMult` com o argumento 7, ela mostra:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

Isto é bom, exceto que nós provavelmente quereríamos que a tabela fosse quadrada - com o mesmo número de linhas e colunas. Para fazer isso, adicionamos outro parâmetro a `imprimeMultiplos` para especificar quantas colunas a tabela deveria ter.

Só para confundir, chamamos este novo parâmetro de *altura*, demonstrando que diferentes funções podem ter parâmetros com o mesmo nome (como acontece com as variáveis locais). Aqui está o programa completo:

```
def imprimeMultiplos(n, altura):
    i = 1
    while i <= altura:
        print n*i, 't',
        i = i + 1
    print

def imprimeTabMult(altura):
    i = 1
    while i <= altura:
        imprimeMultiplos(i, altura)
        i = i + 1
```

Note que quando adicionamos um novo parâmetro, temos que mudar a primeira linha da função (o cabeçalho da função), e nós também temos que mudar o lugar de onde a função é chamada em `imprimeTabMult`.

Como esperado, este programa gera uma tabela quadrada de sete por sete:

1	2	3	4	5	6	7
2	4	6	8	10	12	14
3	6	9	12	15	18	21
4	8	12	16	20	24	28
5	10	15	20	25	30	35
6	12	18	24	30	36	42
7	14	21	28	35	42	49

Quando você generaliza uma função apropriadamente, você muitas vezes tem um programa com capacidades que você não planejou. Por exemplo, você pode ter notado que, porque  $ab = ba$ , todas as entradas na tabela aparecem duas vezes. Você poderia economizar tinta imprimindo somente a metade da tabela. Para fazer isso, você tem que mudar apenas uma linha em `imprimeTabMult`. Mude:

```
imprimeTabMult(i, altura)
para:
```

```
imprimeTabMult(i, i)
```

e você terá:

1						
2	4					
3	6	9				
4	8	12	16			
5	10	15	20	25		
6	12	18	24	30	36	
7	14	21	28	35	42	49

Como um exercício, trace a execução desta versão de `imprimeTabMult` e explique como ela funciona.

### 6.9 Funções

- Há pouco tempo mencionamos "todas as coisas para as quais as funções são boas." Agora, você pode estar pensando que coisas exatamente são estas. Aqui estão algumas delas:
- Dar um nome para uma seqüência de comandos torna seu programa mais fácil de ler e de depurar.
- Dividir um programa longo em funções permite que você separe partes do programa, depure-as isoladamente, e então as componha em um todo.
- Funções facilitam tanto recursão quanto iteração.
- Funções bem projetadas são freqüentemente úteis para muitos programas. Uma vez que você escreva e depure uma, você pode reutilizá-la.

6.10 Glossário

<b>reatribuição (multiple assignment <sup>1</sup>)</b>	quando mais de um valor é atribuído a mesma variável durante a execução do programa.
<b>iteração (iteration)</b>	execução repetida de um conjunto de comandos/instruções (statements) usando uma chamada recursiva de função ou um laço (loop).
<b>laço (loop)</b>	um comando/instrução ou conjunto de comandos/instruções que executam repetidamente até que uma condição de interrupção seja atingida.
<b>laço infinito (infinite loop)</b>	um laço em que a condição de interrupção nunca será atingida.
<b>corpo (body)</b>	o conjunto de comandos/instruções que pertencem a um laço.
<b>variável de laço (loop variable)</b>	uma variável usada como parte da condição de interrupção do laço.
<b>tabulação (tab)</b>	um carácter especial que faz com que o cursor mova-se para a próxima parada estabelecida de tabulação na linha atual.
<b>nova-linha (newline)</b>	um carácter especial que faz com que o cursor mova-se para o início da próxima linha.
<b>cursor (cursor)</b>	um marcador invisível que determina onde o próximo carácter vai ser impresso.
<b>sequência de escape (escape sequence)</b>	um carácter de escape (\) seguido por um ou mais caracteres imprimíveis, usados para definir um carácter não imprimível.
<b>encapsular (encapsulate)</b>	quando um programa grande e complexo é dividido em componentes (como funções) e estes são isolados um do outro (pelo uso de variáveis locais, por exemplo).
<b>generalizar (generalize)</b>	quando algo que é desnecessariamente específico (como um valor constante) é substituído por algo apropriadamente geral (como uma variável ou um parâmetro). Generalizações dão maior versatilidade ao código, maior possibilidade de reuso, e em algumas situações até mesmo maior facilidade para escrevê-lo.
<b>plano de desenvolvimento (development plan)</b>	um processo definido para desenvolvimento de um programa. Neste capítulo, nós demonstramos um estilo de desenvolvimento baseado em escrever código para executar tarefas simples e específicas, usando encapsulamento e generalização.

<sup>1</sup> N.T.: O termo multiple assignment (ou atribuição múltipla) é usado com mais frequência para descrever a sintaxe  $a = b = c$ . Por este motivo optamos pelo termo reatribuição no contexto da seção 6.1 desse capítulo.

## Capítulo 7: Strings

### 7.1 Um tipo de dado composto

Até aqui, vimos três diferentes tipos de dado: int, float e string. Strings são qualitativamente diferentes dos outros dois tipos porque são feitas de pedaços menores - caracteres.

Tipos que consistem de pedaços menores são chamados **tipos de dados compostos**. Dependendo do que estejamos fazendo, pode ser que precisemos tratar um tipo de dado composto como uma coisa única, ou pode ser que queiramos acessar suas partes. Esta ambigüidade é útil.

O operador colchete seleciona um único caractere de uma string:

```
>>> fruta = "banana"
>>> letra = fruta[1]
>>> print letra
```

A expressão `fruta[1]` seleciona o caractere número 1 de `fruta`. A variável `letra` referencia ou refere-se ao resultado da expressão. Quando exibimos `letra`, temos uma surpresa:

```
a
```

A primeira letra de "banana" não é a. A menos que você seja um cientista da computação. Neste caso, você deve entender a expressão dentro dos colchetes como um deslocamento (offset,) a partir do começo da string, e o deslocamento da primeira letra é zero. Assim, `b` é a 0ª ("zero-ésima") letra de "banana", `a` é a 1ª ("um-ésima", diferente de primeira), e `n` é a 2ª ("dois-ésima", diferente de segunda) letra.

Para pegar a primeira letra de uma string, você simplesmente põe `0`, ou qualquer expressão que resulte o valor `0`, dentro dos colchetes:

```
>>> letra = fruta[0]
>>> print letra
b
```

A expressão entre colchetes é chamada de **índice**. Um índice especifica um membro de um conjunto ordenado, neste caso o conjunto de caracteres da string. O índice indica aquele membro que você quer, daí seu nome. Ele pode ser qualquer expressão inteira.

### 7.2 Comprimento

A função `len` retorna o número de caracteres de uma string:

```
>>> fruta = "banana"
>>> len(fruta)
6
```

Para pegar a última letra de uma string, você pode ficar tentado a fazer alguma coisa assim:

```
comprimento = len(fruta)
ultima = fruta[comprimento] # ERRO!
```

Não vai funcionar. Isto vai causar o seguinte erro em tempo de execução (runtime error): `IndexError: string index out of range`. (ErroDeÍndice: índice da string fora do intervalo). A razão é que não existe 6ª letra em "banana". Já que começamos a contar do zero, as seis letras são numeradas de 0 a 5. Para pegar o último caractere, temos que subtrair 1

de comprimento:

```
comprimento = len(fruta)
ultima = fruta[comprimento-1]
```

Como alternativa, podemos usar índices negativos, os quais contam de trás pra frente os elementos da string. A expressão `fruta[-1]` resulta a última letra, `fruta[-2]` resulta a penúltima (a segunda de trás para frente), e assim por diante.

### 7.3 Travessia e o loop for

Várias computações envolvem o processamento de uma string um caractere de cada vez. Muitas vezes elas começam com o primeiro, selecionam um de cada vez, fazem alguma coisa com ele, e continuam até o fim. Este padrão de processamento é chamado uma **travessia** (traversal, com a idéia de "percorrimto"). Uma maneira de codificar uma travessia é com um comando `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print letra
    indice = indice + 1
```

Este loop percorre a string e exhibe cada letra em sua própria linha. A condição do loop é `indice < len(fruta)`, assim, quando índice é igual ao comprimento da string, a condição se torna falsa, e o corpo do loop não é executado. O último caractere acessado é aquele com o índice `len(fruta)-1`, que vem a ser o último caractere da string.

Como um exercício, escreva uma função que tome uma string como argumento e devolva suas letras de trás para frente, uma por linha.

Usar um índice para percorrer um conjunto de valores é tão comum que Python oferece uma sintaxe alternativa simplificada - o loop `for`:

```
for char in fruta:
    print char
```

A cada vez através do loop, o próximo caractere da string é atribuído à variável `char`. O loop continua até que não reste mais caracteres.

O exemplo seguinte mostra como usar concatenação e um loop `for` para gerar uma série abecedário. "Abecedário" se refere a uma série ou lista na qual os elementos aparecem em ordem alfabética. Por exemplo, no livro de Robert McCloskey's *Make Way for Ducklings*, os nomes dos "ducklings" são Jack, Kack, Lack, Mack, Nack, Ouack, Pack e Quack. O loop seguinte, produz como saída aqueles nomes, em ordem:

```
prefixos = "JKLMNOPQ"
sufixo = "ack"

for letra in prefixos:
    print letra + sufixo
```

A saída deste programa é:

```
Jack
Kack
```

```
Lack
Mack
Nack
Oack
Pack
Qack
```

Naturalmente, esta saída não está cem por cento certa porque "Ouack" e "Quack" estão escritos de maneira errada.

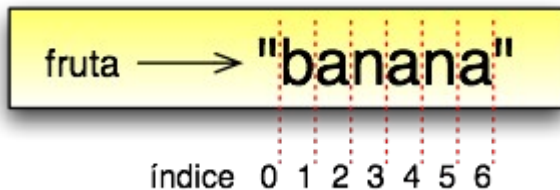
Como um exercício, modifique o programa para corrigir este erro.

### 7.4 Fatias de strings

Um segmento de uma string é chamado de uma fatia. Selecionar uma fatia é similar a selecionar um caractere:

```
>>> s = "Pedro, Paulo e Maria"
>>> print s[0:5]
Pedro
>>> print s[7:12]
Paulo
>>> print s[16:21]
Maria
```

O operador [n:m] retorna a parte da string do "n-ésimo" caractere ao "m-ésimo" caractere, incluindo o primeiro mas excluindo o último. Este comportamento não é intuitivo; ele faz mais sentido se você imaginar os índices apontando para os intervalos entre os caracteres, como no seguinte diagrama:



Se você omitir o primeiro índice (antes dos dois pontos ":"), a fatia começa do início da string. Se você omitir o segundo índice, a fatia vai até o final da string. Assim:

```
>>> fruta = "banana"
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

O que você acha de s[:] significa?

### 7.5 Comparação de strings

O operador de comparação funciona com strings. Para ver se duas strings são iguais:

```
if palavra == "banana":
    print "Sim, nós não temos bananas!"
```

Outras operações de comparação são úteis para colocar palavras em ordem alfabética:

```
if palavra < "banana":
    print "Sua palavra," + palavra + ", vem antes de \
banana."
elif palavra > "banana":
    print "Sua palavra," + palavra + ", vem depois \
de banana."
```

```
else:
    print "Sim, nós não temos bananas!"
```

Entretanto, você deve atentar para o fato de que Python não manipula letras maiúsculas e minúsculas da mesma maneira que as pessoas o fazem. Todas as letras maiúsculas vêm antes das minúsculas. Como resultado:

```
Sua palavra, Zebra, vem antes de banana.
```

Uma maneira comum de resolver este problema é converter as strings para um formato padrão, seja todas minúsculas, ou todas maiúsculas, antes de realizar a comparação. Um problema mais difícil é fazer o programa perceber que zebras não são frutas.

### 7.6 Strings são imutáveis

É tentador usar o operador [] no lado esquerdo de uma expressão de atribuição, com a intenção de alterar um caractere em uma string. Por exemplo:

```
saudacao = "Alô, mundo!"
saudacao[0] = 'E' # ERRO!
print saudacao
```

Em vez de produzir a saída Elô, Mundo!, este código produz o erro em tempo de execução (runtime error): TypeError: object doesn't support item assignment (ErroDeTipo: objeto não dá suporte à atribuição de item.)

Strings são imutáveis, o que significa que você não pode mudar uma string que já existe. O melhor que você pode fazer é criar uma nova string que seja uma variação da original:

```
saudacao = "Alô, mundo!"
novaSaudacao = 'E' + saudacao[1:]
print novaSaudacao
```

A solução aqui é concatenar uma nova primeira letra com uma fatia de saudação. Esta operação não tem nenhum efeito sobre a string original.

### 7.7 Uma função find (encontrar)

O que faz a seguinte função?:

```
def find(str, ch):
    indice = 0
    while indice < len(str):
        if str[indice] == ch:
            return indice
        indice = indice + 1
    return -1
```

Num certo sentido, find (encontrar) é o oposto do operador []. Em vez de pegar um índice e extrair o caractere correspondente, ela pega um caractere e encontra (finds) em qual índice aquele caractere aparece. Se o caractere não é encontrado, a função retorna -1.

Este é o primeiro exemplo que vemos de uma instrução return dentro de um loop. Se str[indice] == ch, a função retorna imediatamente, abandonando o loop prematuramente.

Se o caractere não aparece na string, então o programa sai do loop normalmente e retorna -1.

Este padrão de computação é às vezes chamado de travessia "eureka", porque tão logo ele encontra (find) o que está procurando, ele pode gritar "Eureka!" e parar de procurar.

Como um exercício, modifique a função find



(encontrar) de modo que ela receba um terceiro parâmetro, o índice da string por onde ela deve começar sua procura.

### 7.8 Iterando e contando

O programa seguinte conta o número e vezes que a letra a aparece em uma string:

```
fruta = "banana"
contador = 0
for letra in fruta:
    if letra == 'a':
        contador = contador + 1
print contador
```

Este programa demonstra um outro padrão de computação chamado de **contador**. A variável contador é inicializada em 0 e então incrementada cada vez que um a é encontrado. (**Incrementar** é o mesmo que aumentar em um; é o oposto de **decrementar**, e não tem relação com excremento, que é um substantivo.) Quando se sai do loop, contador guarda o resultado - o número total de a's.

Como um exercício, encapsule este código em uma função chamada contaLetras, e generalize-a de modo que possa aceitar uma string e uma letra como parâmetros.

Como um segundo exercício, reescreva esta função de modo que em vez de percorrer a string, ela use a versão com três parâmetros de find (encontrar) da seção anterior.

### 7.9 O módulo string

O módulo string contém funções úteis que manipulam strings. Conforme é usual, nós temos que importar o módulo antes que possamos utilizá-lo:

```
>>> import string
```

O módulo string inclui uma função chamada find (encontrar) que faz a mesma coisa que a função que escrevemos. Para chamá-la, temos que especificar o nome do módulo e o nome da função usando a notação de ponto.:

```
>>> fruta = "banana"
>>> indice = string.find(fruta, "a")
>>> print indice
1
```

Este exemplo demonstra um dos benefícios dos módulos - eles ajudam a evitar colisões entre nomes de funções nativas e nomes de funções definidas pelo usuário. Usando a notação de ponto podemos especificar que versão de find (encontrar) nós queremos.

De fato, string.find é mais generalizada que a nossa versão. Primeiramente, ela pode encontrar substrings, não apenas caracteres:

```
>>> string.find("banana", "na")
2
```

Além disso, ela recebe um argumento adicional que especifica o índice pelo qual ela deve começar sua procura:

```
>>> string.find("banana", "na", 3)
4
```

Ou ela pode receber dois argumentos adicionais que especificam o intervalo de índices:

```
>>> string.find("bob", "b", 1, 2)
-1
```

Neste exemplo, a busca falha porque a letra b não aparece no intervalo entre 1 e 2 (não incluindo o 2) do índice.

### 7.10 Classificação de caracteres

Muitas vezes é útil examinar um caractere e testar se ele é maiúsculo ou minúsculo, ou se ele é um caractere ou um dígito. O módulo string oferece várias constantes que são úteis para esses propósitos.

A string string.lowercase contém todas as letras que o sistema considera como sendo minúsculas. Similarmente, string.uppercase contém todas as letras maiúsculas. Tente o seguinte e veja o que você obtém:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

Nós podemos usar essas constantes e find (encontrar) para classificar caracteres. Por exemplo, se find(lowercase, ch) retorna um valor outro que não -1, então ch deve ser minúsculo:

```
def eMinusculo(ch):
    return string.find(string.lowercase, ch) != -1
```

Como uma alternativa, podemos tirar vantagem do operador in, que determina se um caractere aparece em uma string:

```
def eMinusculo(ch):
    return ch in string.lowercase
```

Ainda, como uma outra alternativa, podemos usar o operador de comparação:

```
def eMinusculo(ch):
    return 'a' <= ch <= 'z'
```

Se ch estiver entre a e z, ele deve ser uma letra minúscula.

Como um exercício, discuta que versão de eMinusculo você acha que será a mais rápida. Você pode pensar em outras razões além da velocidade para preferir uma em vez de outra?

Outra constante definida no módulo string pode te surpreender quando você executar um print sobre ela:

```
>>> print string.whitespace
```

Caracteres de **espaçamento** (ou espaços em branco) movem o cursor sem "imprimir" qualquer coisa. Eles criam os espaços em branco entre os caracteres visíveis (pelo menos numa folha de papel branco). A string constante string.whitespace contém todos os caracteres de espaçamento, incluindo espaço, tabulação (\t) e nova linha (\n).

Existem outras funções úteis no módulo string, mas este livro não pretende ser um manual de referência. Por outro lado, Python Library Reference é exatamente isto. Em meio a uma abundante documentação, ele está disponível no site da web do Python, [www.python.org](http://www.python.org).

### 7.11 Glossário

<b>tipo de dado composto (compound data type)</b>	Um tipo de dado no qual o valor consiste de componentes, ou elementos, que são eles mesmos valores.
<b>travessia (traverse)</b>	Iterar através dos elementos de um conjunto, realizando uma operação similar em cada um

	deles.
<b>índice (index)</b>	Uma variável ou valor usados para selecionar um membro de um conjunto ordenado, como um caractere em uma string.
<b>fatia (slice)</b>	Uma parte de uma string especificada por um intervalo de índices.
<b>mutável (mutable)</b>	Um tipo de dado composto a cujos elementos podem ser atribuídos novos valores.
<b>contador (counter)</b>	Uma variável utilizada para contar alguma coisa, usualmente inicializada em zero e então incrementada.
<b>incrementar (increment)</b>	aumentar o valor de uma variável em 1.
<b>decrementar (decrement)</b>	diminuir o valor de uma variável em 1.
<b>espaçamento (whitespace)</b>	Qualquer um dos caracteres que move o cursor sem imprimir caracteres visíveis. A constante <code>string.whitespace</code> contém todos os caracteres de espaçamento.

## 7.11 Glossário2

<b>tipo de dado composto (compound data type)</b>	Um tipo de dado em que os valores são compostos de componentes, ou elementos, que podem ser tratados como valores separados.
<b>atravessar (traverse)</b>	Iterar através dos elementos definidos, executando uma operação similar em cada.
<b>índice (index)</b>	Uma variável ou valor usado para selecionar um membro de uma definição ordenada, como um caractere de uma string.
<b>fatia (slice)</b>	Uma parte da string especificada por um intervalo de índice.
<b>mutável (mutable)</b>	Um tipo de dado composto do qual elementos podem atribuir novos valores.
<b>contador (counter)</b>	Uma variável usada para contar alguma coisa, geralmente iniciada em zero e incrementada.
<b>incremento (increment)</b>	Para aumentar o valor da variável.
<b>decremento (decrement)</b>	Para diminuir o valor da variável.
<b>espaço em branco (whitespace)</b>	Qualquer caractere que move o cursor sem imprimir caracteres visíveis. A constante <code>string.whitespace</code> contém todos os caracteres de espaço em branco.

## Capítulo 8: Listas

Uma lista é um conjunto ordenado de valores, onde cada valor é identificado por um índice. Os valores que compõem uma lista são chamados **elementos**. Listas são similares a strings, que são conjuntos ordenados de caracteres, com a diferença que os elementos de uma lista podem possuir qualquer tipo. Listas e strings XXX e outras coisas que se comportam como conjuntos ordenados XXX são chamados **seqüências**.

### 8.1 Valores da lista

Existem várias maneiras de criar uma nova lista; a mais simples é envolver os elementos em colchetes ([ e ]):

```
>>> [10, 20, 30, 40]
>>> ['spam', 'bungee', 'swallow']
```

O primeiro exemplo é uma lista de quatro inteiros. O segundo é uma lista de três strings. Os elementos de uma lista não necessitam ser do mesmo tipo. A lista a seguir contém uma string, um valor float, um valor inteiro, e mirabile dictu uma outra lista:

```
>>> ['alo', 2.0, 5, [10,20]]
```

Uma lista dentro de outra lista é dita estar **aninhada**.

Listas que contém inteiros consecutivos são comuns, então Python fornece uma maneira simples de criá-los:

```
>>> range(1,5)
[1, 2, 3, 4]
```

A função range pega dois argumentos e devolve uma lista que contém todos os inteiros do primeiro até o segundo, incluindo o primeiro mas não incluindo o segundo!

Existem outras formas de range. Com um argumento simples, ela cria uma lista que inicia em 0:

```
>>> range(10)
[0,1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se existe um terceiro argumento, ele especifica o espaço entre os valores sucessivos, que é chamado de tamanho do passo. Este exemplo conta de 1 até 10 em passos de 2:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Finalmente, existe uma lista especial que não contém elementos. Ela é chamada lista vazia, e sua notação é [].

Com todas estas formas de criar listas, seria decepcionante se não pudéssemos atribuir valores de listas a variáveis ou passar listas como parâmetros a funções. Felizmente, podemos.

```
>>> vocabulario = ['melhorar', 'castigar', \
'defenestrar']
>>> numeros = [17, 123]
>>> vazio = []
>>> print vocabulario, numeros, vazio
['melhorar', 'castigar', 'defenestrar'] [17, 123] []
```

### 8.2 Acessado elementos

A sintaxe para acessar os elementos de uma lista é a mesma que a sintaxe para acessar os caracteres de uma string XXX o operador colchete ([]). A expressão dentro dos colchetes especifica o índice. Lembre-se que os índices iniciam em 0:

```
>>> print numeros[0]
>>> numeros[1]= 5
```

O operador colchete pode aparecer em qualquer lugar em uma expressão. Quando ele aparece no lado esquerdo de uma atribuição, ele modifica um dos elementos em uma lista, de forma que o um-ésimo elemento de numeros, que era 123, é agora 5.

Qualquer expressão inteira pode ser utilizada como um índice:

```
>>> numeros[3-2]
5
>>> numeros[1.0]
TypeError: sequence index must be integer
```

Se você tentar ler ou escrever um elemento que não existe, você recebe um erro de tempo de execução (runtime error):

```
>>> numeros[2]=5
IndexError: list assignment index out of range
```

Se um índice possui um valor negativo, ele conta ao contrário a partir do final da lista:

```
>>> numeros[-1]
5
>>> numeros[-2]
17
>>> numeros[-3]
IndexError: list index out of range
```

numeros[-1] é o último elemento da lista, numeros[-2] é o penúltimo e numeros[-3] não existe.

É comum utilizar uma variável de laço como um índice da lista:

```
>>> cavaleiros = ['guerra', 'fome', 'peste', 'morte']
i = 0
while i < 4:
    print cavaleiros[i]
    i = i + 1
```

Este laço while conta de 0 até 4. Quando a variável do laço i é 4, a condição falha e o laço se encerra. Desta forma o corpo do laço é executado somente quando i é 0, 1, 2 e 3.

Em cada vez dentro do laço, a variável i é utilizada como um índice para a lista, exibindo o i-ésimo elemento. Este padrão de computação é chamado de **percurso na lista**.

### 8.3 Comprimento da lista

A função len devolve o comprimento de uma lista. É uma boa idéia utilizar este valor como o limite superior de um laço ao invés de uma constante. Desta forma, se o tamanho da lista

mudar, você não precisará ir através de todo o programa modificando todos os laços; eles funcionarão corretamente para qualquer tamanho de lista:

```
>>> cavaleiros = ['guerra', 'fome', 'peste', 'morte']
i = 0
while i < len(cavaleiros):
    print cavaleiros[i]
    i = i + 1
```

A última vez que o corpo do laço é executado,  $i$  é  $\text{len}(\text{cavaleiros}) - 1$ , que é o índice do último elemento. Quando  $i$  é igual a  $\text{len}(\text{cavaleiros})$ , a condição falha e o corpo não é executado, o que é uma boa coisa, porque  $\text{len}(\text{cavaleiros})$  não é um índice legal.

Embora uma lista possa conter uma outra lista, a lista aninhada ainda conta como um elemento simples. O comprimento desta lista é quatro:

```
>>> ['spam!', 1, ['Brie', 'Roquefort', \
'Pol lê Veq'], [1, 2 3]]
```

Como um exercício, escreva um laço que percorra a lista anterior e exiba o comprimento de cada elemento. O que acontece se você manda um inteiro para  $\text{len}$ ?

### 8.4 Membros de uma lista

$\text{in}$  é um operador lógico que testa se um elemento é membro de uma seqüência. Nós o utilizamos na Seção 7.10 com strings, mas ele também funciona com listas e outras seqüências:

```
>>> cavaleiros = ['guerra', 'fome', 'peste', 'morte']
>>> 'peste' in cavaleiros
True
>>> 'depravação' in cavaleiros
False
```

Uma vez que 'peste' é um membro da lista cavaleiros, o operador  $\text{in}$  devolve verdadeiro. Uma vez que depravação não está na lista,  $\text{in}$  devolve falso.

Podemos utilizar também o  $\text{not}$  em combinação com o  $\text{in}$  para testar se um elemento não é um membro de uma lista:

```
>>> ``depravação`` not in cavaleiros
True
```

### 8.5 Listas e laços for

O laço  $\text{for}$  que vimos na Seção 7.3 também funciona com listas. A sintaxe generalizada de um laço  $\text{for}$  é:

```
for VARIÁVEL in LISTA:
    CORPO
```

Esta declaração é equivalente a:

```
>>> i = 0
while i < len(LISTA):
    VARIABLE = LISTA[i]
    XXX BODY
    i = i + 1
```

O laço  $\text{for}$  é mais conciso porque podemos eliminar a variável do laço,  $i$ . Aqui está o laço anterior escrito com um laço  $\text{for}$ :

```
>>> for cavaleiro in cavaleiros:
    print cavaleiro
```

Quase se lê como Português: "For (para cada) cavaleiro  $\text{in}$  (na lista de) cavaleiros,  $\text{print}$  (imprima o nome do) cavaleiro."

Qualquer expressão de lista pode ser utilizada num laço  $\text{for}$ :

```
>>> for numero in range(20):
    if numero % 2 == 0:
        print numero

>>> for fruta in ["banana", "abacaxi", "laranja"]:
    print "Eu gosto de comer " + fruta + "s!"
```

O primeiro exemplo exibe todos os números pares entre zero e dezenove. O segundo exemplo expressa o entusiasmo por várias frutas.

### 8.6 Operações em listas

O operador  $+$  concatena listas:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarmente, o operador  $*$  repete uma lista um número dado de vezes:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

O primeiro exemplo repete  $[0]$  quatro vezes. O segundo exemplo repete a lista  $[1, 2, 3]$  três vezes.

### 8.7 Fatiamento de listas

A operação de fatiamento que vimos na Seção 7.4 também funciona sobre listas:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3]
['b', 'c']
>>> lista[:4]
['a', 'b', 'c', 'd']
>>> lista[3:]
['d', 'e', 'f']
>>> lista[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

### 8.8 Listas são mutáveis

Diferente das strings, as listas são mutáveis, o que significa que podemos modificar seus elementos. Utilizando o operador colchete no lado esquerdo de uma atribuição, podemos atualizar um de seus elementos:

```
>>> fruta = ["banana", "abacaxi", "laranja"]
>>> fruta[0] = "abacate"
>>> fruta[-1] = "tangerina"
>>> print fruta
['abacate', 'abacaxi', 'tangerina']
```

Com o operador de fatiamento podemos atualizar vários elementos de uma vez:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = ['x', 'y']
>>> print lista
['a', 'x', 'y', 'd', 'e', 'f']
```

Também podemos remover elementos de uma lista atribuindo a lista vazia a eles:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = []
>>> print lista
['a', 'd', 'e', 'f']
```

E podemos adicionar elementos a uma lista enfiando-os numa fatia vazia na posição desejada:

```
>>> lista = ['a', 'd', 'f']
>>> lista[1:1] = ['b', 'c']
>>> print lista
['a', 'b', 'c', 'd', 'f']
>>> lista[4:4] = ['e']
>>> print lista
['a', 'b', 'c', 'd', 'e', 'f']
```

### 8.9 Remoção em lista

Utilizando fatias para remover elementos pode ser complicado, e desta forma propenso a erro. Python fornece uma alternativa que é mais legível.

`del` remove um elemento de uma lista:

```
>>> a = ['um', 'dois', 'tres']
>>> del a[1]
>>> a
['um', 'tres']
```

Como você deveria esperar, `del` trata valores negativos e causa erros de tempo de execução se o índice estiver fora da faixa.

Você também pode utilizar uma faixa como um índice para `del`:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del lista[1:5]
>>> print lista
['a', 'f']
```

Como de costume, fatias selecionam todos os elementos até, mas não incluindo, o segundo índice.

### 8.10 Ojetos e valores

Se executamos estas declarações de atribuição:

```
>>> a = "banana"
>>> b = "banana"
```

sabemos que `a` e `b` se referem a uma string com as letras banana. Mas não podemos dizer se elas apontam para a mesma string.

Existem dois possíveis estados:



Em um caso, `a` e `b` se referem a duas coisas diferentes que possuem o mesmo valor. No segundo caso, elas

se referem à mesma coisa. Estas "coisas" possuem nomes - elas são chamadas **objetos**. Um objeto é algo ao qual uma variável pode se referenciar.

Todo objeto possui um **identificador** único, que podemos obter com a função `id`. Exibindo o identificador de `a` e `b`, podemos dizer se elas se referem ao mesmo objeto.

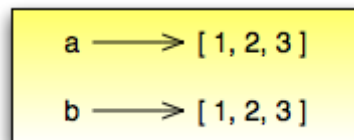
```
>>> id(a)
135044008
>>> id(b)
135044008
```

De fato, obtivemos o mesmo identificador duas vezes, o que significa que Python criou apenas uma string, e tanto `a` quanto `b` se referem a ela.

Interessantemente, listas se comportam de forma diferente. Quando criamos duas listas, obtemos dois objetos:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

Então o diagrama de estado fica assim:



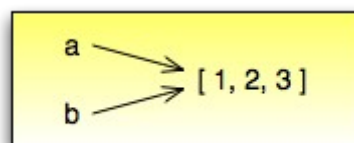
`a` e `b` possuem o mesmo valor mas não se referem ao mesmo objeto.

### 8.11 Apelidos

Uma vez que variáveis se referem a objetos, se atribuímos uma variável a uma outra, ambas as variáveis se referem ao mesmo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
```

Neste caso, o diagrama de estado se parece com isto:



Uma vez que a lista possui dois nomes diferentes, `a` e `b`, dizemos que ela está "apelidada" (aliased). Mudanças feitas em um apelido afetam o outro nome:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Embora este comportamento possa ser útil, ele é às vezes inesperado e indesejado. Em geral, é mais seguro evitar os apelidos quando você está trabalhando com objetos mutáveis. É claro, para objetos imutáveis, não há problema. É por isto que Python é livre para apelidar cadeias de caracteres quando vê uma oportunidade de economizar.



## 8.12 Clonando listas

Se queremos modificar uma lista e também manter uma cópia da original, precisamos ter condições de fazer uma cópia da própria lista, não apenas uma referência. Este processo é algumas vezes chamado clonagem, para evitar a ambigüidade da palavra "cópia".

A maneira mas fácil de clonar uma lista é utilizar o operador de fatia:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Pegar qualquer fatia de a cria uma nova lista. Neste caso acontece da fatia consistir da lista inteira.

Agora estamos livres para fazer alterações a b sem nos preocuparmos com a:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

Como exercício, desenhe um diagrama de estado para a e b antes e depois desta mudança.

## 8.13 Lista como parâmetro

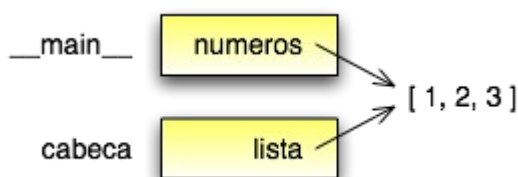
Passar uma lista como um argumento passa realmente uma referência à lista, não uma cópia da lista. Por exemplo, a função cabeça pega uma lista como parâmetro e devolve a cabeça da lista, ou seja, seu primeiro elemento:

```
>>> def cabeça(lista):
    return lista[0]
```

Eis como ela é utilizada:

```
>>> numeros = [1, 2, 3]
>>> cabeça(numeros)
1
```

O parâmetro lista e a variável numeros são apelidos para o mesmo objeto. O diagrama de estado se parece com isto:



Uma vez que o objeto é compartilhado pelos dois quadros, o desenhamos entre eles.

Se a função modifica um parâmetro da lista, a função chamadora vê a mudança. Por exemplo, removeCabeça remove o primeiro elemento da lista:

```
>>> def removecabeça(lista):
    del lista[0]
```

Aqui está a maneira como ela é utilizada:

```
>>> numeros = [1, 2, 3]
>>> removeCabeça(numeros)
>>> print numeros
[2, 3]
```

Se uma função devolve uma lista, ela devolve uma referência à lista. Por exemplo, cauda devolve uma lista

que contém todos menos o primeiro elemento de uma determinada lista:

```
>>> def cauda(lista):
    return lista[1:]
```

Aqui está a maneira como ela é utilizada:

```
>>> numeros = [1, 2, 3]
>>> resto = cauda(numeros)
>>> print resto
[2, 3]
```

Uma vez que o valor de retorno foi criado com o operador de fatia, ele é uma nova lista. A criação de resto, e qualquer alteração subsequente a resto, não tem efeito sobre numeros.

## 8.14 Lista aninhadas

Uma lista aninhada é uma lista que aparece como um elemento de uma outra lista. Nesta lista, o terceiro elemento é uma lista aninhada:

```
>>> lista = ["alo", 2.0, 5, [10, 20]]
```

Se exibimos lista[3], obtemos [10, 20]. Para extrairmos um elemento de uma lista aninhada, podemos agir em duas etapas:

```
>>> elem = lista[3]
>>> elem[0]
10
```

Ou podemos combiná-las:

```
>>> lista[3][1]
20
```

Os operadores colchete avaliam da esquerda para a direita, então a expressão pega o terceiro elemento de lista e extrai o primeiro elemento dela.

## 8.15 Matrizes

Listas aninhadas são freqüentemente utilizadas para representar matrizes. Por exemplo, a matriz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

poderia ser representada como:

```
>>> matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

matriz é uma lista com três elementos, onde cada elemento é uma linha da matriz. Podemos selecionar uma linha inteira da matriz da maneira habitual:

```
>>> matriz[1]
[4, 5, 6]
```

Ou podemos extrair um único elemento da matriz utilizando a forma de duplo índice:

```
>>> matriz[1][1]
5
```

O primeiro índice seleciona a linha, e o segundo índice seleciona a coluna. Embora esta maneira de representar matrizes seja comum, ela não é a única possibilidade. Uma pequena variação é utilizar uma lista de colunas ao invés de uma lista de linhas.

Mais adiante veremos uma alternativa mais radical utilizando um dicionário. ■ mesmo para qualquer string? Quando eles seriam diferentes?

### 8.16 Strings e listas

Duas das mais úteis funções no módulo string envolvem listas de strings. A função `split` (separar) quebra uma string em uma lista de palavras. Por padrão, qualquer número de caracteres espaço em branco é considerado um limite de uma palavra:

```
>>> import string
>>> poesia = "O orvalho no carvalho..."
>>> string.split(poesia)
['O', 'orvalho', 'no', 'carvalho...']
```

Um argumento opcional chamado um **delimitador** pode ser utilizado para especificar qual caracter utilizar como limites da palavra. O exemplo a seguir utiliza a string `va`:

```
>>> string.split(poesia, 'va')
['O or', 'lho no car', 'lho...']
```

Perceba que o delimitador não aparece na lista.

A função `join` (juntar) é o inverso de `split`. Ela pega uma lista de strings e concatena os elementos com um espaço entre cada par:

```
>>> lista = ['O', 'orvalho', 'no', 'carvalho...']
>>> string.join(lista)
'O orvalho no carvalho...'
```

Como `split`, `join` recebe um delimitador que é inserido entre os elementos:

```
>>> string.join(lista, '_')
'O_orvalho_no_carvalho...'
```

■ Como um exercício, descreva o relacionamento entre `string.join(string.split(poesia))` e `poesia`. Eles são o

### 8.17 Glossário

<b>lista (list)</b>	Uma coleção denominada de objetos, onde cada objeto é identificado por um índice.
<b>índice (index)</b>	Uma variável inteira ou valor que indica um elemento de uma lista.
<b>elemento (element)</b>	Um dos valores em uma lista(ou outra seqüência). O operador colchete seleciona elementos de uma lista.
<b>seqüência (sequence)</b>	Qualquer um dos tipos de dados que consiste de um conjunto ordenado de elementos, com cada elemento identificado por um índice.
<b>lista aninhada (nested list)</b>	Uma lista que é um elemento de uma outra lista.
<b>percurso na lista (list traversal)</b>	O acesso seqüencial de cada elemento em uma lista.
<b>objeto (object)</b>	Um coisa a qual uma variável pode se referir.
<b>apelidos (aliases)</b>	Múltiplas variáveis que contém referências ao mesmo objeto.
<b>clonar (clone)</b>	Criar um novo objeto que possui o mesmo valor de um objeto existente. Copiar a referência a um objeto cria um apelido (alias) mas não clona o objeto.
<b>delimitador (delimiter)</b>	Um caracter uma string utilizados para indicar onde uma string deveria ser dividida( <code>split</code> ).

## Capítulo 9: Tuplas

### 9.1 Mutabilidade e tuplas

Até agora, você tem visto dois tipos compostos: strings, que são compostos de caracteres; e listas, que são compostas de elementos de qualquer tipo. Uma das diferenças que notamos é que os elementos de uma lista podem ser modificados, mas os caracteres em uma string não. Em outras palavras, strings são **imutáveis** e listas são **mutáveis**.

Há um outro tipo em Python chamado **tupla** (tuple) que é similar a uma lista exceto por ele ser **imutável**. Sintaticamente, uma tupla é uma lista de valores separados por vírgulas:

```
>>> tupla = 'a', 'b', 'c', 'd', 'e'
```

Embora não seja necessário, é convencional colocar tuplas entre parênteses:

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
```

Para criar uma tupla com um único elemento, temos que incluir uma vírgula final:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Sem a vírgula, Python entende ('a') como uma string entre parênteses:

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

Questões de sintaxe de lado, as operações em tuplas são as mesmas operações das listas. O operador índice seleciona um elemento da tupla.

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
>>> tupla[0]
'a'
```

E o operador slice (fatia) seleciona uma "faixa" (range) de elementos.

```
>>> tupla[1:3]
('b', 'c')
```

Mas se tentarmos modificar um dos elementos de uma tupla, teremos um erro:

```
>>> tupla[0] = 'A'
TypeError: object doesn't support item assignment
```

Naturalmente, mesmo que não possamos modificar os elementos de uma tupla, podemos substituí-la por uma tupla diferente:

```
>>> tupla = ('A',) + tupla[1:]
>>> tupla
('A', 'b', 'c', 'd', 'e')
```

### 9.2 Atribuições de tupla

De vez em quando, é necessário trocar entre si os valores de duas variáveis. Com operações de atribuição convencionais, temos que utilizar uma variável temporária. Por exemplo, para fazer a troca entre a e b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Se você tiver que fazer isso com frequência, esta abordagem se torna incômoda. Python fornece uma forma de **atribuição de tupla** que resolve esse problema elegantemente:

```
>>> a, b = b, a
```

O lado esquerdo é uma tupla de variáveis; o lado direito é uma tupla de valores. Cada valor é atribuído à sua respectiva variável. Todas as expressões do lado direito são avaliadas antes de qualquer das atribuições. Esta característica torna as atribuições de tupla bastante versáteis.

Naturalmente, o número de variáveis na esquerda e o número de valores na direita deve ser igual:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

### 9.3 Tuplas como valores de retorno

Funções podem retornar tuplas como valor de retorno. Por Exemplo, nós poderíamos escrever uma função que troca dois parâmetros entre si:

```
def troca(x, y):
    return y, x
```

Então nós poderíamos atribuir o valor de retorno para uma tupla com duas variáveis:

```
a, b = troca(a, b)
```

Neste caso, não existe uma grande vantagem em fazer de troca (swap) uma função. De fato, existe um perigo em tentar encapsular troca, o qual é a tentação de cometer o seguinte erro:

```
def troca(x, y):
    # versao incorreta
    x, y = y, x
```

Se nós chamarmos esta função desta forma:

```
troca(a, b)
```

então a e x são apelidos para um mesmo valor. Mudar x dentro da função troca, faz com que x se referencie a um valor diferente, mas sem efeito sobre a dentro de `__main__`. Do mesmo modo, a mudança em y não tem efeito sobre b.

Esta função roda sem produzir uma mensagem de erro, mas ela não faz o que pretendemos. Este é um exemplo de um erro semântico.

Como exercício, desenhe um diagrama de estado pra esta função de modo que você possa ver porque ela não funciona.

### 9.4 Números aleatórios

A maioria dos programas de computador fazem a mesma coisa sempre que são executados, então, podemos dizer que eles são **determinísticos**. Determinismo em geral é uma coisa boa, se

nós esperamos que um cálculo dê sempre o mesmo resultado. Entretanto, para algumas aplicações queremos que o computador se torne imprevisível. Jogos são um exemplo óbvio, mas existem outros.

Fazer um programa realmente não-determinístico se mostra não ser tão fácil, mas existem maneiras de fazê-lo ao menos parecer não-determinístico. Uma dessas maneiras é gerar números aleatórios e usá-los para determinar o resultado de um programa. Python tem uma função nativa que gera números pseudo aleatórios, os quais não são verdadeiramente aleatórios no sentido matemático, mas para os nossos propósitos eles são.

O módulo `random` contém uma função chamada `random` que retorna um número em ponto flutuante (floating-point number) entre 0.0 e 1.0. Cada vez que você chama `random`, você recebe o próximo número de uma longa série. Para ver uma amostra, execute este loop:

```
import random

for i in range(10):
    x = random.random()
    print x
```

Para gerar um número aleatório entre 0.0 e um limite superior, digamos `superior`, multiplique `x` por `superior`.

Como exercício, gere um número aleatório entre 'inferior' e 'superior'.

Como exercício adicional, gere um número inteiro aleatório entre 'inferior' e 'superior', inclusive os dois extremos.

### 9.5 Lista de números aleatórios

O primeiro passo é gerar uma lista aleatória de valores. `listaAleatoria` pega um parâmetro inteiro e retorna uma lista de números aleatórios com o comprimento dado. Inicia-se com uma lista de `n` zeros. A cada iteração do loop, ele substitui um dos elementos por um número aleatório. O valor retornado é uma referência para a lista completa:

```
def listaAleatoria(n):
    s = [0] * n
    for i in range(n):
        s[i] = random.random()
    return s
```

Vamos realizar um teste desta função com uma lista de oito elementos. Para efeitos de depuração, é uma boa idéia começar com uma lista pequena.

```
>>> listaAleatoria(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

Os números gerados por `random` são supostamente uniformemente distribuídos, o que significa que cada valor tem uma probabilidade igual de acontecer.

Se nós dividirmos a faixa de valores possíveis em intervalos do mesmo tamanho, e contarmos o número de vezes que um determinado valor aleatório caiu em seu respectivo intervalo, nós devemos obter o mesmo número aproximado de valores em cada um dos intervalos.

Nós podemos testar esta teoria escrevendo um

programa que divida a faixa de valores em intervalos e conte o número de valores de cada intervalo.

### 9.6 Contando

Uma boa maneira de abordar problemas como esse é dividir o problema em subproblemas, e encontrar um subproblema que se enquadre em um padrão de solução computacional que você já tenha visto antes.

Neste caso, queremos percorrer uma lista de números e contar o número de vezes que valor se encaixa em um determinado intervalo. Isso soa familiar. Na Seção 7.8, nós escrevemos um programa que percorria uma string e contava o número de vezes que uma determinada letra aparecia.

Assim, podemos prosseguir copiando o programa original e adaptando-o para o problema atual. O programa original era:

```
contador = 0
for letra in fruta:
    if letra == 'a':
        contador = contador + 1
print contador
```

O primeiro passo é substituir `fruta` por lista e letra por número. Isso não muda o programa, apenas o ajusta para que ele se torne mais fácil de ler e entender.

O segundo passo é mudar o teste. Nós não estamos interessados em procurar letras. Nós queremos ver se número está entre inferior e superior.:

```
contador = 0
for numero in lista:
    if inferior < numero < superior:
        contador = contador + 1
print contador
```

O último passo é encapsular este código em uma função chamada `noIntervalo`. Os parâmetros são a lista e os valores inferior e superior:

```
def noIntervalo(lista, inferior, superior):
    contador = 0
    for numero in lista:
        if inferior < numero < superior:
            contador = contador + 1
    return contador
```

Através da cópia e da modificação de um programa existente, estamos aptos a escrever esta função rapidamente e economizar um bocadinho de tempo de depuração. Este plano de desenvolvimento é chamado de **casamento de padrões**. Se você se encontrar trabalhando em um problema que você já solucionou antes, re-use a solução.

### 9.7 Vários intervalos

Conforme o número de intervalos aumenta, `noIntervalo` torna-se intragável. Com dois intervalos, não é tão ruim:

```
inferior = noIntervalo(a, 0.0, 0.5)
superior = noIntervalo(a, 0.5, 1)
```

Mas com quatro intervalos, começa a ficar desconfortável.:

```
intervalo1 = noIntervalo(a, 0.0, 0.25)
intervalo2 = noIntervalo(a, 0.25, 0.5)
intervalo3 = noIntervalo(a, 0.5, 0.75)
```

```
intervalo4 = noIntervalo(a, 0.75, 1.0)
```

Existem aqui dois problemas. Um é que temos que criar novos nomes de variável para cada resultado. O outro é que temos que calcular os limites de cada intervalo.

Vamos resolver o segundo problema primeiro. Se o número de intervalos é numeroDeIntervalos, então a largura de cada intervalo é 1.0 / numeroDeIntervalos.

Vamos usar um laço (loop) para calcular a faixa, ou largura, de cada intervalo. A variável do loop, i, conta de 0 até numeroDeIntervalos-1:

```
larguraDoIntervalo = 1.0 / numeroDeIntervalos
for i in range(numeroDeIntervalos):
    inferior = i * larguraDoIntervalo
    superior = inferior + larguraDoIntervalo
    print "do" inferior, "ao", superior
```

Para calcular o limite inferior (inferior) de cada intervalo, nós multiplicamos a variável do loop (i) pela largura do intervalo (larguraDoIntervalo). O limite superior (superior) está exatamente uma "largura de intervalo" acima.

Com numeroDeIntervalos = 8, o resultado é:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

Você pode confirmar que cada intervalo tem a mesma largura, que eles não se sobrepõe, e que eles cobrem toda a faixa de valores de 0.0 a 1.0.

Agora, de volta ao primeiro problema. Nós precisamos de uma maneira de guardar oito inteiros, usando a variável do loop para indicar cada um destes inteiros. Você deve estar pensando, "Lista!"

Nós temos que criar a lista de intervalos fora do loop, porque queremos fazer isto apenas uma vez. Dentro do loop, nós vamos chamar noIntervalo repetidamente e atualizar o i-ésimo elemento da lista:

```
numeroDeIntervalos = 8
intervalos = [0] * numeroDeIntervalos
larguraDoIntervalo = 1.0 / numeroDeIntervalos
for i in range(numeroDeIntervalos):
    inferior = i * larguraDoIntervalo
    superior = inferior + larguraDoIntervalo
    intervalos[i] = noIntervalo(lista, inferior, superior)
print intervalos
```

Com uma lista de 1000 valores, este código vai produzir esta lista de quantidades de valores em cada intervalo:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

Esses números estão razoavelmente próximos de 125, o que era o que esperávamos. Pelo menos eles estão próximos o bastante para nos fazer acreditar que o gerador de número aleatórios está funcionando.

Como exercício, teste esta função com algumas listas longas, e veja se o número de valores em cada um dos intervalos tendem a uma distribuição nivelada.

## 9.8 Uma solução em um só passo

Embora este programa funcione, ele não é tão eficiente quanto poderia ser. Toda vez que ele chama noIntervalo, ele percorre a lista inteira. Conforme o número de intervalos aumenta, a lista será percorrida um bocadinho de vezes.

Seria melhor fazer uma única passagem pela lista e calcular para cada valor o índice do intervalo ao qual o valor pertença. Então podemos incrementar o contador apropriado.

Na seção anterior, pegamos um índice, i, e o multiplicamos pela larguraDoIntervalo para encontrar o limite inferior daquele intervalo. Agora queremos pegar um valor entre 0.0 e 1.0 e encontrar o índice do intervalo ao qual ele se encaixa.

Já que este problema é o inverso do problema anterior, podemos imaginar que deveríamos dividir por larguraDoIntervalo em vez de multiplicar. Esta suposição está correta.

Já que larguraDoIntervalo = 1.0 / numeroDeIntervalos, dividir por larguraDoIntervalo é o mesmo que multiplicar por numeroDeIntervalos. Se multiplicarmos um número na faixa entre 0.0 e 1.0 por numeroDeIntervalos, obtemos um número na faixa entre 0.0 e numeroDeIntervalos. Se arredondarmos este número para baixo, ou seja, para o menor inteiro mais próximo, obtemos exatamente o que estamos procurando - o índice do intervalo:

```
numeroDeIntervalos = 8
intervalos = [0] * numeroDeIntervalos
for i in lista:
    indice = int(i * numeroDeIntervalos)
    intervalos[indice] = intervalos[indice] + 1
```

Usamos a função int para converter um número em ponto flutuante (float) para um inteiro.

Existe a possibilidade deste cálculo produzir um índice que esteja fora dos limites (seja negativo ou maior que len(intervalos)-1)?

Uma lista como intervalos que contém uma contagem do número de valores em cada intervalo é chamada de histograma.

Como exercício, escreva uma função chamada "histograma" que receba uma lista e um número de intervalos como argumentos e retorne um histograma com o número de intervalos solicitado.

## 9.9 Glossário

<b>tipo imutável (immutable type)</b>	Um tipo de elemento que não pode ser modificado. Atribuições a um elemento ou "fatiamento (slices)" XXX aos tipos imutáveis causarão erro.
<b>tipo mutável (mutable type)</b>	Tipo de dados onde os elementos podem ser modificados. Todos os tipos mutáveis, são tipos compostos. Listas e dicionários são exemplos de tipos de dados mutáveis. String e tuplas não são.
<b>tupla (tuple)</b>	Tipo sequencial similar as listas com exceção de que ele é imutável. Podem ser usadas Tuplas sempre que um tipo imutável for necessário, por exemplo uma "chave (key)" em um dicionário
<b>Atribuição a tupla (tuple)</b>	Atribuição a todos os elementos de uma tupla feita num único comando de



## Como pensar como um cientista da Computação usando Python

---

<b>assignment)</b>	atribuição. A atribuição aos elementos ocorre em paralelo, e não em sequência, tornando esta operação útil para swap, ou troca recíproca de valores entre variáveis (ex: a,b=b,a).
<b>determinístico (deterministic)</b>	Um programa que realiza a mesma coisa sempre que é executado.
<b>pseudo aleatório (pseudorandom)</b>	Uma sequência de números que parecem ser aleatórios mas são na verdade o resultado de uma computação "determinística"
<b>histograma (histogram)</b>	Uma lista de inteiros na qual cada elemento conta o número de vezes que algo acontece.

## Capítulo 10: Dicionários

Os tipos compostos que voce aprendeu - strings, listas e tuplas - utilizam inteiros como índices. Se voce tentar utilizar qualquer outro tipo como indice, voce receberá um erro.

Dicionários sao similiares a outros tipos compostos exceto por eles poderem user qualquer tipo imutavel de dados como indice. Como exemplo, nos criaremos um dicionário para traduzir palavras em Inglês para Espanhol. Para esse dicionário, os indices serão strings.

Uma maneira de criar um dicionario é começando com um dicionário vazio e depois adicionando elementos. Um dicionário vazio é denotado assim {}:

```
>>> ing2esp = {}
>>> ing2esp['one'] = 'uno'
>>> ing2esp['two'] = 'dos'
```

A primeira atribuição cria um dicionario chamado ing2esp; as outras atribuições adicionam novos elementos para o dicionário. Nos podemos imprimir o valor corrente de um dicionario da maneira usual:

```
>>> print ing2esp
{'one': 'uno', 'two': 'dos'}
```

Os elementos de um dicionário aparecem em uma lista separada por vírgulas. Cada entrada contém um índice e um valor separado por dois-pontos. Em um dicionário, os índices sao chamados de chaves, entao os elementos são chamados de pares chave-valor.

Outra maneira de criar dicionários é fornecendo uma lista de pares chaves-valor utilizando a mesma sintaxe da última saída.

```
>>> ing2esp = {'one': 'uno', 'two': 'dos', \
'three': 'tres'}
```

Se nos imprimirmos o valor de ing2esp novamente, nos teremos uma surpresa:

```
>>> print ing2esp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Os pares chave-valor não estão em ordem! Felizmente, não a motivos para se preocupar com a ordem, desde que os elementos do dicionário nunca sejam indexados com indices inteiros. Podemos usar as chaves para buscar os valores correspondentes:

```
>>> print ing2esp['two']
'dos'
```

A chave 'two' retornou o valor 'dos' mesmo pensando que retornaria o terceiro par chave-valor.

### 10.1 Operações dos Dicionários

O comando del remove um par chave-valor de um dicionário. Por exemplo, o dicionário abaixo contem os nomes de varias frutas e o numero de cada fruta em no estoque:

```
>>> inventario = {'abacaxis': 430, 'bananas': 312, \
'laranjas': 525, 'peras': 217}
>>> print inventario
{'laranjas': 525, 'abacaxis': 430, 'peras': 217, \
'bananas': 312}
```

Se alguém comprar todas as peras, podemos remover a entrada do dicionário:

```
>>> del inventario['peras']
>>> print inventario
{'laranjas': 525, 'abacaxis': 430, 'bananas': 312}
```

Ou se nós esperamos por mais peras em breve, nos podemos simplesmente trocar o valor associado as peras:

```
>>> inventario['peras'] = 0
>>> print inventario
{'laranjas': 525, 'abacaxis': 430, 'peras': 0, \
'bananas': 312}
```

A função len também funciona com dicionários; retornando o número de pares chave-valor:

```
>>> len(inventario)
4
```

### 10.2 Métodos dos Dicionários

Um método é parecido com uma função - possui parametros e retorna valores - mas a sintaxe é diferente. Por exemplo, o metodo keys recebe um dicionário e retorna uma lista com as chaves, mas em vez de usarmos a sintaxe de função keys(ing2esp), nos usamos a sintaxe de método ing2esp.keys():

```
>>> ing2esp.keys()
['one', 'three', 'two']
```

Dessa forma o ponto especifica o nome da função, keys, e o nome do objeto em que deve ser aplicada a função, ing2esp. Os parenteses indicam que esse método não possui parametros.

Ao invés de chamarmos um método, dizemos que ele é invocado, nesse caso, nós podemos dizer que nós estamos invocando keys do objeto ing2esp.

O método values é parecido; retorna a lista de valores de um dicionário:

```
>>> ing2esp.values()
['uno', 'tres', 'dos']
```

O método items retorna os dois, na forma de uma lista de tuplas - cada tupla com um par chave-valor:

```
>>> ing2esp.items()
[('one', 'uno'), ('three', 'tres'), ('two', 'dos')]
```

A sintaxe fornece uma informação util. Os colchetes indicam que isso é uma lista. Os parenteses indicam que os elementos da lista são tuplas.

Se o método recebe de algum parâmetro, se utiliza a mesma sintaxe das funções. Por exemplo, o método has\_key recebe uma chave e retorna verdadeiro (1) se a chave existe no dicionário:

```
>>> ing2esp.has_key('one')
True
>>> ing2esp.has_key('deux')
False
```

Se voce tentar chamar um método sem especificar em qual objeto, voce obterá um erro. Nesse caso, a mensagem de erro não é muito útil:

```
>>> has_key('one')
```

```
NameError: has_key
```

utilizamos o operador []:

```
>>> matriz[0,3]
1
```

Note que a sintaxe da representação de um dicionário não é a mesma que a sintaxe usada pela representação pelas listas. Em vez de usarmos dois índices inteiros, nós usamos apenas um índice, que é uma tupla de inteiros.

Mas existe um problema. Se tentarmos buscar um elemento zero, obteremos um erro, pois não existe uma entrada no dicionário para a chave especificada:

```
>>> matriz[1,3]
KeyError: (1,3)
```

O método **get** resolve esse problema:

```
>>> matriz.get((0,3), 0)
1
```

O primeiro parâmetro é a chave; o segundo é o valor que **get** retornará caso não existe a chave no dicionário:

```
>>> matriz.get((1,3), 0)
0
```

**get** definitivamente melhora a semântica e a sintaxe do acesso a matrizes esparsas.

### 10.3 Aliasing (XXX) e Copiar

Uma vez que os dicionários são mutáveis, você precisa saber sobre **Aliasing**. Sempre que duas variáveis referenciarem o mesmo objeto, quando uma é alterada, afeta a outra.

Se você quer modificar um dicionário e continuar com uma cópia original, utilize o método **copy**. Por exemplo, **opposites** é um dicionário que contém pares de antônimos:

```
>>> opposites = {'up': 'down', 'right': 'wrong', \
'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

**alias** e **opposites** se referem ao mesmo objeto; **copy** se refere a um novo objeto igual ao dicionário **opposites**. Se você modificar o **alias**, **opposites** também será alterado.

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

Se modificarmos **copy**, **opposites** não será modificado:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

### 10.4 Matrizes Esparsas

Na seção 8.14, nós usamos uma lista de listas para representar uma matriz. Essa é uma boa escolha se a matriz for principalmente de valores diferentes de zero, mas considerando uma matriz esparsa como essa:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Uma representação usando uma lista contém muitos zeros:

```
>>> matriz = [ [0,0,0,1,0],
               [0,0,0,0,0],
               [0,2,0,0,0],
               [0,0,0,0,0],
               [0,0,0,3,0] ]
```

Uma alternativa é usarmos um dicionário. Para as chaves, nós podemos usar tuplas que contêm os números da linha e a coluna. Abaixo uma representação em um dicionário da mesma matriz:

```
>>> matriz = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

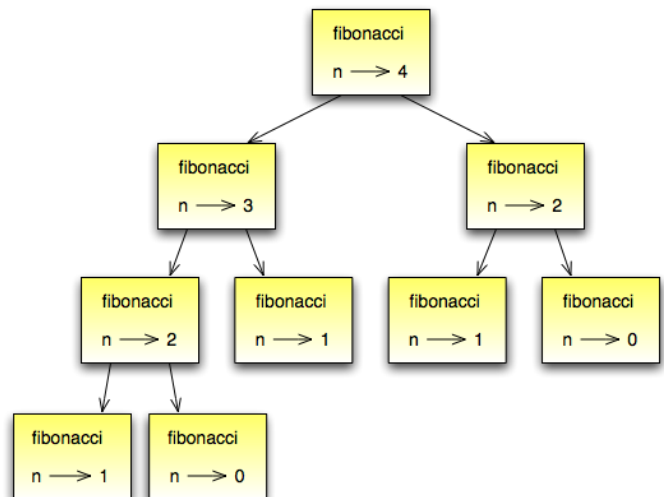
Nós precisamos apenas de três pares chave-valor, cada um sendo um elemento diferente de zero da matriz. Cada chave é uma tupla, e cada valor é um número inteiro.

Para acessarmos um elemento da matriz, nos

### 10.5 Hint

Se você brincou com a função **fibonacci** da seção 5.7, é provável que você notou que quanto maior o número passado para a função, mais tempo a função demora para executar. Além disso, o tempo da execução aumenta rapidamente. Em uma das nossas máquinas, **fibonacci(20)** executa instantaneamente, **fibonacci(30)** demora cerca de um segundo, e **fibonacci(40)** demora uma eternidade.

Para entender o porque, considere o gráfico de chamadas para **fibonacci** com  $n=4$ :



O gráfico mostra a estrutura da função, com linhas conectando cada execução com a execução que a chamou. No topo do gráfico, **fibonacci** tem  $n=4$ , que chama **fibonacci** com  $n=3$  e  $n=2$ . Em seguida, **fibonacci** com  $n=3$  chama **fibonacci** com  $n=2$  e  $n=1$ . É assim por diante.

Conte quantas vezes **fibonacci(0)** e **fibonacci(1)** são chamadas. Essa é uma solução ineficiente para o problema, e torna-se pior quando o parâmetro recebido é um número maior.

Uma boa solução é guardar os valores que já foram calculados armazenando-os em um dicionário. Um valor previamente calculado que é guardado para ser utilizado mais tarde é chamado de **hint**. Abaixo uma implementação de **fibonacci** usando **hints**:

```
>>> previous = {0:1, 1:1}
>>> def fibonacci(n):
    if previous.has_key(n):
        return previous[n]
    else:
        newValue = fibonacci(n-1) + \
fibonacci(n-2)
        previous[n] = newValue
        return newValue
```

O dicionário chamado **previous** guarda os números de Fibonacci que nós já conhecemos. Ele começa com apenas dois pares: 0 possui 1; e 1 possui 1.

Sempre que **fibonacci** é chamada, ela verifica o dicionário para determinar se ele já possui o resultado. Se o resultado estiver ali, a função pode retornar imediatamente sempre precisar fazer mais chamadas recursivas. Se o resultado não estiver ali, ele é calculado no **newValue**. O valor de **newValue** é adicionado no dicionário antes da função retornar.

Usando essa versão de **fibonacci**, nossa máquina consegue calcular **fibonacci(40)** em um piscar de olhos. Mas quando tentamos calcular **fibonacci(50)**, nós veremos um problema diferente:

```
>>> fibonacci(50)
OverflowError: integer addition
```

A resposta, que você verá em um minuto, é 20.365.011.074. O problema é que esse número é muito grande para guardarmos como um inteiro do Python<sup>1</sup>. Isso é overflow. Felizmente, esse problema tem uma solução simples.

### 10.6 Inteiros Longos

Python possui um tipo chamado **long int** que permite trabalharmos com qualquer tamanho de inteiros. Existem duas maneiras de criarmos um valor **long int**. A primeira é escrever um inteiro seguido de um L no final:

```
>>> type(1L)
<type 'long int'>
```

A outra maneira é usarmos a função **long** que converte um valor para um **long int**. **long** pode receber qualquer valor numérico e até mesmo uma string de dígitos:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

Todas as operações matemáticas funcionam com **long int** s, então não precisamos modificar muito para adaptar **fibonacci**:

```
>>> previous = {0: 1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

Somente trocando os valores iniciais de **previous**, conseguimos mudar o comportamento da **fibonacci**.

1 N.T. A partir do Python 2. XXX este erro não ocorre mais, pois em caso de sobrecarga o valor inteiro é automaticamente promovido para o tipo long.

Os dois primeiros números da sequência são **long ints**, então todos os números subsequentes da sequência também serão.

Como exercício, converta fatorial para produzir um inteiro longo como resultado.

### 10.7 Contando Letras

No capítulo 7, escrevemos uma função que contava o número de ocorrências de uma letra em uma string. A versão mais comum desse problema é fazer um histograma das letras da string, ou seja, quantas vezes cada letra aparece na string.

Um histograma pode ser útil para comprimir um arquivo de texto. Pois diferentes letras aparecem com diferentes frequências, podemos comprimir um arquivo usando pequenos códigos para letras comuns e longos códigos para letras que aparecem em menor frequência.

Dicionários fornecem uma maneira elegante de gerar um histograma:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...     letterCounts[letter] =
letterCounts.get(letter,0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Começamos com um dicionário vazio. Para cada letra da string, achamos o contador (possivelmente zero) e o incrementamos. No final, o dicionário contém pares de letras e as suas frequências.

É mais atraente mostrarmos o histograma na ordem alfabética. Podemos fazer isso com os métodos **items** e **sort**:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Você já tinha visto o método **items** antes, mas **sort** é o primeiro método que você se depara para aplicar em listas. Existem muitos outros métodos de listas, incluindo **append**, **extend**, e **reverse**. Consulte a documentação do Python para maiores detalhes.

### 10.8 Glossário

<b>dicionário (dictionary)</b>	Uma coleção de pares de chaves-valores que são mapeados pelas chaves, para se obter os valores. As chaves podem ser qualquer tipo de dados imutável, e os valores podem ser de qualquer tipo.
<b>chave (key)</b>	Um valor que é usado para buscar uma entrada em um dicionário.
<b>par chave-valor (key-value pair)</b>	Um dos itens de um dicionário.
<b>método (method)</b>	Um tipo de função que é chamada com uma sintaxe diferente e invocada no contexto de um objeto.
<b>invocar (invoke)</b>	Chamar um método.
<b>hint</b>	O armazenamento temporário de um valor pré-computado para evitar a computação redundante.
<b>overflow</b>	Um resultado numérico que é muito

grande para ser representado no formato numérico.
---



## Capítulo 11: Arquivos e exceções

### Arquivos e exceções

Durante a execução de um programa, seus dados ficam na memória. Quando o programa termina, ou o computador é desligado, os dados na memória desaparecem. Para armazenar os dados permanentemente, você tem que colocá-los em um **arquivo**. Arquivos usualmente são guardados em um disco rígido (HD), num disquete ou em um CD-ROM.

Quando existe um número muito grande de arquivos, eles muitas vezes são organizados dentro de **diretórios** (também chamados de *pastas* ou ainda *folders*). Cada arquivo é identificado por um nome único, ou uma combinação de um nome de arquivo com um nome de diretório.

Lendo e escrevendo em arquivos, os programas podem trocar informações uns com os outros e gerar formatos imprimíveis como PDF.

Trabalhar com arquivos é muito parecido com trabalhar com livros. Para utilizar um livro, você tem que abri-lo. Quando você termina, você tem que fechá-lo. Enquanto o livro estiver aberto, você pode tanto lê-lo quanto escrever nele. Em qualquer caso, você sabe onde você está situado no livro. Na maioria das vezes, você lê o livro inteiro em sua ordem natural, mas você também pode saltar através de alguns trechos (skip around).

Tudo isso se aplica do mesmo modo a arquivos. Para abrir um arquivo, você especifica o nome dele e indica o que você quer, seja ler ou escrever (gravar).

Abrir um arquivo cria um objeto arquivo. Neste exemplo, a variável `f` se referencia ao novo objeto arquivo.

```
>>> f = open("teste.dat", "w")
>>> print f
<open file "teste.dat", mode "w" at fe820>
```

A função `open` recebe dois argumentos. O primeiro é o nome do arquivo, e o segundo é o modo. Modo `w`? significa que estamos abrindo o arquivo para gravação (*write*, escrever).

Se não existir nenhum arquivo de nome `teste.dat`, ele será criado. Se já existir um, ele será substituído pelo arquivo que estamos gravando (ou escrevendo).

Quando executamos um comando `print` sobre o objeto arquivo, visualizamos o nome do arquivo, o modo e a localização do objeto na memória.

Para colocar dados dentro do arquivo, invocamos o método `write` do objeto arquivo:

```
>>> f.write("Agora é hora")
>>> f.write("de fechar o arquivo")
```

Fechar o arquivo diz ao sistema que terminamos de escrever (gravar) e que o arquivo está livre para ser lido:

```
>>> f.close()
```

Agora podemos abrir o arquivo de novo, desta vez para leitura, e ler o seu conteúdo para uma string. Desta vez, o argumento modo é `r` para leitura (*reading*):

```
>>> f = open("teste.dat", "r")
```

Se tentarmos abrir um arquivo que não existe,

temos um erro:

```
>>> f = open("teste.cat", "r")
IOError: [Errno 2] No such file or directory: \
'teste.cat'
```

Sem nenhuma surpresa, o método `read` lê dados do arquivo. Sem argumentos, ele lê todo o conteúdo do arquivo:

```
>>> texto = f.read()
>>> print texto
Agora é horade fechar o arquivo
```

Não existe espaço entre *hora* e *de* porque nós não gravamos um espaço entre as strings.

`read` também pode receber um argumento que indica quantos caracteres ler:

```
>>> f = open("teste.dat", "r")
>>> print f.read(9)
Agora é h
```

Se não houver caracteres suficientes no arquivo, `read` retorna os caracteres restantes. Quando chegamos ao final do arquivo, `read` retorna a string vazia:

```
>>> print f.read(1000006)
orade fechar o arquivo
>>> print f.read()

>>>
```

A função seguinte, copia um arquivo, lendo e gravando até cinquenta caracteres de uma vez. O primeiro argumento é o nome do arquivo original; o segundo é o nome do novo arquivo:

```
def copiaArquivo(velhoArquivo, novoArquivo):
    f1 = open(velhoArquivo, "r")
    f2 = open(novoArquivo, "w")
    while 1:
        texto = f1.read(50)
        if texto == "":
            break
        f2.write(texto)
    f1.close()
    f2.close()
    return
```

O comando `break` é novo. O que ele faz é saltar a execução para fora do loop; o fluxo de execução passa para o primeiro comando depois do loop.

Neste exemplo, o loop `while` é infinito porque o valor `1` é sempre verdadeiro. O único modo de sair do loop é executando o `break`, o que ocorre quando `texto` é a string vazia, o que ocorre quando alcançamos o fim do arquivo.

### 11.1 Arquivos texto

Um arquivo texto é um arquivo que contém caracteres imprimíveis e espaços, organizados dentro de linhas separadas por caracteres de nova linha. Já que Python é especialmente projetado para processar arquivos texto, ele oferece métodos que tornam esta tarefa mais fácil.

## Como pensar como um cientista da Computação usando Python

Para demonstrar, vamos criar um arquivo texto com três linhas de texto separadas por caracteres de nova linha:

```
>>> f = open("teste.dat", "w")
>>> f.write("linha um\nlinha dois\nlinha três\n")
>>> f.close()
```

O método `readline` lê todos os caracteres até, e incluindo, o próximo caractere de nova linha:

```
>>> f = open("teste.dat", "r")
>>> print f.readline()
linha um
>>>
```

`readlines` retorna todas as linhas restantes como uma lista de strings:

```
>>> print f.readlines()
['linha dois\n', 'linha três\n']
```

Neste caso, a saída está em formato de lista, o que significa que as strings aparecem entre aspas e o caractere de nova linha aparece como a seqüência de escape `\n`.

No fim do arquivo, `readline` retorna a string vazia e `readlines` retorna a lista vazia:

```
>>> print f.readline()
>>> print f.readlines()
[]
```

A seguir temos um exemplo de um programa de processamento de linhas. `filtraArquivo` faz uma cópia de `velhoArquivo`, omitindo quaisquer linhas que comecem por `#`:

```
def filtraArquivo(velhoArquivo, novoArquivo):
    f1 = open(velhoArquivo, "r")
    f2 = open(novoArquivo, "w")
    while 1:
        texto = f1.readline()
        if texto == "":
            break
        if texto[0] == '#':
            continue
        f2.write(texto)
    f1.close()
    f2.close()
    return
```

O comando `continue` termina a iteração corrente do loop, mas continua iterando o loop. O fluxo de execução passa para o topo do loop, checka a condição e prossegue conforme o caso.

Assim, se `texto` for a string vazia, o loop termina. Se o primeiro caractere de `texto` for o jogo da velha (`? # ?`), o fluxo de execução passa para o topo do loop. Somente se ambas as condições falharem é que `texto` será copiado para dentro do novo arquivo.

### 11.2 Gravando variáveis

O argumento de `write` tem que ser uma string, assim se quisermos colocar outros valores em um arquivo, temos de convertê-los para strings primeiro. A maneira mais fácil de fazer isso é com a função `str`:

```
>>> x = 52
>>> f.write(str(x))
```

Uma alternativa é usar o **operador de formatação** `%`. Quando aplicado a inteiros, `%` é o operador

módulo. Mas quando o primeiro operador é uma string, `%` é o operador de formatação.

O primeiro operando é a **string de formatação**, e o segundo operando é uma tupla de expressões. O resultado é uma string que contém os valores das expressões, formatadas de acordo com a string de formatação.

Num exemplo simples, a **seqüência de formatação** `"??%d?"` significa que a primeira expressão na tupla deve ser formatada como um inteiro. Aqui a letra `d` representa `?decimal?`.

```
>>> carros = 52
>>> "%d" % carros
'52'
```

O resultado é a string `'52'`, que não deve ser confundida com o valor inteiro 52.

Uma seqüência de formatação pode aparecer em qualquer lugar na string de formatação, assim, podemos embutir um valor em uma seqüência:

```
>>> carros = 52
>>> "Em julho vendemos %d carros." % carros
'Em julho vendemos 52 carros.'
```

A seqüência de formatação `"%f"` formata o próximo item da tupla como um número em ponto flutuante, e `"%s"` formata o próximo como uma string:

```
>>> "Em %d dias fizemos %f milhões %s." % \
(34,6.1, 'reais')
'Em 34 dias fizemos 6.100000 milhões de reais.'
```

Por padrão, o formato de ponto flutuante exhibe seis casas decimais.

O número de expressões na tupla tem que ser igual ao número de seqüências de formatação na string. Além disso, os tipos das expressões têm que iguais aos da seqüência de formatação:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'reais'
TypeError: illegal argument type for built-in \
operation
```

No primeiro exemplo, não existem expressões suficientes; no segundo, a expressão é do tipo errado.

Para um controle maior na formatação de números, podemos especificar o número de dígitos como parte da seqüência de formatação:

```
>>> "%6d" % 62
'   62'
>>> "%12f" % 6.1
'   6,100000'
```

O número depois do sinal de porcentagem é o número mínimo de espaços que o valor ocupará. Se o valor fornecido tiver um número menor de dígitos, espaços em branco serão adicionados antes para preencher o restante. Se o número de espaços for negativo, os espaços serão adicionados depois:

```
>>> "%-6d" % 62
'62'
```

Para números em ponto-flutuante, também podemos especificar o número de dígitos depois da vírgula:

```
>>> "%12.2f" % 6.1
'   6.10'
```

Neste exemplo, o resultado reserva 12 espaços e inclui dois dígitos depois da vírgula. Esta formatação é útil para exibir valores monetários com os centavos alinhados.

Por exemplo, imagine um dicionário que contém nomes de estudantes como chaves e salários-hora como valores. Aqui está uma função que imprime o conteúdo do dicionário como um relatório formatado:

```
def relatorio(salarios):
    estudantes = salarios.keys()
    estudantes.sort()
    for estudante in estudantes:
        print "%-20s %12.02f" % (estudante, \
salarios[estudante])
```

Para testar esta função, criaremos um pequeno dicionário e imprimiremos o conteúdo:

```
>>> salarios = {'maria': 6.23, 'joão': 5.45, \
'josué': 4.25}
>>> relatorio(salarios)
joão                5.45
josué                4.25
maria                6.23
```

Controlando a largura de cada valor, podemos garantir que as colunas ficarão alinhadas, desde que os nomes contêm menos que vinte e um caracteres e os salários sejam menores do que um bilhão de reais por hora.

### 11.3 Diretórios

Quando você cria um novo arquivo abrindo-o e escrevendo nele, o novo arquivo fica no diretório corrente (seja lá onde for que você esteja quando rodar o programa). Do mesmo modo, quando você abre um arquivo para leitura, Python procura por ele no diretório corrente.

Se você quiser abrir um arquivo que esteja em algum outro lugar, você tem que especificar o **caminho** (path) para o arquivo, o qual é o nome do diretório (ou folder) onde o arquivo está localizado:

```
>>> f = open("/usr/share/dict/words", "r")
>>> print f.readline()
Aarhus
```

Este exemplo abre um arquivo chamado words que reside em um diretório de nome dict, o qual reside em share, o qual reside em usr, o qual reside no diretório de mais alto nível do sistema, chamado /.

Você não pode usar / como parte do nome de um arquivo; ela é um caractere reservado como um delimitador entre nomes de diretórios e nomes de arquivos.

O arquivo /usr/share/dict/words contém uma lista de palavras em ordem alfabética, na qual a primeira palavra é o nome de uma universidade Dinamarquesa.

### 11.4 Pickling

Para colocar valores em um arquivo, você tem que convertê-los para strings. Você já viu como fazer isto com str:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

O problema é que quando você lê de volta o valor, você tem uma string. O Tipo original da informação foi perdido. De fato, você não pode sequer dizer onde começa um valor e termina outro:

```
>>> f.readline()
?12.3[1, 2, 3]?
```

A solução é o pickling, assim chamado porque ?preserva? estruturas de dados. O módulo pickle contém os

comandos necessários. Para usá-lo, importe pickle e então abra o arquivo da maneira usual:

```
>>> import pickle
>>> f = open('test.pck?', 'w?')
```

Para armazenar uma estrutura de dados, use o método dump e então feche o arquivo do modo usual:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Então, podemos abrir o arquivo para leitura e carregar as estruturas de dados que foram descarregadas (dumped):

```
>>> f = open('test.pck?', 'r?')
>>> x = pickle.load(f)
>>> x
12,3
>>> type(x)
<type 'float'?>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'?>
```

Cada vez que invocamos load, obtemos um único valor do arquivo, completo com seu tipo original.

### 11.5 Exceções

Whenever que um erro em tempo de execução acontece, ele gera uma exceção. Usualmente, o programa pára e Python exhibe uma mensagem de erro.

Por exemplo, dividir por zero gera uma exceção:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

Do mesmo modo, acessar um item de lista inexistente:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

Ou acessar uma chave que não está em um dicionário:

```
>>> b = {}
>>> print b['what?']
KeyError: what
```

Em cada caso, a mensagem de erro tem duas partes: o tipo do erro antes dos dois pontos, e especificidades do erro depois dos dois pontos. Normalmente Python também exhibe um *traceback* de onde estava a execução do programa, mas nós temos omitido esta parte nos exemplos.

Às vezes queremos executar uma operação que pode causar uma exceção, mas não queremos que o programa pare. Nós podemos tratar a exceção usando as instruções try e except.

Por exemplo, podemos pedir ao usuário um nome de arquivo e então tentar abri-lo. Se o arquivo não existe, não queremos que o programa trave; queremos tratar a exceção:

```
nomedoarquivo = raw_input('Entre com o nome do \
arquivo: ?')
try:
```

```
f = open (nomedoarquivo, 'r')
except:
    print 'Não existe arquivo chamado', nomedoarquivo
```

A instrução `try` executa os comandos do primeiro bloco. Se não ocorrerem exceções, ele ignora a instrução `except`. Se qualquer exceção acontece, ele executa os comandos do ramo `except` e continua.

Podemos encapsular esta habilidade numa função: `existe` toma um nome de arquivo e retorna verdadeiro se o arquivo existe e falso se não existe:

```
def existe(nomedoarquivo):
    try:
        f = open(nomedoarquivo)
        f.close()
        return 1
    except:
        return 0
```

Você pode usar múltiplos blocos `except` para tratar diferentes tipos de exceções. O Manual de Referência de Python (Python Reference Manual) tem os detalhes.

Se o seu programa detecta uma condição de erro, você pode fazê-lo lançar uma exceção. Aqui está um exemplo que toma uma entrada do usuário e testa se o valor é 17. Supondo que 17 não seja uma entrada válida por uma razão qualquer, nós lançamos uma exceção.

```
def entraNumero():
    x = input ('Escolha um número: ?')
    if x == 17:
        raise 'ErroNumeroRuim', '17 é um número ruim?'
    return x
```

O comando `raise` toma dois argumentos: o tipo da exceção e informações específicas sobre o erro. `ErroNumeroRuim` é um novo tipo de exceção que nós inventamos para esta aplicação.

Se a função que chamou `entraNumero` trata o erro, então o programa pode continuar; de outro modo, Python exibe uma mensagem de erro e sai:

```
>>> entraNumero()
Escolha um número: 17
ErroNumeroRuim: 17 é um número ruim
```

A mensagem de erro inclui o tipo da exceção e a informação adicional que você forneceu.

Como um exercício, escreva uma função que use `entraNumero` para pegar um número do teclado e que trate a exceção `ErroNumeroRuim`.

## 11.6 Glossário

<b>arquivo (file)</b>	Uma entidade nomeada, usualmente armazenada em um disco rígido (HD), disquete ou CD-ROM, que contém uma seqüência de caracteres.
<b>diretório (directory)</b>	Uma coleção nomeada de arquivos, também chamado de pasta ou folder.
<b>caminho (path)</b>	Uma seqüência de nomes de diretórios que especifica a exata localização de um arquivo.
<b>arquivo texto (text file)</b>	Um arquivo que contém caracteres organizados em linhas separadas por caracteres de nova linha.
<b>comando break (break statement)</b>	Um comando que força a atual iteração de um loop a terminar. O fluxo de execução vai para o topo do loop, testa a condição e prossegue conforme o caso.

## Capítulo 12: Classes e objetos

### 12.1 Tipos compostos definidos pelo usuário

Depois de usarmos alguns tipos nativos do Python, estamos prontos para criar um tipo de dados: o Ponto.

Considere o conceito matemático de um ponto. Em duas dimensões, um ponto é um par de números (coordenadas) que são tratadas coletivamente como um objeto simples. Na notação matemática, pontos são freqüentemente escritos entre parênteses com vírgula separando as coordenadas. Por exemplo, (0, 0) representa a origem, e (x, y) representa o ponto x unidades à direita, e y unidades acima da origem.

Uma maneira natural para representar um ponto em Python, é com dois valores numéricos em ponto flutuante. A questão, então, é como agrupar estes dois valores em um objeto composto. A maneira rápida e rasteira é usar uma lista ou uma tupla, e para algumas aplicações, isso pode ser a melhor escolha<sup>1</sup>.

Uma alternativa é definir um novo tipo composto, também chamado uma **classe**. Esta abordagem envolve um pouco mais de esforço, mas ela tem vantagens que logo ficarão evidentes.

Eis a definição de uma classe:

```
class Ponto:
    pass
```

Definições de classes podem aparecer em qualquer parte de um programa, mas elas costumam ficar próximas do começo do programa (após os comandos import). As regras de sintaxe para a definição de classes são as mesmas de outros comandos compostos (veja Seção 4.4).

A definição acima cria uma nova classe chamada Ponto. O comando pass não tem nenhum efeito; aqui ele é necessário porque um comando composto precisa ter algo no seu corpo.

Quando criamos a classe Ponto, criamos um novo tipo de dado, também chamado Ponto. Os membros deste novo tipo são chamados **instâncias** deste tipo ou **objetos**. Criar uma nova instância é **instanciar**. Para instanciar o objeto Ponto, invocamos a função (adivinhou?) Ponto:

```
final = Ponto()
```

A variável final agora contém uma referência a um novo objeto da classe Ponto. Uma função como Ponto, que cria novos objetos, é chamada **construtor**.

### 12.2 Atributos

Podemos adicionar novos dados em uma instância usando a notação de ponto (dot notation):

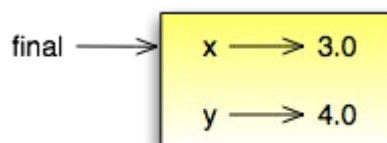
```
>>> final.x = 3.0
```

1 N.T.: A linguagem Python também incorpora um tipo nativo complex que representa números complexos. Uma instância de complex, como a=3+5j possui dois valores de ponto flutuante em seus atributos a.real e a.imag, e pode ser utilizada para armazenar pontos em um espaço bi-dimensional.

```
>>> final.y = 4.0
```

Esta sintaxe é similar à sintaxe para acessar uma variável de um módulo, como math.pi ou string.uppercase. Neste caso, porém, estamos acessando um item de dado de uma instância. Estes itens são chamados **atributos**.

O seguinte diagrama de estado mostra o resultado destas atribuições:



A variável final refere a um objeto Ponto, que contém dois atributos. Cada atributo faz referência a um número em ponto flutuante.

Podemos ler o valor de um atributo usando a mesma sintaxe:

```
>>> print final.y
4.0
>>> x = final.x
>>> print x
3.0
```

A expressão final.x significa, "Vá ao objeto final e pegue o valor de x". Neste caso, atribuímos este valor a uma variável cujo nome é 'x'. Não há conflito entre a variável x e o atributo x. O propósito da notação objeto.atributo é identificar a qual variável você está fazendo referência de forma que não é ambíguo.

Você pode usar a notação objeto.atributo como parte de qualquer expressão; assim os seguintes comandos são válidos:

```
print '(' + str(final.x) + ', ' + str(final.y) + ')'
distAoQuadrado = final.x * final.x + final.y * \
    final.y
```

A primeira linha imprime (3.0, 4.0); a segunda linha calcula o valor 25.0.

É tentador imprimir o valor do próprio objeto final:

```
>>> print final
<__main__.Ponto instance at 80f8e70>
```

O resultado indica que final é uma instância da classe Ponto e foi definida no programa principal: \_\_main\_\_. 80f8e70 é o identificador único deste objeto, escrito em hexadecimal (base 16). Esta não é provavelmente a forma mais informativa para mostrar um objeto Ponto. Logo você irá ver como mudar isso.

Como exercício, crie e imprima um objeto Ponto, e então use id para imprimir o identificador único do objeto. Traduza a forma hexadecimal para a forma decimal e confirme se são compatíveis.



### 12.3 Instâncias como parâmetros

Você pode passar uma instância como um parâmetro da forma usual. Por exemplo:

```
def mostrarPonto(p):
    print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

A função `mostrarPonto` pega o ponto (`p`) como um argumento e mostra-o no formato padrão. Se você chamar `mostrarPonto(final)`, a saída será (3.0, 4.0).

Como um exercício, re-escreva a função distância da Seção 5.2 para receber dois pontos como parâmetros, ao invés de quatro números.

### 12.4 O significado de "mesmo"

O significado da palavra "mesmo" parece perfeitamente claro até que você pense a respeito, e então você percebe que há mais nesta palavra do que você esperava.

Por exemplo, se você diz "Cris e eu temos o mesmo carro", você está dizendo que o carro de Cris e o seu são do mesmo fabricante e modelo, mas são dois carros diferentes. Se você disser "Cris e eu temos a mesma mãe", você está dizendo que a mãe de Cris e a sua, são a mesma pessoa<sup>1</sup>. Portanto a idéia de 'semelhança' é diferente dependendo do contexto.

Quando falamos de objetos, há uma ambigüidade similar. Por exemplo, se dois Pontos forem os mesmos, isto quer dizer que eles contêm os mesmos dados (coordenadas) ou que são realmente o "mesmo" objeto?

Para verificar se duas referências se referem ao 'mesmo' objeto, use o operador '=='<sup>2</sup>. Por exemplo:

```
>>> p1 = Ponto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Ponto()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
False
```

Mesmo que `p1` e `p2` conttenham as mesmas coordenadas, os dois não representam o mesmo objeto. Se atribuirmos `p1` a `p2`, então as duas variáveis são pseudônimos do mesmo objeto.

```
>>> p2 = p1
>>> p1 == p2
True
```

Este tipo de igualdade é chamado de igualdade rasa porque ela compara somente as referências e não o conteúdo dos objetos.

- 1 Nem todos os idiomas têm este problema. Por exemplo, em alemão há palavras diferentes para diferentes sentidos de "mesmo". "Mesmo carro" nesse contexto seria "gleiche Auto", e "mesma mãe" seria "selbe Mutter".
- 2 LR: Eu não diria que devemos usar `==` para verificar se dois objetos são o mesmo. Isto é uma falha do livro que talvez se origine no original que falava de Java. Em Python o operador `is` faz o mesmo que o `==` de Java: compara referências, e portanto serve para determinar se duas variáveis apontam para o mesmo objeto. No entanto, a o código acima está correto porque em Python a implementação default de `==` (método `__eq__`) é comparar o id das instâncias, porém as classes `list` e `dict`, por exemplo, implementam `__eq__` comparando os valores contidos (ex.: isto retorna True: `l1 = [1,2,3]; l2 = [1,2,3]; l1 == l2`).

Para comparar o conteúdo dos objetos -- igualdade profunda -- podemos escrever uma função chamada `mesmoPonto`:

```
def mesmoPonto(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)
```

Agora se criarmos dois diferentes objetos que contêm os mesmos dados, podemos usar `mesmoPonto` para verificar se eles representam o mesmo ponto.

```
>>> p1 = Ponto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Ponto()
>>> p2.x = 3
>>> p2.y = 4
>>> mesmoPonto(p1, p2)
True
```

É claro, se as duas variáveis referirem ao mesmo objeto, elas têm igualdade rasa e igualdade profunda.

### 12.5 Retângulos

Digamos que desejemos uma classe para representar um retângulo. A questão é, qual informação temos de prover para especificar um retângulo? Para manter as coisas simples, assuma que o retângulo é orientado verticalmente ou horizontalmente, nunca em um ângulo.

Há algumas possibilidades: poderíamos especificar o centro do retângulo (duas coordenadas) e seu tamanho (largura e altura); ou poderíamos especificar um dos lados e o tamanho; ou poderíamos especificar dois lados opostos. A escolha convencional é especificar o canto superior esquerdo do retângulo e o tamanho.

Novamente, vamos definir uma nova classe:

```
class Rectangle:
    pass
```

E instanciá-la:

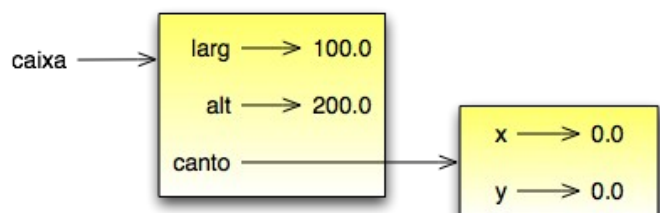
```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

Este código cria um novo objeto Retângulo com dois atributos ponto-flutuante. Para especificar o canto superior esquerdo, podemos embutir um objeto dentro de um objeto!

```
box.corner = Ponto()
box.corner.x = 0.0;
box.corner.y = 0.0;
```

A expressão `box.corner.x` significa, "vá ao objeto referenciado por 'box' e selecione o atributo 'corner'; então vá ao objeto 'corner' e deste, selecione o atributo de nome 'x'".

A figura mostra o estado deste objeto:



## 12.6 Instancias como valores retornados

Funções podem retornar instâncias. Por exemplo, `findCenter` pega um `Retângulo` como um argumento e retorna um `Ponto` que contem as coordenadas do centro do retângulo:

```
def findCenter(box):
    p = Ponto()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
```

Para chamar esta função, passe 'box' como um argumento e coloque o resultado em uma variável.

```
>>> center = findCenter(box)
>>> print mostrarPonto(center)
(50.0, 100.0)
```

## 12.7 Objetos são mutáveis

Podemos mudar o estado de um objeto fazendo uma atribuição a um dos seus atributos. Por exemplo, para mudar o tamanho de um retângulo sem mudar sua posição, podemos modificar os valores de sua largura e altura. Veja:

```
box.width = box.width + 50
box.height = box.height + 100
```

Poderíamos encapsular este código em um método e generaliza-lo para aumentar o tamanho deste retângulo em qualquer medida:

```
def growRect(box, dwidth, dheight):
    box.width = box.width + dwidth
    box.height = box.height + dheight
```

As variáveis `dwidth` e `dheight` indicam em quanto vamos aumentar o tamanho do retângulo em cada direção. Chamando este método, teríamos o mesmo efeito.

Por exemplo, poderíamos criar um novo `Retângulo` com o nome de 'bob' e passar este nome para o método `growRect`:

```
>>> bob = Rectangle()
>>> bob.width = 100.00
>>> bob.height = 200.00
>>> bob.corner.x = 0.0;
>>> bob.corner.y = 0.0;
>>> growRect(bob, 50, 100)
```

Enquanto `growRect` está sendo executado, o parâmetro 'box' é um alias (apelido) para 'bob'. Qualquer mudança feita em 'box', também irá afetar 'bob'.

Como exercício, escreva uma function (método) com o nome de `moveRect` que pega um `Rectangle` e dois parâmetros com o nome de 'dx' e 'dy'. Esta função deverá mudar a localização do retângulo através da adição de 'dx' à coordenada 'x' e da adição de 'dy' à coordenada 'y'.

## 12.8 Copiando

Ao usar 'alias' - como fizemos na seção anterior - podemos tornar o programa um pouco difícil de ler ou entender, pois as mudanças feitas em um local, podem afetar inesperadamente um outro objeto. E pode se tornar difícil de encontrar todas as variáveis que podem afetar um dado objeto.

Copiar um objeto é frequentemente uma alternativa ao 'alias'. O modulo 'copy' contém uma função chamada 'copy' que duplica um qualquer objeto. Veja:

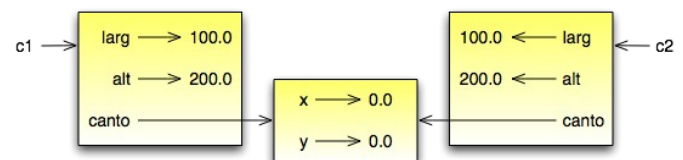
```
>>> import copy
>>> p1 = Ponto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
0
>>> mesmoPonto(p1, p2)
1
```

Uma vez que importamos o modulo 'copy', podemos usar o método 'copy' para criar um outro 'Ponto'. `p1` e `p2` não representam o mesmo ponto, mas eles contem os mesmos dados.

Para copiar um simples objeto como um 'Ponto', que não contem nenhum objeto embutido, 'copy' é suficiente. Isto eh chamado 'shallow' copia.

Mas para um objeto como um 'Rectangle', que contem uma referencia para um 'Ponto', o método 'copy' não irá executar corretamente a copia. Ele irá copiar a referencia para o objeto 'Ponto', portanto o que acontece aqui é que os dois `Rectangle` (o novo e o antigo) irão fazer referencia a um simples 'Ponto'.

Em outras palavras, se criarmos um 'box', `c1`, utilizando a forma usual, e depois fazer uma copia, `c2`, usando o método 'copy', o diagrama de estado resultante ficará assim:



o resultado não será o que esperamos. Neste caso, invocando 'growRect' em um dos retângulos (`c1`), isto não irá afetar o outro retângulo (`c2`, neste exemplo). Mas se usarmos o método 'moveRect' em qualquer um deles, isto irá inevitavelmente afetar o outro. Este comportamento é confuso e propenso a erros!

Mas felizmente o modulo 'copy' contem um método chamado 'deepcopy' que copia não somente o objeto, mas também copia todo e qualquer objeto 'embutido' neste objeto. Por isto, você não ficará surpreso porque este método chama-se 'deepcopy' (copia profunda) não é? Veja como funciona:

```
>>> c2 = copy.deepcopy(c1)
```

Agora, `c1` e `c2` são objetos completamente separados.

Podemos usar 'deepcopy' para re-escrever 'growRect' sendo que ao invés de modificar um `Rectangle` existente, ele cria um novo que tem a mesma localização do outro, mas com novas dimensões:

```
def growRect(box, dwidth, dheight):
    import copy
    newBox = copy.deepcopy(box)
    newBox.width = newBox.width + dwidth
    newBox.height = newBox.height + dheight
    return newBox
```

Como exercício, re-escreva o método 'moveRect' para ele criar e retornar um novo `Rectangle` ao invés de apenas modificar o antigo.

### 12.9 Glossário

---

<b>classe (class)</b>	Um tipo composto (XXX compound type) definido pelo usuário. Uma classe também pode ser visualizada como um molde que define a forma dos objetos que serão suas instâncias.
<b>instanciar (instantiate)</b>	Criar uma instância de uma classe.
<b>instância (instance)</b>	Um objeto que pertence a uma classe.
<b>objeto (object)</b>	Um tipo de dado composto comumente utilizado para representar uma coisa ou um conceito do mundo real.
<b>construtor (constructor)</b>	Um método utilizado para criar novos objetos.
<b>atributo (attribute)</b>	Um dos itens de dados nomeados que compõem uma instância.
<b>igualdade rasa (shallow equality)</b>	Igualdade de referências; ocorre quando duas referências apontam para o mesmo objeto.
<b>igualdade profunda (deep equality)</b>	Igualdade de valores; ocorre quando duas referências apontam para objetos que têm o mesmo valor.
<b>cópia rasa (shallow copy)</b>	Ato de copiar o conteúdo de um objeto, incluindo as referências a objetos embutidos (XXX embedded); implementada pela função copy do módulo copy.
<b>cópia profunda (deep copy)</b>	Ato de copiar o conteúdo de um objeto, bem como dos objetos embutidos (XXX embedded), e dos objetos embutidos nestes, e assim por diante; implementada pela função deepcopy do módulo copy.

## Capítulo 13: Classes e funções

### 13.1 Horário

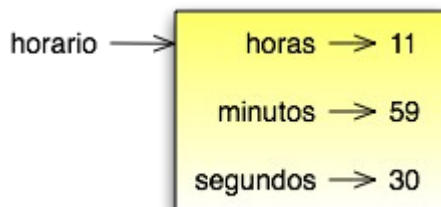
Como exemplo de outro tipo definido pelo usuário, vamos definir uma classe chamada Horário que grava os registros de horário do dia. Eis a definição da classe:

```
class Horário:
    pass
```

Podemos criar uma nova instância de Horário e determinar atributos para horas, minutos e segundos:

```
horario = Horário()
horario.horas = 11
horario.minutos = 59
horario.segundos = 30
```

O diagrama de estado para o objeto Horário parece com isso:



Como exercício, escreva uma função 'imprimirHorario' que tenha como argumento um objeto Horário e imprima-o na forma horas:minutos:segundos.

Como um segundo exercício, escreva uma função booleana que tenha como argumento dois objetos Horário, h1 e h2, e retorne verdadeiro (1) se h1 vem depois de h2 cronologicamente, do contrário, retorne falso (0).

### 13.2 Funções Puras

Nas próximas sessões, vamos escrever duas versões de uma função chamada adicionaHorario, que calcula a soma de dois horários. Elas vão demonstrar 2 tipos de funções: funções puras e funções modificadoras.

Segue uma versão rudimentar de somaHorario:

```
def somaHorario(h1, h2):
    soma = Horário()
    soma.horas = h1.horas + h2.horas
    soma.minutos = h1.minutos + h2.minutos
    soma.segundos = h1.segundos + h2.segundos
    return soma
```

A função cria um novo objeto Horário, inicializa os seus atributos, e retorna uma referência para o novo objeto. Isso é chamado de função pura pois ela não modifica nenhum dos objetos que são passados como parâmetros e não tem nenhum efeito colateral, como imprimir um valor ou pegar entrada do usuário.

Aqui está um exemplo de como usar esta função. Nós vamos criar dois objetos Horário: horarioAtual, que contém o horário atual; e horarioDoPao, que contém a

quantidade de tempo que a máquina de fazer pão gasta para fazer pão. Então vamos usar somaHorario para tentar saber quando o pão estará pronto. Se você não tiver terminado de escrever imprimirHorario ainda, de uma olhada na seção 14.2 antes de você continuar isso:

```
>>> horarioAtual = Horário()
>>> horarioAtual.horas = 9
>>> horarioAtual.minutos = 14
>>> horarioAtual.segundos = 30
```

```
>>> horarioDoPao = Horário()
>>> horarioDoPao.horas = 3
>>> horarioDoPao.minutos = 35
>>> horarioDoPao.segundos = 0
```

```
>>> horarioTermino = somaHorario(horarioAtual, \
horarioDoPao)
>>> imprimirHorario(horarioTermino)
```

A saída deste programa é 12:49:30, o que é correto. Por outro lado, existem casos onde o resultado não é correto. Você pode pensar em algum ?

O problema é que esta função não lida com casos onde o número de segundos ou minutos é acrescentado em mais de sessenta. Quando isso acontece, temos de "transportar" os segundos extras para a coluna dos minutos ou os minutos extras na coluna das horas.

Aqui está a segunda versão corrigida da função:

```
def somaHorario(t1, t2):
    soma = Horário()
    soma.horas = t1.horas + t2.horas
    soma.minutos = t1.minutos + t2.minutos
    soma.segundos = t1.segundos + t2.segundos

    if soma.segundos >= 60:
        soma.segundos = soma.segundos - 60
        soma.minutos = soma.minutos + 1

    if soma.minutos >= 60:
        soma.minutos = soma.minutos - 60
        soma.horas = soma.horas + 1

    return soma
```

Apesar desta função estar correta, ela está começando a ficar grande. Depois vamos sugerir uma aproximação alternativa que rende um código menor. Clique aqui para feedback

### 13.3 Modificadores

Existem momentos quando é útil para uma função modificar um ou mais dos objetos que ela recebe como parâmetro. Usualmente, quem está chamando a função mantém uma referência para os objetos que ele passa, de forma que quaisquer mudanças que a função faz são visíveis para quem está chamando. Funções que trabalham desta forma são chamadas modificadores.

incrementar, que adiciona um número dado de segundos para um objeto Horário, poderia ser escrito quase naturalmente como um modificador. Um rascunho rudimentar da função seria algo parecido com isso:

```
def incrementar(horario, segundos):
    horario.segundos = \
horario.segundos + segundos

    if horario.segundos >= 60:
        horario.segundos = \
horario.segundos - 60
        horario.minutos = horario.minutos + 1

    if horario.minutos >= 60:
        horario.minutos = \
horario.minutos - 60
        horario.horas = horario.horas + 1
```

A primeira linha executa a operação básica; o resto lida com os casos especiais que vimos antes.

Esta função está correta? O que aconteceria se o parâmetro segundos for muito maior que sessenta? Nesse caso, não é suficiente transportar apenas uma vez; teríamos de continuar fazendo isso até que segundos seja menor que sessenta. Uma solução seria substituir os comando if por comandos while:

```
def incrementar(horario, segundos):
    horario.segundos = \
horario.segundos + segundos

    while horario.segundos >= 60:
        horario.segundos = horario.segundos -
60
        horario.minutos = horario.minutos + 1

    while horario.minutos >= 60:
        horario.minutos = \
horario.minutos - 60
        horario.horas = horario.horas + 1
```

Esta função agora está correta, mas não é a solução mais eficiente.

Como um exercício, reescreva esta função de maneira que ela não contenha nenhum loop. Como um segundo exercício, reescreva incrementar como uma função pura, e escreva chamadas de funções para as duas funções. [Clique aqui para feedback](#)

### 13.4 O que é melhor ?

Qualquer coisa que pode ser feita com modificadores também podem ser feitas com funções puras. De fato, algumas linguagens de programação permitem apenas funções puras. Existe alguma evidência que programas que usam funções puras são desenvolvidos mais rapidamente e são menos propensos a erros que programas que usam modificadores. No entanto, modificadores às vezes são convenientes, e em alguns casos, programação funcional é menos eficiente.

Em geral, recomendamos que você escreva funções puras sempre que for necessário e recorrer para modificadores somente se existir uma grande vantagem. Esta aproximação poderia ser chamada de um estilo de programação funcional. [Clique aqui para feedback](#)

### 13.5 Desenvolvimento Prototipado versus Desenvolvimento Planejado

Neste capítulo, demonstramos uma aproximação para o desenvolvimento de programas que chamamos de desenvolvimento prototipado. Em cada caso, escrevemos um rascunho rudimentar (ou protótipo) que executou os cálculos básicos e então, o testamos em uns poucos casos, corrigindo as falhas que fomos encontrando.

Embora esta aproximação possa ser eficaz, ela pode conduzir a código que é desnecessariamente complicado pois trata de muitos casos especiais e não é confiável, pois é difícil saber se você encontrou todos os erros.

Uma alternativa é o desenvolvimento planejado, onde uma visão de alto nível do problema pode tornar a codificação muito mais fácil. Nesse caso, a abstração é que um objeto Horário é realmente um número de três dígitos na base 60! O componente segundo é a "coluna dos uns", o componente minuto é a "coluna do 60", e o componente hora é a "coluna do 360".

Quando nós escrevemos somaHorario e incrementar, nós estávamos efetivamente fazendo adições em base 60, que é o motivo pelo qual tínhamos de transportar de uma coluna para a próxima.

Essa observação sugere uma outra aproximação para todo o problema - nós podemos converter um objeto Horário em um número simples e levar vantagem do fato de que um computador sabe como fazer aritmética com números. A seguinte função converte o objeto Horário em um inteiro:

```
def converterParaSegundos(t):
    minutos = t.horas * 60 + t.minutos
    segundos = minutos * 60 + t.segundos
    return segundos
```

Agora, tudo que precisamos é uma maneira de converter de um inteiro para um objeto Horário:

```
def criarHorario(segundos):
    horario = Time()
    horario.horas = segundos/3600
    segundos = segundos - horario.horas * 3600
    horario.minutos = segundos/60
    segundos = segundos - horario.minutos * 60
    horario.segundos = segundos
    return horario
```

Você deve ter que pensar um pouco para se convencer que esta técnica de converter de uma base para outra é correta. Assumindo que você está convencido, você pode usar essas funções para reescrever somaHorario:

```
def somaHorario(t1, t2):
    segundos = \
converterParaSegundos(t1) + converterParaSegundos(t2)
    return criarHorario(segundos)
```

Esta versão é muito mais curta que a original, e é muito mais fácil para demonstrar que está correta (assumindo, como sempre, que as funções que são chamadas estão corretas).

Como um exercício, reescreva incrementar da mesma forma. [Clique aqui para feedback](#)

### 13.6 Generalização

Algumas vezes, converter de base 60 para base 10 e voltar é mais difícil do que simplesmente lidar com horários.



Conversão de base é mais abstrata; nossa intuição para lidar com horários é melhor.

Mas se conseguirmos abstrair horários como números de base 60 e investirmos em criar as funções de conversão (`converterParaSegundos` e `criarHorario`), nós conseguimos um programa que é menor, fácil de ler e depurar, e mais confiável.

É também fácil para adicionar funcionalidades depois. Por exemplo, imagine subtrair dois `Horarios` para encontrar a duração entre eles. Uma aproximação ingênua seria implementar subtração com empréstimo. Usando as funções de conversão será mais fácil e provavelmente estará correto.

Ironicamente, algumas vezes fazer um problema mais difícil (ou mais genérico) o torna mais simples (porque existem alguns poucos casos especiais e poucas oportunidades para errar). Clique aqui para feedback

### 13.7 Algoritmos

Quando você escreve uma solução genérica para uma classe de problemas, ao contrário de uma solução específica para um único problema, você escreveu um algoritmo. Nós mencionamos isso antes mas não definimos cuidadosamente. Isso não é fácil para definir, então vamos tentar definir demonstrando algumas situações.

Primeiramente, considere alguma coisa que não seja um algoritmo. Quando você aprendeu a multiplicar números de um dígito, você provavelmente memorizou a tabela de multiplicação. Como resultado, você memorizou 100 soluções específicas. Esse tipo de conhecimento não é algoritmo.

Mas se você é "preguiçoso", você provavelmente trapaceou por ter aprendido alguns truques. Por exemplo, para encontrar o produto de  $n$  e 9, você pode escrever  $n-1$  como o primeiro dígito e  $10-n$  como o segundo dígito. Este truque é um solução genérica para multiplicar qualquer número de um dígito por 9. Isso é um algoritmo!

De modo parecido, as técnicas que você aprendeu para somar com transporte, subtração com empréstimo, e divisão longa são todas algoritmos. Uma das características dos algoritmos é que eles não requerem nenhuma inteligência para serem executados. Eles são processos mecânicos no qual cada passo segue o último de acordo com um conjunto simples de regras.

Na nossa opinião, é preocupante que humanos gastem tanto tempo na escola aprendendo a executar algoritmos que, literalmente, não requerem inteligência.

Por outro lado, o processo de projetar algoritmos é interessante, intelectualmente desafiante, e uma parte central daquilo que chamamos programação.

Algumas das coisas que as pessoas fazem naturalmente, sem dificuldade ou consciência, são as mais difíceis de se expressar através de algoritmos. Entender a linguagem natural é um bom exemplo. Todos nós fazemos isso, mas até hoje ninguém conseguiu explicar como fazemos isso, pelo menos não na forma de algoritmo. Clique aqui para feedback.

### 13.8 Glossário

<b>função pura (pure function)</b>	Uma função que não modifica nenhum dos objetos que ela recebe como parâmetro. A maioria das funções puras é frutífera.
<b>modificador (modifier)</b>	Uma função que muda um ou mais dos objetos que ela recebe como parâmetros. A maioria dos modificadores é nula.
<b>estilo de programação funcional (functional programming style)</b>	Um estilo de programação onde a maioria das funções são puras.
<b>desenvolvimento prototipado (prototype development)</b>	Uma maneira de desenvolver programas começando com um protótipo e gradualmente melhorando-o.
<b>desenvolvimento planejado (planned development)</b>	Uma maneira de desenvolver programas que envolvem uma percepção de alto nível do problema e mais planejamento do que desenvolvimento incremental ou desenvolvimento prototipado.
<b>algoritmo (algorithm)</b>	Um conjunto de instruções para resolver uma classe de problemas usando um processo mecânico, não inteligente.

## Capítulo 14: Classes e métodos

**ATENÇÃO** As referências cruzadas a nomes em códigos de outros capítulos (especialmente 13) ainda não foram unificadas...

### 14.1 Características da orientação a objetos

Python é uma **linguagem de programação orientada a objetos**, o que significa que ela tem características que suportam a **programação orientada a objetos**.

Não é fácil definir programação orientada a objetos, mas temos visto already algumas de suas características:

- Programas são construídos sobre definições de objetos e definições de funções, e a maioria das computações é expressa em termos de operações sobre objetos.
- Cada definição de objeto corresponde a algum objeto ou conceito do mundo real, e as funções que operam com aqueles objetos correspondem à maneira como os objetos do mundo real interagem.

Por exemplo, a classe Tempo, definida no capítulo 13 corresponde à maneira como as pessoas registram as horas do dia, e as funções que definimos correspondem aos tipos de coisas que as pessoas fazem com times. Do mesmo modo, as classes Ponto e Retângulo correspondem aos conceitos matemáticos de um ponto e de um retângulo.

Até aqui, não tiramos vantagem das características fornecidas por Python que suportam a programação orientada a objetos. Estritamente falando, estas características não são necessárias. Na maior parte das vezes, elas fornecem uma sintaxe alternativa para as coisas que já fizemos, mas em muitos casos, a alternativa é mais concisa e convém mais acuradamente à estrutura do programa.

Por exemplo, no programa Time, não existe uma conexão óbvia entre a definição da classe e a definição da função que segue. Com alguma investigação, fica aparente que toda função toma pelo menos um objeto Time como um parâmetro.

Esta observação é a motivação por trás dos métodos. Já temos visto alguns métodos, tais como keys (chaves) e values (valores), os quais foram invocados em dicionários. Cada método é associado com uma classe e é intended para ser invocado em instâncias daquela classe.

Métodos são simplesmente como funções, com duas diferenças:

- Métodos são definidos dentro da definição de uma classe para tornar explícita a relação entre a classe e o método.
- A sintaxe para a chamada do método é diferente da sintaxe para a chamada de uma função.

Nas próximas seções, vamos pegar as funções dos dois capítulos anteriores e transformá-las em métodos. Esta transformação é puramente mecânica: você pode conseguí-la simplesmente seguindo uma seqüência de passos. Se você se sentir confortável convertendo de uma forma para a outra, você estará apto para escolher a melhor forma para seja o lá o que for que você estiver fazendo.

### 14.2 exhibeHora (printTime)

No capítulo 13, definimos uma classe chamada Horário (Time) e você escreveu uma função chamada exhibeHora (printTime), que deve ter ficado mais ou menos assim:

```
class Horário:
    pass

def exhibeHora(time)
    print str(time.horas) + '?:? + \
        str(time.minutos) + '?:? + \
        str(time.segundos)
```

Para chamar esta função, passamos um objeto Time como um parâmetro:

```
>>> horaCorrente = Hora()
>>> horaCorrente.horas = 9
>>> horaCorrente.minutos = 14
>>> horaCorrente.segundos = 30
>>> exhibeHora(horaCorrente)
```

Para fazer de exhibeHora um método, tudo o que temos a fazer é mover a definição da função para dentro da definição da classe. Note a mudança na endentação:

```
class Horário:
    def exhibeHora(time):
        print str(time.horas) + '?:? + \
            str(time.minutos) + '?:? + \
            str(time.segundos)
```

Agora podemos chamar exhibeHora usando a natação de ponto:

```
>>> horaCorrente.exibeHora()
```

Como é usual, o objeto no qual o método é invocado aparece antes do ponto e o nome do método aparece depois do ponto.

O objeto no qual o método é invocado é atribuído ao primeiro parâmetro, então, neste caso, horaCorrente é atribuído ao parâmetro time.

Por convenção, o primeiro parâmetro de um método é chamado self. A razão para isto é um pouco convoluted, mas é baseada numa metáfora útil.

A sintaxe para uma chamada de função, exhibeHora(horaCorrente), sugere que a função é um agente ativo. Diz algo como, 'Ei, exhibeHora! Aqui está um objeto para você exibir.'

Na programação orientada a objetos, os objetos são agentes ativos. Uma chamado do tipo horaCorrente.exibeHora() diz 'Ei, horaCorrente! Por favor exhiba-se a si mesmo!'

Esta mudança de perspectiva pode ser mais polida, mas não fica óbvio que seja útil. Nos exemplos que temos visto até aqui, pode ser que não seja. Mas às vezes, deslocar a responsabilidade das funções para cima dos objetos torna possível escrever funções mais versáteis, e torna mais fácil manter e reutilizar o código.

### 14.3 Um outro exemplo

Vamos converter incremento (da Seção 13.3) em um método. Para poupar espaço, deixaremos de fora métodos definidos previamente (anteriormente?), mas você deve mantê-los em sua versão:

```
class Time:
    #previous method definitions here...

    def increment(self, segundos):
        self.seconds = segundos + self.seconds

    while self.seconds >= 60:
        self.seconds = self.seconds - 60
        self.minutes = self.minutos + 1

    while self.minutes >= 60:
        self.minutes = self.minutos - 60
        self.hours = self.horas + 1
```

A transformação é puramente mecânica ? movemos a definição do método para dentro da definição da classe e mudamos o nome do primeiro parâmetro.

Agora podemos chamar incremento como um método:

```
horaCorrente.incremento(500)
```

De novo, o objeto no qual o método é chamado gets atribuí ao primeiro parâmetro, self. O segundo parâmetro, segundo toma(gets) o valor 500.

Como um exercício, converta ? converteParaSegundos? (da Seção 13.5) para um método na classe ?Time?.

<b>orientada a objetos</b>	tais como classes definidas pelo usuário e herança, que facilitam a programação orientada a objetos.
<b>programação orientada a objetos</b>	Um estilo de programação na qual os dados e as operações que os manipulam estão organizados em classes e métodos.
<b>método</b>	Uma função que é definida dentro de uma definição de classe e é chamada em instâncias desta classe.
<b>override (sem tradução; termo consagrado)</b>	Substituir uma definição já pronta. Exemplos incluem substituir um parâmetro padrão por um argumento particular e substituir um método padrão, fornecendo um novo método com o mesmo nome.
<b>método de inicialização (também chamado de construtor)</b>	Um método especial que é invocado automaticamente quando um novo objeto é criado e que inicializa os atributos deste objeto.
<b>sobrecarga de operador</b>	Estender a funcionalidade dos operadores nativos (+, -, *, >, <, etc.) de forma que eles funcionem também com tipos definidos pelo usuário.
<b>produto escalar</b>	Operação definida na álgebra linear que multiplica dois pontos (com coordenadas (x,y,z)) e retorna um valor numérico.
<b>multiplicação por escalar</b>	Operação definida na álgebra linear que multiplica cada uma das coordenadas de um ponto por um valor numérico.
<b>polimórfica</b>	Uma função que pode operar com mais de um tipo. Se todas as operações de uma função pode ser aplicadas a um certo tipo, então a função pode ser aplicada a este tipo.

### 14.10 Glossário

<b>linguagem</b>	Uma linguagem que provê características
------------------	---

## Capítulo 15: Conjuntos de objetos

### 15.1 Composição

Até agora, você viu diversos exemplos de composição. Um dos primeiros exemplos foi o uso de uma invocação de método como parte de uma expressão. Outro exemplo é a estrutura aninhada dos comandos: você pode pôr um comando `if` dentro de um laço `while`, dentro de outro comando `if`, e assim por diante.

Tendo visto este padrão, e tendo aprendido a respeito de listas e objetos, você não deveria ficar surpreso em aprender que você pode criar listas de objetos. Você também pode criar objetos que contêm listas (como atributos); você pode criar listas que contêm listas; você pode criar objetos que contêm objetos; e assim por diante.

Neste capítulo e no próximo, você irá ver alguns exemplos destas combinações, usando objetos `Carta` como exemplo.

### 15.2 Objetos Carta

Se você não estiver familiarizado com jogos de cartas, agora é um bom momento para conseguir um baralho, ou então esse capítulo pode não fazer muito sentido. Há 52 cartas em um baralho, cada uma das quais pertence a um dos quatro naipes e a uma das treze posições. Os naipes são Espadas, Copas, Ouros e Paus (em ordem descendente no bridge). As posições são As, 2, 3, 4, 5, 6, 7, 8, 9, 10, Valete, Rainha e Rei. Dependendo do jogo, a posição do As pode ser maior do que a do Rei ou menor do que a do 2.

Se quisermos definir um novo objeto para representar uma carta, é óbvio que os atributos devem ser `posicao` e `naipe`. Não tão óbvio são os tipos aos quais devem pertencer os atributos. Uma possibilidade é usar strings contendo palavras como "Espada" para naipes e "Rainha" para posições. Um problema com esta implementação é que não seria fácil comparar cartas para ver qual possui o maior naipe ou posição.

Uma alternativa é usar inteiros para **codificar** as posições e naipes. "Codificar", neste caso, não significa o mesmo que as pessoas normalmente pensam, que é criptografar ou traduzir para um código secreto. O que um cientista da computação quer dizer com "codificar" é "definir um mapeamento entre uma seqüência de números e os itens que eu quero representar". Por exemplo:

- Espadas -> 3
- Copas -> 2
- Ouros -> 1
- Paus -> 0

Uma característica óbvia deste mapeamento é que os naipes são mapeados para inteiros na ordem, de modo que nós podemos comparar naipes pela comparação de inteiros. O mapeamento de posições é bastante óbvio. Cada uma das posições numéricas mapeia para o inteiro correspondente e, as cartas com figura são mapeadas conforme abaixo:

- Valete -> 11

- Rainha -> 12
- Rei -> 13

O motivo pelo qual nós estamos usando notação matemática para estes mapeamentos é que eles não são parte do programa Python. Eles são parte do projeto do programa, mas eles nunca aparecem explicitamente no código. A definição de classe para o tipo `Carta` fica parecida com esta:

```
class Carta:
    def __init__(self, naipe=0, posicao=0):
        self.naipe = naipe
        self.posicao = posicao
```

Como sempre, nós fornecemos um método de inicialização que recebe um parâmetro opcional para cada atributo.

Para criar um objeto que representa o 3 de Paus, usa-se este comando:

```
tresDePaus = Carta(0, 3)
```

O primeiro argumento, 0, representa o naipe de Paus.

### 15.3 Atributos de classe e o método `__str__`

Para imprimir objetos `Carta` de uma maneira que as pessoas possam facilmente ler, nós gostaríamos de mapear os códigos inteiros para palavras. Uma forma natural de fazer isso é usar listas de strings. Nós atribuímos estas listas para **atributos de classe** no topo da definição de classe:

```
class Carta:
    listaDeNaipes = ["Paus", "Ouros", "Copas",
                    "Espadas"]
    listaDePosicoes = ["narf", "As", "2", "3", "4",
                      "5", "6", "7",
                      "8", "9", "10", "Valete",
                      "Rainha", "Rei"]
    # método init omitido
    def __str__(self):
        return (self.listaDePosicoes[self.posicao] +
                " de " + self.listaDeNaipes[self.naipe])
```

Um atributo de classe é definido fora de qualquer método, e ele pode ser acessado por quaisquer métodos da classe.

Dentro de `__str__`, nós podemos usar `listaDeNaipes` e `listaDePosicoes` para mapear os valores numéricos de `naipe` e `posicao` para strings. Por exemplo, a expressão `self.listaDeNaipes[self.naipe]` significa "use o atributo `naipe` do objeto `self` como um índice para o atributo de classe chamado `listaDeNaipes`, e selecione a string apropriada".

O motivo para o "narf" no primeiro elemento em `listaDePosicoes` é preencher o lugar do 0-ésimo elemento da lista, que nunca será usado. As únicas posições válidas são de 1 a 13. Este item desperdiçado não é inteiramente necessário. Nós poderíamos ter iniciado com 0, como é normal. Porém, é menos confuso codificar 2 como 2, 3 como

3, e assim por diante.

Com os métodos que nós temos até agora, nós podemos criar e imprimir cartas:

```
>>> carta1 = Carta(1, 11)
>>> print carta1
Valete de Ouros
```

Atributos de classe como `listaDeNaipes` são compartilhados por todos os objetos `Carta`. A vantagem disso é que nós podemos usar qualquer objeto `Carta` para acessar os atributos de classe:

```
>>> carta2 = Carta(1, 3)
>>> print carta2
3 de Ouros
>>> print carta2.listaDeNaipes[1]
Ouros
```

A desvantagem é que se nós modificarmos um atributo de classe, isso afetará cada instância da classe. Por exemplo, se nós decidirmos que "Valete de Ouros" deveria realmente se chamar "Valete de Baleias Rodopiantes", nós poderíamos fazer isso:

```
>>> carta1.listaDeNaipes = "Baleias Rodopiantes"
>>> print carta1
3 de Baleias Rodopiantes
```

O problema é que todos os Ouros se tornam Baleias Rodopiantes:

```
>>> print carta2
3 de Baleias Rodopiantes
```

Normalmente, não é uma boa idéia modificar atributos de classe.

### 15.4 Comparando cartas

Para tipos primitivos, existem operadores condicionais (`<`, `>`, `==`, etc.) que comparam valores e determinam quando um é maior que, menor que ou igual a outro. Para tipos definidos pelo usuário, nós podemos sobrescrever o comportamento dos operadores pré-definidos fornecendo um método `__cmp__`. Por convenção, `__cmp__` recebe dois parâmetros, `self` e `other`, e retorna 1 se o primeiro objeto for maior, -1 se o segundo objeto for maior, e 0 se eles forem iguais.

Alguns tipos são totalmente ordenados, o que significa que nós podemos comparar quaisquer dois elementos e dizer qual é o maior. Por exemplo, os inteiros e os números de ponto flutuante são totalmente ordenados. Alguns conjuntos são não-ordenados, o que significa que não existe maneira significativa de dizer que um elemento é maior que o outro. Por exemplo, as frutas são não-ordenadas, e é por isso que não podemos comparar maçãs e laranjas.

O conjunto de cartas de jogo é parcialmente ordenado, o que significa que às vezes você pode comparar cartas, e às vezes não. Por exemplo, você sabe que o 3 de Paus é maior do que o 2 de Paus, e que o 3 de Ouros é maior do que o 3 de Paus. Mas qual é o melhor, o 3 de Paus ou o 2 de Ouros? Um tem uma posição maior, mas o outro tem um naipe maior.

Para tornar as cartas comparáveis, você tem que decidir o que é mais importante: posição ou naipe. Para ser honesto, a escolha é arbitrária. Por questão de escolha, nós iremos dizer que naipe é mais importante, porque um baralho de cartas novo vem ordenado com todas as cartas de Paus juntas, seguidas pelas de Ouros, e assim por diante.

Com essa decisão, nós podemos escrever `__cmp__`:

```
def __cmp__(self, other):
```

```
# verificar os naipes
if self.naipe > other.naipe: return 1
if self.naipe < other.naipe: return -1
# as cartas têm o mesmo naipe... verificar as
posições
if self.posicao > other.posicao: return 1
if self.posicao < other.posicao: return -1
# as posições são iguais... é um empate
return 0
```

Nesta ordenação, Ases são menores do que 2.

Como um exercício, modifique `__cmp__`, de modo que os Ases sejam maiores do que os Reis.

### 15.5 Baralhos

Agora que nós temos objetos para representar Cartas, o próximo passo lógico é definir uma classe para representar um Baralho. É claro que um baralho é formado por cartas; portanto, cada objeto Baralho irá conter uma lista de cartas como um atributo.

A seguir, damos uma definição para a classe Baralho. O método de inicialização cria o atributo `cartas` e gera o conjunto padrão de 52 cartas:

```
class Baralho:
    def __init__(self):
        self.cartas = []
        for naipe in range(4):
            for posicao in range(1, 14):
                self.cartas.append(Carta(naipe, posicao))
```

A maneira mais fácil de popular o baralho é com um laço aninhado. O laço externo enumera os naipes de 0 até 3. O laço interno enumera as posições de 1 até 13. Como o laço externo repete quatro vezes e o laço interno 13 vezes, o número total de vezes que o corpo é executado é 52 (13 vezes quatro). Cada iteração cria uma nova instância de `Carta` com o naipe e posição atuais e a inclui na lista `cartas`.

O método `append` trabalha sobre listas mas não, obviamente, sobre tuplas.

### 15.6 Imprimindo o baralho

Como sempre, quando nós definimos um novo tipo de objeto, nós gostaríamos de ter um método para imprimir o conteúdo de um objeto. Para imprimir um Baralho, nós percorremos a lista e imprimimos cada `Carta`:

```
class Baralho:
    ...
    def imprimirBaralho(self):
        for carta in self.cartas:
            print carta
```

Aqui, e a partir daqui, as reticências (...) indicam que nós omitimos os outros métodos da classe.

Como uma alternativa a `imprimirBaralho`, nós poderíamos escrever um método `__str__` para a classe Baralho. A vantagem de `__str__` é que ela é mais flexível. Em vez de apenas imprimir o conteúdo de um objeto, ela gera uma representação em string que outras partes do programa podem manipular antes de imprimir ou armazenar para uso posterior.

Abaixo, uma versão de `__str__` que devolve uma representação em string de um Baralho. Para adicionar um pouco de estilo, ela distribui as cartas em uma cascata, na qual cada carta é indentada um espaço a mais do que a carta anterior:



```
class Baralho:
    ...
    def __str__(self):
        s = ""
        for i in range(len(self.cartas)):
            s = s + " " * i + str(self.cartas[i]) + "\n"
        return s
```

Este exemplo demonstra diversas características. Primeiro, em vez de percorrer `self.cartas` e atribuir cada carta a uma variável, nós estamos usando `i` como uma variável de laço e um índice para a lista de cartas.

Segundo, nós estamos usando o operador de multiplicação de strings para indentar cada carta com um espaço adicional com relação à anterior. A expressão " " \* `i` produz um número de espaços igual ao valor atual de `i`.

Terceiro, em vez de usar o comando `print` para imprimir as cartas, nós usamos a função `str`. Passar um objeto como um argumento para `str` equivale a invocar o método `__str__` sobre o objeto.

Finalmente, nós estamos usando a variável `s` como um **acumulador**. Inicialmente, `s` é a string vazia. A cada repetição do laço, uma nova string é gerada e concatenada com o valor antigo de `s` para obter um novo valor. Quando o laço termina, `s` contém a representação em string completa do Baralho, que se parece com:

```
>>> baralho = Baralho()
>>> print Baralho
As de Paus
 2 de Paus
 3 de Paus
 4 de Paus
 5 de Paus
 6 de Paus
 7 de Paus
 8 de Paus
 9 de Paus
10 de Paus
Valete de Paus
Rainha de Paus
Rei de Paus
  As de Ouros
```

E assim por diante. Mesmo que o resultado apareça em 52 linhas, é uma string longa que contém newlines.

## 15.7 Embaralhando

Se um baralho estiver perfeitamente embaralhado, então cada carta tem a mesma probabilidade de aparecer em qualquer lugar no baralho, e qualquer localização no baralho tem a mesma probabilidade de conter qualquer carta.

Para embaralhar as cartas, nós usaremos a função `randrange` do módulo `random`. Com dois argumentos inteiros, `a` e `b`, `randrange` escolhe um inteiro aleatório no intervalo  $a \leq x < b$ . Como o limite superior é estritamente menor que `b`, nós podemos usar o comprimento de uma lista como o segundo parâmetro, e nós garantimos que o índice sempre será válido. Por exemplo, esta expressão escolhe o índice de uma carta aleatória em um baralho:

```
random.randrange(0, len(self.cartas))
```

Uma maneira fácil de embaralhar as cartas é percorrer a lista e trocar cada carta por outra escolhida aleatoriamente. É possível que a carta seja trocada por ela mesma, mas isso não é problema. Na verdade, se nós excluíssemos essa possibilidade, a ordem das cartas não seria

totalmente aleatória:

```
class Baralho:
    ...
    def embaralhar(self):
        import random
        nCartas = len(self.cartas)
        for i in range(nCartas):
            j = random.randrange(i, nCartas)
            self.cartas[i], self.cartas[j] =
self.cartas[j], self.cartas[i]
```

Em vez de assumir que existem 52 cartas no baralho, nós obtivemos o comprimento real da lista e o guardamos na variável `nCartas`.

Para cada carta no baralho, nós escolhemos uma carta aleatória dentre as cartas que ainda não foram embaralhadas. Então, nós trocamos a carta atual (`i`) pela carta selecionada (`j`). Para trocar as cartas, nós usamos uma atribuição de tupla, como visto na Seção 9.2:

```
self.cartas[i], self.cartas[j] = self.cartas[j],
self.cartas[i]
```

Como exercício, reescreva esta linha de código sem usar uma atribuição de seqüência.

## 15.8 Removendo e distribuindo cartas

Outro método que pode ser útil para a classe Baralho é `removeCarta`. Ele recebe uma carta como parâmetro, remove-a do baralho e retorna verdadeiro (1), se a carta estava no baralho e falso (0), caso contrário:

```
class Baralho:
    ...
    def removeCarta(self, carta):
        if carta in self.cartas:
            self.cartas.remove(carta)
            return 1
        else:
            return 0
```

O operador `in` retorna verdadeiro se o primeiro operando estiver contido no segundo, que deve ser uma lista ou uma tupla. Se o primeiro operando for um objeto, Python usa o método `__cmp__` do objeto para determinar igualdade com os itens da lista. Como o método `__cmp__` da classe `Carta` verifica por igualdade profunda, o método `removeCarta` também testa por igualdade profunda.

Para distribuir as cartas, nós iremos remover e devolver a carta do topo. O método de lista `pop` fornece uma maneira conveniente de fazer isso:

```
class Baralho:
    ...
    def distribuirCarta(self):
        return self.cards.pop()
```

Na verdade, `pop` remove a última carta da lista. Portanto, nós estamos realmente distribuindo as cartas do fim para o início do baralho.

Uma última operação que nós poderíamos querer é a função booleana `estahVazio`, que retorna verdadeiro se o baralho não contém cartas:

```
class Baralho:
    ...
    def estahVazio(self):
        return (len(self.cartas) == 0)
```

### 15.9 Glossário

---

<b>codificar (encode)</b>	Representar um conjunto de valores usando outro conjunto de valores, construindo um mapeamento entre eles.
<b>atributo de classe (class attribute)</b>	Uma variável que é definida dentro de uma definição de classe, mas fora de qualquer método. Atributos de classe podem ser acessados a partir de qualquer método da classe e são compartilhados por todas as instâncias da classe.
<b>acumulador (accumulator)</b>	Uma variável usada em um laço para acumular uma série de valores, para, por exemplo, concatená-los em uma string ou somá-los a uma soma em andamento.

## Capítulo 16: Herança

### 16.1 Herança

Uma das características mais marcantes das linguagens orientadas a objetos é a **herança**. Herança é a habilidade de definir uma nova classe que é uma versão modificada de uma classe existente.

A principal vantagem dessa característica é que você pode adicionar novos métodos a uma classe sem ter que modificar a classe existente. Chama-se "herança" porque a nova classe herda todos os métodos da classe existente. Ampliando a metáfora, podemos dizer que a classe existente é às vezes chamada de classe **mãe** (parent). A nova classe pode ser chamada de classe **filha** ou, simplesmente, "subclasse".

A herança é uma característica poderosa. Alguns programas que seriam complicados sem herança podem ser escritos de forma simples e concisa graças a ela. E a herança também pode facilitar o reuso do código, uma vez que você pode adaptar o comportamento de classes existentes sem ter que modificá-las. Em alguns casos, a estrutura da herança reflete a natureza real do problema, tornando o programa mais fácil de entender.

Por outro lado, a herança pode tornar um programa seja difícil de ler. Quando um método é invocado, às vezes não está claro onde procurar sua definição. A parte relevante do código pode ser espalhada em vários módulos. E, também, muitas das coisas que podem ser feitas utilizando herança também podem ser feitas de forma igualmente elegante (ou até mais) sem ela. Se a estrutura natural do problema não se presta a utilizar herança, esse estilo de programação pode trazer mais problemas que vantagens.

Nesse capítulo, vamos demonstrar o uso de herança como parte de um programa que joga uma variante de Mico. Um dos nossos objetivos é escrever um código que possa ser reutilizado para implementar outros jogos de cartas.

### 16.2 Uma mão de cartas

Para quase todos os jogos de baralho, é preciso representar uma mão de cartas. Uma mão de cartas é similar a um maço de baralho. Porque ambos são formados por uma série de cartas e ambos requerem operações, como, adicionar e remover cartas. Fora isso, a habilidade de embaralhar a mão e o baralho também são úteis.

Mas, ao mesmo tempo, a mão é também diferente do baralho. Dependendo do jogo que está sendo jogado, precisamos realizar algumas operações nas mãos de cartas que não fazem sentido para o baralho inteiro. Por exemplo, no pôquer, podemos classificar uma mão (trinca, flush, etc.) ou compará-la com outra mão. No jogo de bridge, podemos querer computar a quantidade de pontos que há numa mão, a fim de fazer um lance.

Essa situação sugere o uso de herança. Se Mao é uma subclasse de Baralho, terá todos os métodos de Baralho, e novos métodos podem ser adicionados.

Na definição de classe, o nome da classe pai aparece entre parênteses:

```
class Mao(Baralho):
    pass
```

Esse comando indica que a nova classe Mao herda da classe existente Baralho.

O construtor de Mao inicializa os atributos da mão, que são nome e cartas. A string nome identifica essa mão, provavelmente pelo nome do jogador que está segurando as cartas. O nome é um parâmetro opcional com a string vazia como valor default. cartas é a lista de cartas da mão, inicializada com uma lista vazia

```
class Mao(Baralho):
    def __init__(self, nome=""):
        self.cartas = []
        self.nome = nome
```

Em praticamente todos os jogos de cartas, é necessário adicionar e remover cartas do baralho. Remover cartas já está resolvido, uma vez que Mao herda removerCarta de Baralho. Mas precisamos escrever adicionarCarta:

```
class Mao(Baralho):
    #...
    def adicionarCarta(self, carta):
        self.cartas.append(carta)
```

De novo, a elipse indica que omitimos outros métodos. O método de listas append adiciona a nova carta no final da lista de cartas.

### 16.3 Dando as cartas

Agora que temos uma classe Mao, queremos distribuir cartas de Baralho para mãos de cartas. Não é imediatamente óbvio se esse método deve ir na classe Mao ou na classe Baralho, mas como ele opera num único baralho e (possivelmente) em várias mãos de cartas, é mais natural colocá-lo em Baralho.

O método distribuir deve ser bem geral, já que diferentes jogos terão diferentes requerimentos. Podemos querer distribuir o baralho inteiro de uma vez só ou adicionar uma carta a cada mão.

distribuir recebe dois argumentos, uma lista (ou tupla) de mãos e o número total de cartas a serem dadas. Se não houver cartas suficientes no baralho, o método dá todas as cartas e pára:

```
class Baralho:
    #...
    def distribuir(self, maos, nCartas=999):
        nMaos = len(maos)
        for i in range(nCartas):
            if self.estaVazia(): break
            # interromper se acabaram as cartas
            carta = self.pegarCarta()
            # pegar a carta do topo
            mao = maos[i % nMaos]
            # quem deve receber agora?
            mao.adicionarCarta(carta)
            # adicionar a carta à mao
```

O segundo parâmetro, nCartas, é opcional; o default é um número grande, o que na prática significa que todas as cartas do baralho serão dadas se este parâmetro for omitido.

A variável do laço `i` vai de 0 a `nCartas-1`. A cada volta do laço, uma carta é removida do baralho, usando o método de lista `pop`, que remove e retorna o último item na lista.

O operador módulo (%) permite dar cartas em ao redor da mesa (uma carta de cada vez para cada mão). Quando `i` é igual ao número de mãos na lista, a expressão `i % nMaos` volta para o começo da lista (índice 0).

### 16.4 Exibindo a mão

Para exibir o conteúdo de uma mão, podemos tirar vantagem dos métodos `exibirBaralho` e `__str__` herdados de `Baralho`. Por exemplo:

```
>>> baralho = Baralho()
>>> baralho.embaralhar()
>>> mao = Mao("fabio")
>>> baralho.distribuir([mao], 5)
>>> print mao
Mão fabio contém
2 de espadas
3 de espadas
4 de espadas
  Ás de copas
  9 de paus
```

Não é lá uma grande mão, mas tem potencial para um straight flush.

Embora seja conveniente herdar os métodos existentes, há outras informacoes num objeto `Mao` que podemos querer incluir quando ao exibí-lo. Para fazer isso, podemos fornecer um método `__str__` para a classe `Mao` que sobrescreva o da classe `Baralho`:

```
class Mao(Baralho)
#...
def __str__(self):
    s = "Mao " + self.nome
    if self.estahVazia():
        return s + " está vazia\n"
    else:
        return s + " contém\n" + Baralho.__str__(self)
```

Inicialmente, `s` é uma string que identifica a mão. Se a mão estiver vazia, o programa acrescenta as palavras está vazia e retorna o resultado.

Se não, o programa acrescenta a palavra contém e a representação de string do `Baralho`, computada pela invocação do método `__str__` na classe `Baralho` em `self`.

Pode parecer estranho enviar `self`, que se refere à `Mao` corrente, para um método `Baralho`, mas isso só até voce se lembrar que um `Mao` é um tipo de `Baralho`. Objetos `Mao` podem fazer tudo que os objetos `Baralho` fazem, então, é permitido passar uma instância de `Mao` para um método `Baralho`.

Em geral, sempre é permitido usar uma instância de uma subclasse no lugar de uma instância de uma classe mãe.

### 16.5 A classe `JogoDeCartas`

A classe `JogoDeCartas` toma conta de algumas tarefas básicas comuns a todos os jogos, como, criar o `baralho` e `embaralhá-lo`:

```
class JogoDeCartas:
```

```
def __init__(self):
    self.baralho = Baralho()
    self.baralho.embaralhar()
```

Este é o primeiro dos casos que vimos até agora em que o método de inicialização realiza uma computação significativa, para além de inicializar atributos.

Para implementar jogos específicos, podemos herdar de `JogoDeCartas` e adicionar características para o novo jogo. Como exemplo, vamos escrever uma simulação de `Mico`.

O objetivo do jogo é livrar-se das cartas que estiverem na mão. Para fazer isso, é preciso combinar cartas formando pares ou casais que tenham a mesma cor e o mesmo número ou figura. Por exemplo, o 4 de paus casa com o 4 de espadas porque os dois naipes são pretos. O Valete de copas combina com o Valete de ouros porque ambos são vermelhos.

Antes de mais nada, a Dama de paus é removida do baralho, para que a Dama de espadas fique sem par. A Dama de espadas então faz o papel do mico. As 51 cartas que sobram são distribuídas aos jogadores em ao redor da mesa (uma carta de cada vez para cada mão). Depois que as cartas foram dadas, os jogadores devem fazer todos os casais possíveis que tiverem na mão, e em seguida descartá-los na mesa.

Quando ninguém mais tiver nenhum par para descartar, o jogo começa. Na sua vez de jogar, o jogador pega uma carta (sem olhar) do vizinho mais próximo à esquerda, que ainda tiver cartas. Se a carta escolhida casar com uma carta que ele tem na mão, ele descarta esse par. Quando todos os casais possíveis tiverem sido feitos, o jogador que tiver sobrado com a Dama de espadas na mão perde o jogo.

Em nossa simulação computacional do jogo, o computador joga todas as mãos. Infelizmente, algumas nuances do jogo presencial se perdem. Num jogo presencial, o jogador que está com o mico na mão pode usar uns truques para induzir o vizinho a pegar a carta, por exemplo, segurando-a mais alto que as outras, ou mais baixo, ou se esforçando para que ela não fique em destaque. Já o computador simplesmente pega a carta do vizinho aleatoriamente...

### 16.6 Classe `MaoDeMico`

Uma mão para jogar `Mico` requer algumas habilidades para além das habilidades gerais de uma `Mao`. Vamos definir uma nova classe, `MaoDeMico`, que herda de `Mao` e provê um método adicional chamado `descartarCasais`:

```
class MaoDeMico(Mao):
def descartarCasais(self):
    conta = 0
    cartasIniciais = self.cartas[:]
    for carta in cartasIniciais:
        casal = Carta(3 - carta.naipes, carta.valor)
        if casal in self.cartas:
            self.cartas.remove(carta)
            self.cartas.remove(casal)
            print "Mao %s: %s casais %s" %
(self.nome, carta, casal)
            conta = conta + 1
    return conta
```

Começamos fazendo uma cópia da lista de cartas, para poder percorrer a cópia enquanto removemos cartas do original. Uma vez que `self.cartas` é modificada no laço, não queremos usá-la para controlar o percurso. Python pode ficar bem confuso se estiver percorrendo uma lista que está mudando!

Para cada carta na mão, verificamos qual é a carta que faz par com ela e vamos procurá-la. O par da carta tem o mesmo valor (número ou figura) e naipe da mesma cor. A expressão `3 - carta.naipe` transforma um paus (naipe 0) numa espadas (naipe 3) e um ouros (naipe 1) numa copas (naipe 2). Você deve analisar a fórmula até se convencer de que as operações opostas também funcionam. Se o par da carta também estiver na mão, ambas as cartas são removidas.

O exemplo a seguir demonstra como usar `descartarCasais`:

```
>>> jogo = JogoDeCartas()
>>> mao = MaoDeMico("fabio")
>>> jogo.baralho.distribuir([mao], 13)
>>> print mao
mão fabio contém
Âs de espadas
 2 de ouros
 7 de espadas
 8 de paus
 6 de copas
 8 de espadas
 7 de paus
  Rainha de paus
 7 de ouros
 5 de paus
  Valete de ouros
 10 de ouros
 10 de copas

>>> mao.descartarCasais()
Mão fabio: 7 de espadas faz par com 7 de paus
Mão fabio: 8 de espadas faz par com 8 de paus
Mão fabio: 10 de ouros faz par com 10 de copas
>>> print mao
Mão fabio contém
Âs de espadas
 2 de ouros
 6 de copas
  Rainha de paus
 7 de ouros
 5 de paus
  Valete de ouros
```

Observe que não existe um método `__init__` para a classe `MaoDeMico`. Ele é herdado de `Mao`.

### 16.7 Classe Mico

Agora podemos focar nossa atenção no jogo em si. `Mico` é uma subclasse de `JogoDeCartas` com um novo método chamado `jogar` que recebe uma lista de jogadores como argumento.

Já que `__init__` é herdado de `JogoDeCartas`, um novo objeto `Mico` contém um novo baralho embaralhado:

```
class Mico(JogoDeCartas):
    def jogar(self, nomes):
        # remover a Dama de paus
        self.baralho.removerCarta(Carta(0,12))

        # fazer uma mão para cada jogador
        self.maos = []
        for nome in nomes:
            self.maos.append(MaoDeMico(nome))
```

```
# distribuir as cartas
self.baralho.distribuir(self.maos)
print "----- As cartas foram dadas"
self.exibirMaos()

# remover casais iniciais
casais = self.removerTodosOsCasais()
print "----- Os pares foram descartados, o
jogo começa"
self.exibirMaos()

# jogar até que 25 casais se formem
vez = 0
numMaos = len(self.maos)
while casais < 25:
    casais = casais + self.jogarVez(vez)
    vez = (vez + 1) % numMaos

print "----- Fim do jogo"
self.exibirMaos()
```

Algumas etapas do jogo foram separadas em métodos. `removerTodosOsCasais` percorre a lista de mãos e invoca `descartarCasais` em cada uma:

```
class Mico(JogoDeCartas):
    #...
    def removerTodosOsCasais(self):
        conta = 0
        for mao in self.maos:
            conta = conta + mao.descartarCasais()
        return conta
```

Como exercício, escreva `exibirMaos` que percorre `self.maos` e exibe cada mão.

`conta` é um acumulador que soma o número de pares em cada mão e retorna o total.

Quando o número total de pares alcança 25, 50 cartas foram removidas das mãos, o que significa que sobrou só uma carta e o jogo chegou ao fim.

A variável `vez` mantém controle sobre de quem é a vez de jogar. Começa em 0 e incrementa de um em um; quando atinge `numMaos`, o operador módulo faz ela retornar para 0.

O método `jogarVez` recebe um argumento que indica de quem é a vez de jogar. O valor de retorno é o número de pares feitos durante essa rodada:

```
class Mico(JogoDeCartas):
    #...
    def jogarVez(self, i):
        if self.maos[i].estahVazia():
            return 0
        vizinho = self.buscarVizinho(i)
        novaCarta = self.maos[vizinho].pegarCarta()
        self.maos[i].adicionarCarta(novaCarta)
        print "Mao", self.maos[i].nome, "pegou",
        novaCarta
        conta = self.maos[i].descartarCasais()
        self.maos[i].embaralhar()
        return conta
```

Se a mão de um jogador estiver vazia, ele está fora do jogo, então, ele não faz nada e retorna 0.

Do contrário, uma jogada consiste em achar o primeiro jogador à esquerda que tenha cartas, pegar uma carta dele, e tentar fazer pares. Antes de retornar, as cartas na mão são embaralhadas, para que a escolha do próximo jogador seja aleatória.



O método `buscarVizinho` começa com o jogador imediatamente à esquerda e continua ao redor da mesa até encontrar um jogador que ainda tenha cartas:

```
class Mico(JogoDeCartas):
    #...
    def buscarVizinho(self, i):
        numMaos = len(self.maos)
        for next in range(1,numMaos):
            vizinho = (i + next) % numMaos
            if not self.maos[vizinho].estahVazia():
                return vizinho
```

Se `buscarVizinho` alguma vez circulasse pela mesa sem encontrar cartas, retornaria `None` e causaria um erro em outra parte do programa. Felizmente, podemos provar que isso nunca vai acontecer (desde que o fim do jogo seja detectado corretamente).

Não mencionamos o método `exibirBaralhos`. Esse você mesmo pode escrever.

A saída a seguir é produto de uma forma reduzida do jogo, onde apenas as 15 cartas mais altas do baralho (do 10 para cima) foram dadas, para três jogadores. Com esse baralho reduzido, a jogada pára depois que 7 combinações foram feitas, ao invés de 25:

```
>>> import cartas
>>> jogo = cartas.Mico()
>>> jogo.jogar(["Alice","Jair","Clara"])
----- As cartas foram dadas
Mão Alice contém
Rei de copas
Valete de paus
Rainha de espadas
Rei de espadas
10 de ouros

Mão Jair contém
Rainha de copas
Valete de espadas
Valete de copas
Rei de ouros
Rainha de ouros

Mão Clara contém
Valete of ouros
Rei de paus
10 de espadas
10 de copas
10 de paus

Mão Jair: Dama de copas faz par com Dama de ouros
Mão Clara: 10 de espadas faz par com 10 de paus
----- Os pares foram descartados, o jogo começa
Mão Alice contém
Rei de copas
Valete de paus
Rainha de espadas
```

```
Rei de espadas
10 de ouros

Mão Jair contém
Valete de espadas
Valete de copas
Rei de ouros

Mão Clara contém
Valete de ouros
Rei de paus
10 de copas

Mão Alice pegou o Rei de ouros
Mão Alice: Rei de copas faz par com Rei de ouros
Mão Jair pegou 10 de copas
Mão Clara pegou Valete de paus
Mão Alice pegou Valete de copas
Mão Jair pegou Valete de ouros
Mão Clara pegou Dama de espadas
Mão Alice pegou Valete de ouros
Mão Alice: Valete de copas faz par com Valete de
ouros
Mão Jair pegou Rei de paus
Mão Clara pegou Rei de espadas
Mão Alice pegou 10 de copas
Mão Alice: 10 de ouros faz par com 10 de copas
Mão Jair pegou Dama de espadas
Mão Clara pegou Valete de espadas
Mão Clara: Valete de paus faz par com Valete de
espadas
Mão Jair pegou Rei de espadas
Mão Jeff: Rei de paus faz par com Rei de espadas
----- Fim do jogo
Mão Alice está vazia

Mão Jair contém
Rainha de espadas

Mão Clara está vazia

Então, o Jair perdeu.
```

### 16.8 Glossário

<b>herança (inheritance)</b>	Habilidade de definir uma nova classe que é a versão modificada de uma classe definida anteriormente.
<b>classe mãe (parent class)</b>	A classe de quem a classe filha herda.
<b>classe filho (child class)</b>	Um nova classe criada herdando de uma classe existente; também chamada de "subclasse".

## Capítulo 17: Listas encadeadas

### 17.1 Referências Embutidas

Nós temos visto exemplos de atributos que referenciam outros objetos, que são chamados **referências embutidas** (veja a Seção 12.8). Uma estrutura de dados comum, a **lista ligada**, tira vantagem desta característica.

Listas ligadas são constituídas de **nós** (nodos), onde cada nó contém uma referência para o próximo nó na lista. Além disto, cada nó contém uma unidade de dados chamada a **carga**.

Uma lista ligada é considerada uma **estrutura de dados recorrente** porque ela tem uma definição recorrente.

Uma lista ligada é:

- Uma lista vazia, representada por None, ou
- Um nó que contém um objeto carga e uma referência para uma lista ligada.

Estruturas de dados recorrentes são adequadas para métodos recorrentes.

### 17.2 A classe No (Node)

Como é usual quando se escreve uma nova classe, nós começaremos com os métodos de inicialização e `__str__` de modo que podemos testar o mecanismo básico de se criar e mostrar o novo tipo:

```
class No:
    def __init__(self, carga=None, proximo=None):
        self.carga = carga
        self.proximo = proximo

    def __str__(self):
        return str(self.carga)
```

Como de costume, os parâmetros para o método de inicialização são opcionais. Por omissão (default), ambos, a carga e a ligação, proximo, são definidas como None.

A representação string de um nó é simplesmente a representação string da carga. Como qualquer valor pode ser passado para a função str, nós podemos armazenar qualquer valor em uma lista.

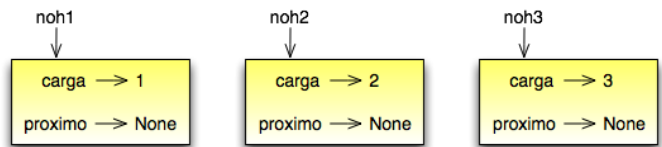
Para testar a implementação até agora, nós criamos um No e o imprimimos:

```
>>> no = No("teste")
>>> print no
teste
```

Para ficar interessante, nós precisamos uma lista com mais do que um nó:

```
>>> no1 = No(1)
>>> no2 = No(2)
>>> no3 = No(3)
```

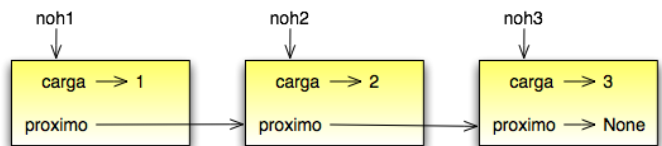
Este código cria três nós, mas nós ainda não temos uma lista ainda porque os nós não estão **ligados**. O diagrama de estado é parecido com este:



Para ligar os nós, temos que fazer o primeiro nó da lista referir ao segundo e o segundo nó referir ao terceiro:

```
>>> no1.proximo = no2
>>> no2.proximo = no3
```

A referência do terceiro nó é None, que indica que ele é o final da lista. Agora o diagrama de estado se parece com:



Agora você sabe como criar nós e ligá-los em uma lista. O que pode estar menos claro neste ponto é por quê.

### 17.3 Listas como Coleções

Listas são úteis porque elas provêm um modo de montar múltiplos objetos em uma única entidade, algumas vezes chamada uma coleção. No exemplo, o primeiro nó da lista serve como uma referência para toda a lista.

Para passar uma lista como um parâmetro, você apenas tem que passar uma referência ao primeiro nó. Por exemplo, a função `imprimeLista` toma um único nó como um argumento. Iniciando com o cabeça da lista, ela imprime cada nó até que chegue ao fim:

```
def imprimeLista(no):
    while no:
        print no,
        no = no.proximo
    print
```

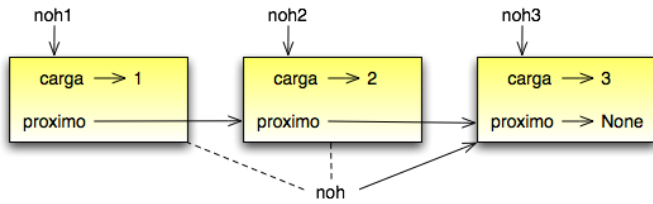
Para chamar este método, nós passamos uma referência ao primeiro nó:

```
>>> imprimeLista(no1)
1 2 3
```

Dentro de `imprimeLista` nós temos uma referência para o primeiro nó da lista, mas não há variáveis que refiram aos outros nós. Nós temos que usar o valor `proximo` de cada nó para alcançar o próximo nó.

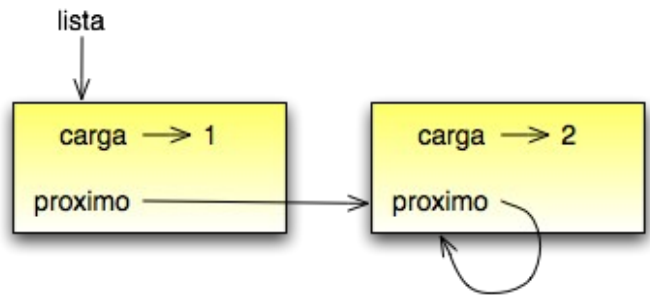
Para percorrer uma lista ligada, é comum usar uma variável `laço` como `no` para referir a cada um dos nós sucessivamente.

Este diagrama mostra o valor de lista e os valores que `no` assume:



Por convenção, listas são freqüentemente impressas em braquetes com vírgulas entre os elementos, como em [1, 2, 3]. Como um exercício, modifique imprimeLista para que ela gere uma saída neste formato.

figura mostra uma lista com dois nós, um dos quais refere-se a si mesmo:



### 17.4 Listas e Recorrência

É natural expressar muitas operações de listas utilizando métodos recorrentes. Por exemplo, o seguinte é um algoritmo recorrente para imprimir uma lista de trás para frente.

1. Separe a lista em dois pedaços: o primeiro nó (chamado a cabeça); e o resto (chamado o rabo).
2. Imprima o rabo de trás para frente.
3. Imprima a cabeça.

Logicamente, o Passo 2, a chamada recorrente, assume que nós temos um modo de imprimir a lista de trás para frente. Mas se nós assumimos que a chamada recorrente funciona -- o passo de fé -- então podemos nos convencer de que o algoritmo funciona.

Tudo o que precisamos são um caso base e um modo de provar que para qualquer lista, nós iremos, ao final, chegar no caso base. Dada a definição recorrente de uma lista, um caso base natural é a lista vazia, representada por None:

```
def imprimeDeTrasParaFrente(lista):
    if lista == None : return
    cabeca = lista
    rabo = lista.proximo
    imprimeDeTrasParaFrente(rabo)
    print cabeca,
```

A primeira linha trata o caso base fazendo nada. As próximas duas linhas dividem a lista em cabeca e rabo. As duas últimas linhas imprimem a lista. A vírgula no final da última linha impede o Python de imprimir uma nova linha após cada nó.

Nós invocamos este método como invocamos o imprimeLista:

```
>>> imprimeDeTrasParaFrente(no1)
3 2 1
```

O resultado é a lista de trás para frente.

Você pode se perguntar por que imprimeLista e imprimeDeTrasParaFrente são funções e não métodos da classe No. A razão é que nós queremos usar None para representa a lista vazia e não é legal invocar um método sobre None. Esta limitação torna complicado escrever código de manipulação de lista em estilo orientado a objeto limpo.

Podemos provar que imprimeDeTrasParaFrente sempre termina? Em outras palavras, irá ela sempre atingir o caso base? De fato, a resposta é não. Algumas listas farão este método falhar.

### 17.5 Listas Infinitas

Não há nada que impeça um nó de referenciar de volta um nó anterior na lista, incluindo ele mesmo. Por exemplo, esta

Se nós invocarmos imprimeLista nesta lista, ele ficará em laço para sempre. Se nós invocarmos imprimeDeTrasParaFrente, ele recorrerá infinitamente. Este tipo de comportamento torna as listas infinitas difíceis de se lidar.

A despeito disto, elas ocasionalmente são úteis. Por exemplo, podemos representar um número como uma lista de dígitos e usar uma lista infinita para representar uma fração repetente.

Mesmo assim, é problemático que não possamos provar que imprimeLista e imprimeDeTrasParaFrente terminem. O melhor que podemos fazer é a afirmação hipotética, "Se a lista não contém laços, então este método terminará." Este tipo de hipótese é chamado uma **pré-condição**. Ele impõe uma limitação sobre um dos parâmetros e descreve o comportamento do método se a limitação é satisfeita. Você verá mais exemplos em breve.

### 17.6 O Teorema da Ambigüidade Fundamental

Uma parte de imprimeDeTrasParaFrente pode ter gerado surpresa:

```
cabeca = lista
rabo = lista.proximo
```

Após a primeira atribuição, cabeca e lista têm o mesmo tipo e o mesmo valor. Então por que nós criamos uma nova variável?

A razão é que as duas variáveis têm diferentes papéis. Quando pensamos em cabeca, pensamos como uma referência a um único nó, e quando pensamos em lista o fazemos como uma referência ao primeiro nó da lista. Estes "papéis" não são parte do programa; eles estão na mente do programador.

Em geral não podemos dizer olhando para o programa qual o papel que uma variável tem. Esta ambigüidade pode ser útil, mas também pode tornar os programas difíceis de serem lidos. Usamos freqüentemente nomes de variáveis como no e lista para documentar como pretendemos usar uma variável e algumas vezes criamos variáveis adicionais para remover a ambigüidade.

Poderíamos ter escrito imprimeDeTrasParaFrente sem cabeca e rabo, que a tornaria mais concisa mas possivelmente menos clara:

```
def imprimeDeTrasParaFrente(lista):
    if lista == None : return
    imprimeDeTrasParaFrente(lista.proximo)
    print lista,
```

Olhando para as duas chamadas de função, temos que lembrar que imprimeDeTrasParaFrente trata seu argumento como uma coleção e print trata seu argumento como um objeto único.

O **teorema da ambigüidade fundamental** descreve a ambigüidade que é inerente à referência a um nó:

Uma variável que refere a um nó pode tratar o nó como um objeto único ou como o primeiro em uma lista de nós.

```
cabeca = lista
rabo = lista.proximo
imprimeDeTrasParaFrente(rabo)
print cabeca,
print "]",
```

Novamente, é uma boa idéia verificar métodos como este para ver se eles funcionam com casos especiais como uma lista vazia ou um singleton.

### 17.7 Modificando Listas

Existem duas maneiras de se modificar uma lista ligada. Obviamente, podemos modificar a carga dos nós, mas as operações mais interessantes são aquelas que adicionam, removem ou reordenam os nós.

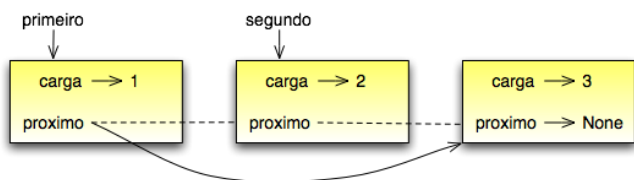
Como um exemplo, vamos escrever um método que remove o segundo nó na lista e retorna uma referência ao nó removido:

```
def removeSegundo(lista):
    if lista == None : return
    primeiro = lista
    segundo = lista.proximo
    # faz o primeiro no referir ao terceiro
    primeiro.proximo = segundo.proximo
    # separa o segundo no do resto da lista
    segundo.proximo = None
    return segundo
```

Novamente, estamos usando variáveis temporárias para tornar o código mais fácil de ser lido. Aqui está como usar este método:

```
>>> imprimeLista(no1)
1 2 3
>>> removido = removeSegundo(no1)
>>> imprimeLista(removido)
2
>>> imprimeLista(no1)
1 3
```

Este diagrama de estado mostra o efeito da operação:



O que acontece se você invocar este método e passar uma lista com somente um elemento (um **singleton**)? O que acontece se você passar a lista vazia como um argumento? Existe uma pré-condição para este método? Se houver, corrija o método para tratar uma violação da pré-condição de modo razoável.

### 17.8 Envoltórios e Ajudadores

Freqüentemente é útil dividir uma operação de lista em dois métodos. Por exemplo, para imprimir uma lista de trás para frente no formato convencional de lista [3, 2, 1], podemos usar o método `imprimeDeTrasParaFrente` para imprimir 3, 2, mas queremos um método separado para imprimir os braquetes e o primeiro nó. Vamos chamá-lo de `imprimeDeTrasParaFrenteLegal`:

```
def imprimeDeTrasParaFrenteLegal(lista):
    print "[",
    if lista != None :
```

Quando usamos este método em algum lugar no programa, invocamos `imprimeDeTrasParaFrenteLegal` diretamente, e ele invoca `imprimeDeTrasParaFrente` por nós. Neste sentido, `imprimeDeTrasParaFrenteLegal` atua como um envoltório, e usa `imprimeDeTrasParaFrente` como um ajudador.

### 17.9 A Classe ListaLigada

Existem alguns problemas sutis com o modo que implementamos listas. Em um inverso de causa e efeito, proporemos uma implementação alternativa primeiro e então explicaremos qual problema ela resolve.

Primeiro, criaremos uma nova classe chamada `ListaLigada`. Seus atributos são um inteiro que contém o comprimento da lista e uma referência para o primeiro nó. Objetos do tipo `ListaLigada` servem como cabos (handles) para se manipular listas de objetos No:

```
class ListaLigada:
    def __init__(self):
        self.comprimento = 0
        self.cabeca = None
```

Uma coisa legal acerca da classe `ListaLigada` é que ela provê um lugar natural para se colocar funções envoltórias como `imprimeDeTrasParaFrenteLegal`, que podemos transformar em um método da classe `ListaLigada`:

```
class ListaLigada:
    ...
    def imprimeDeTrasParaFrente(self):
        print "[",
        if self.cabeca != None :
            self.cabeca.imprimeDeTrasParaFrente()
        print "]",

class No:
    ...
    def imprimeDeTrasParaFrente(self):
        if self.proximo != None:
            rabo = self.proximo
            rabo.imprimeDeTrasParaFrente()
        print self.carga,
```

Apenas para tornar as coisas confusas, mudamos o nome de `imprimeDeTrasParaFrenteLegal`. Agora existem dois métodos chamados `imprimeDeTrasParaFrente`: um na classe `No` (o ajudador); e um na classe `ListaLigada` (o envoltório). Quando o envoltório invoca `self.cabeca.imprimeDeTrasParaFrente`, ele está invocando o ajudador, porque `self.cabeca` é um objeto `No`.

Outro benefício da classe `ListaLigada` é que ela torna mais fácil adicionar e remover o primeiro elemento de uma lista. Por exemplo, `adicionaPrimeiro` é um método para `ListaLigada`; ele toma um item de carga como argumento e o coloca no início da lista:

```
class ListaLigada:
    ...
```

```
def adicionaPrimeiro(self, carga):
    no = No(carga)
    no.proximo = self.cabeca
    self.cabeca = no
    self.comprimento = self.comprimento + 1
```

Como de costume, você deve conferir códigos como este para ver se eles tratam os casos especiais. Por exemplo, o que acontece se a lista está inicialmente vazia?

### 17.10 Invariantes

Algumas listas são "bem formadas"; outras não o são. Por exemplo, se uma lista contém um laço, ela fará muitos de nossos métodos falharem, de modo que podemos querer requerer que listas não contenham laços. Outro requerimento é que o valor de comprimento no objeto ListaLigada seja igual ao número real de nós da lista.

Requerimentos como estes são chamados de invariantes porque, idealmente, eles deveriam ser verdade para cada objeto o tempo todo. Especificar invariantes para objetos é uma prática de programação útil porque torna mais fácil provar a correção do código, verificar a integridade das estruturas de dados e detectar erros.

Uma coisa que algumas vezes é confusa acerca de invariantes é que existem momentos em que eles são violados. Por exemplo, no meio de adicionaPrimeiro, após termos adicionado o nó mas antes de termos incrementado comprimento, o invariante é violado. Este tipo de violação é aceitável; de fato, é freqüentemente impossível modificar um objeto sem violar um invariante por, no mínimo, um pequeno instante. Normalmente, requeremos que cada método que viola um invariante deve restaurar este invariante.

Se há qualquer aumento significativo de código no qual o invariante é violado, é importante tornar isto claro nos comentários, de modo que nenhuma operação seja feita que dependa daquele invariante.

### 17.11 Glossário

<b>referência embutida</b>	Uma	referência
----------------------------	-----	------------

<b>(embedded reference)</b>	armazenada/associada em/a um atributo de um objeto.
<b>lista ligada (linked list)</b>	Uma estrutura de dados que implementa uma coleção usando uma sequência de nós ligados.
<b>nó ou nodo (node)</b>	Um elemento de uma lista, usualmente implementado como um objeto que contém uma referência para outro objeto do mesmo tipo.
<b>carga (cargo)</b>	Um item de dado contido em um nó.
<b>ligação (link)</b>	Uma referência embutida usada para ligar/conectar um objeto a outro.
<b>pré-condição (precondition)</b>	Uma asserção que precisa/deve ser verdadeira para que um método trabalhe corretamente.
<b>teorema da ambigüidade fundamental (fundamental ambiguity theorem)</b>	Uma referência para um nó de uma lista pode ser tratada como um objeto único ou como o primeiro em uma lista de nós.
<b>singleton (singleton)</b>	Uma lista ligada com somente um nó.
<b>envoltório (wrapper)</b>	Um método que atua como um intermediário (middleman) entre um chamador e um método ajudador (helper), freqüentemente tornando a invocação do método mais fácil ou menos propensa a erros.
<b>ajudador (helper)</b>	Um método que não é invocado diretamente pelo chamador (caller) mas é usado por outro método para realizar parte de uma operação.
<b>invariante (invariant)</b>	Uma asserção que deveria ser verdadeira sempre para um objeto (exceto talvez enquanto o objeto estiver sendo modificado).



## Capítulo 18: Pilhas

### 18.1 Tipos abstratos de dados

Os tipos de dados que você viu até agora são todos concretos, no sentido que nós especificamos completamente como eles são implementados. Por exemplo, a classe `Card` (XXX ver como foi traduzido) representa uma carta utilizando dois inteiros. Como discutimos no momento, esta não é a única maneira de representar uma carta; existem muitas implementações alternativas.

Um **tipo abstrato de dado**, ou TAD, especifica um conjunto de operações (ou métodos) e a semântica das operações (o que elas fazem), mas não especifica a implementação das operações. Isto é o que o faz abstrato.

Por que isto é útil?

- Simplifica a tarefa de especificar um algoritmo se você pode XXX denotar (denote) as operações que você precisa sem ter que pensar, ao mesmo tempo, como as operações são executadas.
- Uma vez que existem geralmente muitas maneiras de implementar um TAD, pode ser útil escrever um algoritmo que pode ser usado com qualquer das possíveis implementações.
- TADs bastante conhecidos, como o TAD Pilha deste capítulo, já estão implementados em bibliotecas padrão, então eles podem ser escritos uma vez e usado por muitos programadores.
- As operações em TADs provêm uma linguagem de alto nível comum para especificar e falar sobre algoritmos.

Quando falamos sobre TADs, geralmente distinguimos o código que usa o TAD, chamado **cliente**, do código que implementa o TAD, chamado código **fornecedor**.

### 18.2 O TAD Pilha

Neste capítulo, iremos olhar um TAD comum, a **pilha**. Uma pilha é uma coleção, ou seja, é uma estrutura de dados que contei múltiplos elementos. Outras coleções que vimos incluem dicionários e listas.

Um TAD é definido pelo conjunto de operações que podem ser executadas nele, que é chamado **interface**. A interface para uma pilha consiste nestas operações:

`__init__` : Inicializa uma nova pilha vazia.

`push` : Adiciona um novo item na pilha

`pop` : Remove um item da pilha e o retorna, O item que é retornado é sempre o último adicionado.

`isEmpty` : Verifica se a pilha está vazia.

Uma às vezes é chamada uma estrutura de dados "last in, first out" ou LIFO, porque o último item adicionado é o primeiro a ser removido.

### 18.3 Implementando pilhas com listas de Python

As operações de lista que Python oferecem são similares às operações que definem uma pilha. A interface não é exatamente o que se supõe ser, mas podemos escrever um código para traduzir do TAD Pilha para as operações nativas.

Este código é chamado uma **implementação** do TAD Pilha. Geralmente, uma implementação é um conjunto de métodos que satisfazem os requisitos sintáticos e semânticos de uma interface.

Aqui está uma implementação do TAD Pilha que usa uma lista do Python:

```
class Stack :
    def __init__(self) :
        self.items = []

    def push(self, item) :
        self.items.append(item)

    def pop(self) :
        return self.items.pop()

    def isEmpty(self) :
        return (self.items == [])
```

Um objeto `Stack` contém um atributo chamado `items` que é uma lista de itens na pilha. O método de inicialização define `items` como uma lista vazia.

Para adicionar um novo item na pilha, `push` o coloca em `items`. Para remover um item da pilha, `pop` usa o método de lista homônimo<sup>1</sup> para remover e retornar um último item da lista.

Finalmente, para verificar se a pilha está vazia, `isEmpty` comprara `items` a uma lista vazia.

Uma implementação como esta, na qual os métodos consistem de simples invocações de métodos existentes, é chamado **revestimento**. Na vida real, revestimento é uma fina camada de madeira de boa qualidade usado em XXX\*furniture-making\* para esconder madeira de menor qualidade embaixo. Cientistas da Computação usam esta metáfora para descrever um pequeno trecho de código que esconde os detalhes de uma implementação e fornece uma interface mais simples, ou mais padronizada.

### 18.4 Empilhando e desempilhando

Uma pilha é uma **estrutura de dados genérica**, o que significa que podemos adicionar qualquer tipo de item a ela. O exemplo a seguir empilha dois inteiros e uma XXXstring na pilha:

```
>>> s = Stack()
>>> s.push(54)
>>> s.push(45)
>>> s.push("+")
```

Podemos usar `isEmpty` e `pop` para remover e imprimir todos os itens da pilha:

```
while not s.isEmpty() :
    print s.pop()
```

A saída é + 45 54. Em outras palavras, usamos a pilha para imprimir os itens ao contrário! Sabidamente, este não é o formato padrão de imprimir uma lista, mas, usando uma pilha, foi notavelmente fácil de fazer.

Você deve comparar este trecho de código com a implementação de `printBackward` na seção 17.4. Existe um paralelo natural entre a versão recursiva de `printBackward` e o algoritmo de pilha aqui. A diferença é que `printBackward` usa a pilha de execução para XXXmanter a trilha(keep track) dos nós enquanto percorre a lista, e então imprime-a no retorno da recursão. o algoritmo de pilha faz a mesma coisa, exceto que usa o objeto `Stack` ao invés da pilha de execução.

### 18.5 Usando uma pilha para avaliar expressões pós-fixas

Em muitas linguagens de programação, expressões matemáticas são executadas com o operador entre os dois operandos, como em  $1 + 2$ . Este formato é chamado notação **infixa**. Uma alternativa usada por algumas calculadoras é chamada notação **pós-fixa**. Em notação pós-fixa, o operador segue os operandos, como em  $1\ 2\ +$ .

A razão pela qual a notação pós-fixa é útil algumas vezes é que existe uma maneira natural de avaliar uma expressão pós-fixa usando uma pilha:

- começando no início da expressão, pegue um termo (operador ou operando) de cada vez.
- Se o termo é um operando, empilhe-o
- Se o termo é um operador, desempilhe dois operandos, execute a operação neles, e empilhe o resultado.
- Quando chegar ao fim da expressão, deve existir exatamente um operando sobrando na pilha. Este operando é o resultado.
- Como um exercício, aplique este algoritmo à expressão  $1\ 2\ +\ 3\ *$ .

Este exemplo demonstra uma das vantagens da notação pós-fixa - não é necessário usar parênteses para controlar a ordem das operações. Para ter o mesmo resultado em notação infix, deveríamos escrever  $(1 + 2) * 3$ .

Como um exercício, escreva a expressão pós-fixa que é equivalente a  $1 + 2 * 3$ .

### 18.6 Análise sintática

Para implementar o algoritmo anterior, necessitamos estar prontos para percorrer uma string e quebrá-la em operandos e operadores. Este processo é um exemplo de **parsing**, e o resultado - os pedaços da string - são chamados **tokens**. Você deve lembrar estas palavras do capítulo 1.

Python fornece um método `split` nos módulos `string` e `re` (expressões regulares). A função `string.split` separa uma string numa lista usando um único caracter como **delimitador**. Por exemplo:

```
>>> import string
>>> string.split("Now is the time", " ")
['Now', 'is', 'the', 'time']
```

Neste caso, o delimitador é o caracter de espaço, então a string é dividida a cada espaço.

A função `re.split` é mais poderosa, permitindo fornecer uma expressão regular ao invés de um delimitador. Uma expressão regular é uma maneira de especificar um conjunto de strings. Por exemplo, `[A-z]` é o conjunto de todas as letras e `[0-9]` é o conjunto de todos os dígitos. O operador `^` nega um conjunto, então `[^0-9]` é o conjunto de tudo o que não é número, que é exatamente o que queremos para dividir expressões pós-fixas.

```
>>> import re
>>> re.split("[^0-9]", "123+456*/")
['123', '+', '456', '*', '/', '']
```

Note que a ordem dos argumentos é diferente de `string.split`, o delimitador vem antes da string.

A lista resultante inclui os operandos 123 e 456, e os operadores `*` e `/`. Também inclui duas strings vazias que são inseridas depois dos operadores.

### 18.7 Avaliando em pós-fixo.

Para avaliar uma expressão pós-fixa, usaremos o parser da seção anterior e o algoritmo da seção anterior a ela. Para manter as coisas simples, começaremos com um avaliador que implementa somente os operadores `+` e `*`.

```
def evalPostfix (expr) :
    import re
    tokenList = re.split ("([^\d])", expr)
    stack = Stack()
    for token in tokenList
        if token == '' or token == ' ' :
            continue
        if token == '+' :
            sum = stack.pop() + stack.pop()
            stack.push(sum)
        if token == '*' :
            product = stack.pop() * stack.pop()
            stack.push(product)
        else:
            stack.push(int(token))
    return stack.pop()
```

A primeira condição cuida de espaços e strings vazias. As duas próximas condições manipulam os operadores. Nós assumimos, agora que qualquer coisa é um operador. É claro, seria melhor chegar por entrada errônea e enviar uma mensagem de erro, mas faremos isto depois.

Vamos testá-lo avaliando a forma pós-fixa de  $(56 + 47) * 2$

```
>>> print evalPostfix("56 47 + 2 *")
206
```

### 18.8 Clientes de fornecedores.

Um dos objetivos de um TAD é separar os interesses do fornecedor, quem escreve o código que implementa o TAD, e o cliente, que usa o TAD. O fornecedor tem que se preocupar apenas se a implementação está correta - de acordo com a especificação do TAD - e não como ele será usado.

Inversamente, o cliente assume que a implementação do TAD está correta e não se preocupa com os detalhes. Quando você está usando um tipo nativo do Python, tem o luxo de pensar exclusivamente como um cliente.

É claro, quanto você implementa um TAD, você também tem que escrever código cliente para testá-lo. Neste caso, você faz os dois papéis, o que pode ser confuso. Você

deve fazer algum esforço para manter a trilha do papel que está fazendo a cada momento.

### 18.9 Glossário

<b>tipo abstrato de dados (TAD) (abstract data type(ADT)):</b>	Um tipo de dado(geralmente uma coleção de objetos) que é definido por um conjunto de operações, que podem ser implementadas de várias maneiras.
<b>interface (interface):</b>	É o conjunto de operações que definem um TDA.
<b>implementação (implementation):</b>	Código que satisfaz(preenche?) os requisitos sintáticos e semânticos de uma interface.
<b>cliente (client):</b>	Um programa (ou o programador que o escreveu) que faz utilização de um TDA.
<b>fornecedor (provider):</b>	Código (ou o programador que o escreveu) que implementa um TDA.
<b>revestimento (veneer):</b>	Definição de classe que implementa um TDA com definições de métodos que são chamadas a outros métodos, às vezes com pequenas modificações. A lâmina faz um trabalho insignificante, mas melhora ou padroniza a interface dada ao cliente.
<b>estrutura de dados genérica (generic data structure):</b>	Tipo de estrutura de dados que contém dados de um tipo qualquer(tipo genérico).
<b>infixa (infix):</b>	Notação matemática em que os operadores se situam entre os operandos.
<b>pós-fixa (postfix):</b>	Notação matemática em que os operadores se situam após os operandos.
<b>parse (parse):</b>	Ler um conjunto de caracteres(string de caracteres) ou tokens(símbolos atômicos) e analisar sua estrutura gramatical.
<b>token (token):</b>	Conjunto de caracteres que são tratados como uma unidade atômica para fins de análise gramatical, como as palavras na linguagem natural.
<b>delimitador (delimiter):</b>	Um caractere que é utilizado para separar os símbolos atômicos(tokens), como a pontuação na linguagem natural.

## Capítulo 19: Filas

Este capítulo apresenta dois TDAs: Fila e Fila por Prioridade. Na nossa vida diária, fila é um alinhamento de consumidores aguardando algum tipo de serviço. Na maioria dos casos, o primeiro da fila é o primeiro a ser atendido. Mas há exceções. No aeroporto, passageiros cujo voo vai decolar logo, às vezes são chamados primeiro ao balcão do check-in, mesmo que estejam no meio da fila. No supermercado, é comum na fila do caixa alguém deixar passar na frente uma pessoa que chega à fila só com um ou dois produtos na mão.

A regra que determina quem é o próximo da fila chama-se política de enfileiramento. A política de enfileiramento mais simples chama-se FIFO, sigla de first-in-first-out: primeiro a entrar, primeiro a sair. A política de enfileiramento mais geral é o enfileiramento por prioridade, em que se atribui uma prioridade a cada pessoa da fila e a que tiver maior prioridade vai primeiro, independente da sua ordem de chegada. Dizemos que essa é a política mais geral de todas, porque a prioridade pode ser baseada em qualquer coisa: hora de partida do voo; quantos produtos a pessoa vai passar pelo caixa; o grau de prestígio da pessoa. É claro que nem todas as políticas de enfileiramento são "justas", mas o que é justo depende do ponto de vista.

O TDA Fila e o TDA Fila por Prioridade têm o mesmo conjunto de operações. A diferença está na semântica das operações: a fila usa a política FIFO; e a fila por prioridade (como o próprio nome sugere) usa a política de enfileiramento por prioridade.

### 19.1 Um TDA Fila

O TDA Fila é definido pelas seguintes operações:

<b><code>__init__</code></b>	Inicializar uma nova fila vazia.
<b><code>insert</code></b>	Adicionar um novo item à fila.
<b><code>remove</code></b>	Remover e retornar um item da fila. O item retornado é o que foi adicionado primeiro.
<b><code>isEmpty</code></b>	Checar se a fila está vazia.

### 19.2 Fila encadeada

A primeira implementação que vamos ver de um TDA Fila chama-se **fila encadeada** porque é feita de objetos Nós encadeados. A definição da classe é a seguinte:

```
class Queue:
    def __init__(self):
        self.length = 0
        self.head = None

    def isEmpty(self):
        return (self.length == 0)

    def insert(self, cargo):
        node = Node(cargo)
        node.next = None
        if self.head == None:
            # if list is empty the new node goes first
            self.head = node
```

```
else:
    # find the last node in the list
    last = self.head
    while last.next: last = last.next
    # append the new node
    last.next = node
    self.length = self.length + 1
```

```
def remove(self):
    cargo = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    return cargo
```

Os métodos `isEmpty` e `remove` são idênticos aos métodos `isEmpty` e `removeFirst` de `LinkedList`. O método `insert` é novo e um pouco mais complicado.

Queremos inserir novos itens no fim da lista. Se a fila estiver vazia, basta fazer `head` apontar ao novo nó. Se não, percorremos a lista até o último nó e lá penduramos o novo nó. É possível identificar o último nó porque o seu atributo `next` é `None`.

Existem duas invariantes para um objeto Fila bem formado: o atributo `length` deve ser o número de nós na fila, e o último nó deve ter seu atributo `next` igual a `None`. Estude o método até ficar convencido de que ele preserva ambas invariantes.

### 19.3 Características de performance

Quando invocamos um método, normalmente não estamos preocupados com os detalhes da sua implementação. Porém, há um certo "detalhe" que pode ser bom conhecer: as características de performance do método. Quanto tempo leva, e como o tempo de execução muda à medida em que aumenta o número de itens da coleção?

Primeiro, olhe para `remove`. Não há laços ou chamadas de função aqui, o que sugere que o tempo de execução desse método é sempre o mesmo, toda vez. Um método assim é chamado de operação de **tempo constante**. Na verdade, o método pode ser ligeiramente mais rápido quando a lista está vazia, uma vez que ele pula o corpo da condicional, mas essa diferença não é significativa. XXX: o condicional só aparece na re-implementação do método na classe `ImprovedQueue`, p.200; essa inconsistência pode ser conferida também nas páginas 198-199 do livro original (PDF e impresso).

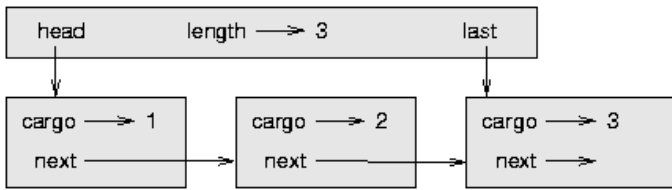
A performance de `insert` é muito diferente. No caso geral, temos de percorrer a lista para achar o último elemento.

Este percurso leva um tempo proporcional à extensão da lista. Uma vez que o tempo de execução é uma função linear da extensão, dizemos que este método opera em **tempo linear**. Isso é bem ruim, se comparado com o tempo constante.

### 19.4 Fila encadeada aprimorada

Queremos uma implementação do TDA Fila que possa

realizar todas as operações em tempo constante. Uma maneira de fazer isso é modificar a classe Fila, de modo que ela mantenha a referência tanto ao primeiro quanto ao último nó, como mostra a figura:



A implementação de ImprovedQueue tem essa cara:

```
class ImprovedQueue:
    def __init__(self):
        self.length = 0
        self.head = None
        self.last = None

    def isEmpty(self):
        return (self.length == 0)
```

Até agora, a única mudança é o atributo last. Ele é usado nos métodos insert e remove:

```
class ImprovedQueue:
    # ...
    def insert(self, cargo):
        node = Node(cargo)
        node.next = None
        if self.length == 0:
            #if list is empty, the new node is head & last
            self.head = self.last = node
        else:
            # find the last node
            last = self.last
            # append the new node
            last.next = node
            self.last = node
        self.length = self.length + 1
```

Uma vez que last não perde de vista o último nó, não é necessário buscá-lo. Como resultado, esse método tem tempo constante.

Mas essa rapidez tem preço. É preciso adicionar um caso especial a remove, para configurar last para None quando o último nó é removido:

```
class ImprovedQueue:
    #...
    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length = self.length - 1
        if self.length == 0:
            self.last = None
        return cargo
```

Essa implementação é mais complicada que a primeira, e mais difícil de se demonstrar que está correta. A vantagem é que o objetivo foi atingido -- tanto insert quanto remove são operações de tempo constante.

Como exercício, escreva uma implementação do TDA Fila usando uma lista nativa do Python. Compare a performance dessa implementação com a de ImprovedQueue, para filas de diversos comprimentos.

### 19.5 Fila por prioridade

O TDA Fila por Prioridade tem a mesma interface que o TDA Fila, mas semântica diferente. Mais uma vez, a interface é a seguinte:

<b><code>__init__</code></b>	Inicializar uma nova fila vazia.
<b><code>insert</code></b>	Adicionar um novo item à fila.
<b><code>remove</code></b>	Remover e retornar um item da fila. O item retornado é aquele que tiver maior prioridade.
<b><code>isEmpty</code></b>	Checar se a fila está vazia.

A diferença semântica é que o item removido da fila não é necessariamente o que foi incluído primeiro e, sim, o que tem maior prioridade. Que prioridades são essas e como elas se comparam umas com as outras não é especificado pela implementação Fila por Prioridade. Isso depende de quais itens estão na fila.

Por exemplo, se os itens da fila tiverem nome, podemos escolhê-los por ordem alfabética. Se for a pontuação de um jogo de boliche, podemos ir da maior para a menor, mas se for pontuação de golfe, teríamos que ir da menor para a maior. Se é possível comparar os itens da fila, é possível achar e remover o que tem maior prioridade. Essa implementação da Fila por Prioridade tem como atributo uma lista Python chamada items, que contém os itens da fila.

```
class PriorityQueue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def insert(self, item):
        self.items.append(item)
```

O método de inicialização, isEmpty, e insert são apenas uma fachada para operações básicas de lista. O único método interessante é remove:

```
class PriorityQueue:
    # ...
    def remove(self):
        maxi = 0
        for i in range(1, len(self.items)):
            if self.items[i] > self.items[maxi]:
                maxi = i
        item = self.items[maxi]
        self.items[maxi:maxi+1] = []
        return item
```

No início de cada iteração, maxi armazena o índice do maior item (a prioridade mais alta de todas) que vimos até agora. A cada volta do laço, o programa compara o i-ésimo item ao campeão. Se o novo item for maior, maxi recebe o valor de i.

Quando o comando for se completa, maxi é o índice do maior item. Esse item é removido da lista e retornado.

Vamos testar a implementação:

```
>>> q = PriorityQueue()
>>> q.insert(11)
>>> q.insert(12)
>>> q.insert(14)
>>> q.insert(13)
>>> while not q.isEmpty(): print q.remove()
14
```



```
13
12
11
```

Se a fila contém números ou strings simples, eles são removidas em ordem numérica decrescente ou alfabética invertida (de Z até A). Python consegue achar o maior inteiro ou string porque consegue compará-los usando os operadores de comparação nativos.

Se a fila contém objetos de outro tipo, os objetos têm que prover um método `__cmp__`. Quando remove usa o operador `>` para comparar dois itens, o método `__cmp__` de um dos itens é invocado, recebendo o segundo item como argumento. Desde que o método `__cmp__` funcione de forma consistente, a Fila por Prioridade vai funcionar.

```
>>> pq = PriorityQueue()
>>> pq.insert(tiger)
>>> pq.insert(phil)
>>> pq.insert(hal)
>>> while not pq.isEmpty(): print pq.remove()
Tiger Woods      : 61
Hal Sutton       : 69
Phil Mickelson   : 72
```

Como exercício, escreva uma implementação do TDA Fila por Prioridade usando uma lista encadeada. Mantenha a lista em ordem para que a remoção seja uma operação de tempo constante. Compare a performance dessa implementação com a implementação usando uma lista nativa do Python.

### 19.6 A classe Golfer

Como exemplo de um objeto com uma definição não-usual de prioridade, vamos implementar uma classe chamada Golfer (golfista), que mantém registro dos nomes e da pontuação de golfistas. Como sempre, começamos definindo `__init__` e `__str__`:

```
class Golfer:
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def __str__(self):
        return "%-16s: %d" % (self.name, self.score)
```

O método `__str__` usa o operador de formato para colocar nomes e pontuações em colunas arrumadas.

Em seguida, definimos uma versão de `__cmp__`, ma qual a pontuação mais baixa fica com prioridade máxima. Como sempre, `__cmp__` retorna 1 se `self` é "maior que" `other`, -1 se `self` é "menor que" `other`, e 0 se eles são iguais.

```
class Golfer:
    #...
    def __cmp__(self, other):
        if self.score < other.score: return 1 # less
        if self.score > other.score: return -1 # more
        return 0
```

Agora estamos prontos para testar a fila por prioridade com a classe Golfer:

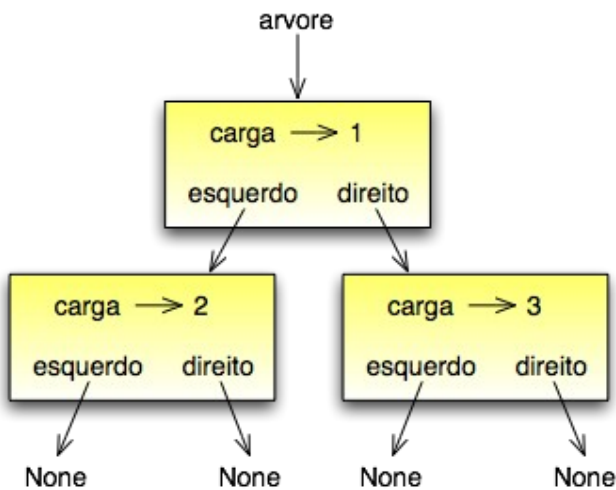
```
>>> tiger = Golfer("Tiger Woods", 61)
>>> phil = Golfer("Phil Mickelson", 72)
>>> hal = Golfer("Hal Sutton", 69)
>>>
```

### 19.7 Glossário

<b>fila (queue)</b>	Conjunto de objetos ordenados esperando algum tipo de serviço.
<b>Fila (Queue)</b>	TAD (Tipo Abstrato de Dado) que realiza operações comuns de acontecerem em uma fila.
<b>política de enfileiramento (queueing policy)</b>	As regras que determinam qual membro de uma fila é o próximo a ser removido.
<b>FIFO</b>	"First In, First Out," (primeiro a entrar, primeiro a sair) política de enfileiramento em que o primeiro membro a chegar é o primeiro a ser removido.
<b>fila por prioridade (priority queue)</b>	Política de enfileiramento em que cada membro tem uma prioridade, determinada por fatores externos. O membro com a maior prioridade é o primeiro a ser removido.
<b>Fila por Prioridade (Priority Queue)</b>	TAD que define as operações comuns de acontecerem em uma fila por prioridade.
<b>fila encadeada (linked queue)</b>	Implementação de uma fila usando uma lista encadeada.
<b>tempo constante (constant time)</b>	Operação cujo tempo de execução não depende do tamanho da estrutura de dados.
<b>tempo linear (linear time)</b>	Operação cujo tempo de execução é uma função linear do tamanho da estrutura de dados.

## Capítulo 20: Árvores

Como listas ligadas, árvores são constituídas de células. Uma espécie comum de árvores é a **árvore binária**, em que cada célula contém referências a duas outras células (possivelmente nulas). Tais referências são chamadas de subárvore esquerda e direita. Como as células de listas ligadas, as células de árvores também contém uma carga. Um diagrama de estados para uma árvore pode aparecer assim:



Para evitar a sobrecarga da figura, nós frequentemente omitimos os Nones.

O topo da árvore (a célula à qual o apontador `tree` se refere) é chamada de **raiz**. Seguindo a metáfora das árvores, as outras células são chamadas de galhos e as células nas pontas contendo as referências vazias são chamadas de **folhas**. Pode parecer estranho que desenhamos a figura com a raiz em cima e as folhas em baixo, mas isto nem será a coisa mais estranha.

Para piorar as coisas, cientistas da computação misturam outra metáfora além da metáfora biológica - a árvore genealógica. Uma célula superior pode ser chamada de **pai** e as células a que ela se refere são chamadas de seus **filhos**. Células com o mesmo pai são chamadas de **irmãos**.

Finalmente, existe também o vocabulário geométrico para falar de árvores. Já mencionamos esquerda e direita, mas existem também as direções "para cima" (na direção da raiz) e "para baixo" (na direção dos filhos/folhas). Ainda nesta terminologia, todas as células situadas à mesma distância da raiz constituem um **nível** da árvore.

Provavelmente não precisamos de três metáforas para falar de árvores, mas aí elas estão.

Como listas ligadas, árvores são estruturas de dados recursivas já que elas são definidas recursivamente:

Uma árvore é

- a árvore vazia, representada por `None`, ou
- uma célula que contém uma referência a um objeto (a carga da célula) e duas referências a árvores.

### 20.1 Construindo árvores

O processo de montar uma árvore é similar ao processo de montar uma lista ligada. cada invocação do construtor cria uma célula.

```

class Tree :
    def __init__(self, carga, left=None, right=None) :
        self.carga = carga
        self.left = left
        self.right = right

    def __str__(self) :
        return str(self.carga)
    
```

A carga pode ser de qualquer tipo, mas os parâmetros `left` e `right` devem ser células. `left` e `right` são opcionais; o valor default é `None`.

Para imprimir uma célula, imprimimos apenas a sua carga.

Uma forma de construir uma árvore é de baixo para cima. Aloque os filhos primeiro:

```

left = Tree(2)
right = Tree(3)
    
```

Em seguida crie a célula pai e ligue ela a seus filhos:

```

tree = Tree(1, left, right);
    
```

Podemos escrever este código mais concisamente encaixando as invocações do construtor:

```

>>> tree = Tree(1, Tree(2), Tree(3))
    
```

De qualquer forma, o resultado é a árvore que apareceu no início do capítulo.

### 20.2 Percorrendo árvores

Cada vez que Você vê uma nova estrutura de dados, sua primeira pergunta deveria ser "Como eu percorro esta estrutura?" A forma mais natural de percorrer uma árvore é fazer o percurso recursivamente. Por exemplo, se a árvore contém inteiros na carga, a função abaixo retorna a soma das cargas:

```

def total(tree) :
    if tree == None : return 0
    return total(tree.left) + total(tree.right) + tree.carga
    
```

O caso base é a árvore vazia, que não contém nenhuma carga, logo a soma das cargas é 0. O passo recursivo faz duas chamadas recursivas para achar a soma das cargas das subárvores dos filhos. Ao finalizar a chamada recursiva, adicionamos a carga do pai e devolvemos o valor total.

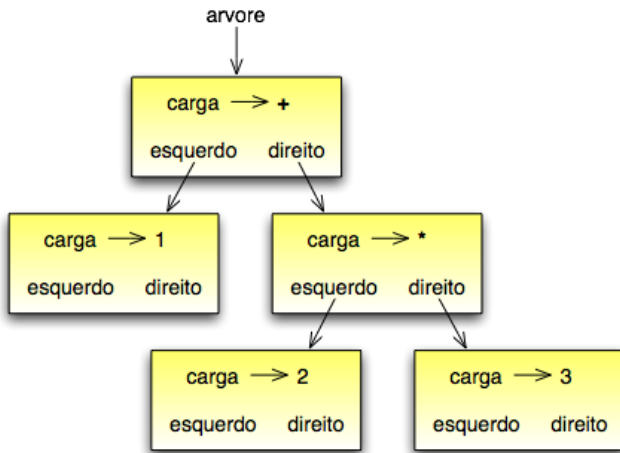
### 20.3 Árvores de expressões

Uma árvore é uma forma natural para representar a estrutura de uma expressão. Ao contrário de outras notações, a árvore pode representar a computação de forma não ambígua. Por exemplo, a expressão infixa `1 + 2 * 3` é ambígua, a menos que

## Como pensar como um cientista da Computação usando Python

saibamos que a multiplicação é feita antes da adição.

A árvore de expressão seguinte representa a mesma computação:



As células de uma árvore de expressão podem ser operandos como 1 e 2 ou operações como + e \*. As células contendo operandos são folhas; aquelas contendo operações devem ter referências aos seus operandos. (Todos os nossos operandos são binários, significando que eles tem exatamente dois operandos.)

Podemos construir árvores assim:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
```

Examinando a figura, não há dúvida quanto à ordem das operações; a multiplicação é feita primeiro para calcular o segundo operando da adição.

Árvores de expressão tem muitos usos. O exemplo neste capítulo usa árvores para traduzir expressões para as notações pósfixa, prefixa e infixa. Árvores similares são usadas em compiladores para analisar sintaticamente, otimizar e traduzir programas.

### 20.4 Percurso de árvores

Podemos percorrer uma árvore de expressão e imprimir o seu conteúdo como segue:

```
def printTree(tree) :
    if tree == None : return
    print tree.carga,
    printTree(tree.left)
    printTree(tree.right)
```

Em outras palavras, para imprimir uma árvore, imprima primeiro o conteúdo da raiz, em seguida imprima toda a subárvore esquerda e finalmente imprima toda a subárvore direita. Esta forma de percorrer uma árvore é chamada de **préordem**, porque o conteúdo da raiz aparece antes dos conteúdos dos filhos. Para o exemplo anterior, a saída é:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
>>> printTree(tree)
+ 1 * 2 3
```

Esta notação é diferente tanto da notação pósfixa quanto da infixa; é uma notação chamada de **prefixa**, em que os operadores aparecem antes dos seus operandos.

Você pode suspeitar que se Você percorre a

árvore numa ordem diferente, Você produzirá expressões numa notação diferente. Por exemplo, se Você imprime subárvores primeiro e depois a raiz, Você terá:

```
def printTreePostorder(tree) :
    if tree == None : return
    printTreePostorder(tree.left)
    printTreePostorder(tree.right)
    print tree.carga,
```

O resultado, 1 2 3 \* +, está na notação pósfixa! Esta ordem de percurso é chamada de pósordem.

Finalmente, para percorrer uma árvore em **inordem**, Você imprime a subárvore esquerda, depois a raiz e depois a subárvore direita:

```
def printTreeInorder(tree) :
    if tree == None : return
    printTreeInorder(tree.left)
    print tree.carga,
    printTreeInorder(tree.right)
```

O resultado é 1 + 2 \* 3, que é a expressão na notação infixa.

Para sermos justos, devemos lembrar que acabamos de omitir uma complicação importante. algumas vezes quando escrevemos expressões na notação infixa devemos usar parêntesis para prescrever a ordem das operações. Ou seja, um percurso em inordem não é suficiente para gerar a expressão infixa.

Ainda assim, com alguns aperfeiçoamentos, a árvore de expressão e os três modos recursivos de percurso resultam em algoritmos para transformar expressões de uma notação para outra.

Como um exercício, modifique `printTreeInorder` de modo que ele coloque parêntesis em volta de cada operador e par de operandos. A saída é correta e não ambígua? Os parêntesis são sempre necessários?

Se percorrermos uma árvore em inordem e acompanharmos em qual nível na árvore estamos, podemos gerar uma representação gráfica da árvore:

```
def printTreeIndented(tree, level=0) :
    if tree == None : return
    printTreeIndented(tree.right, level+1)
    print ' '*level + str(tree.carga)
    printTreeIndented(tree.left, level+1)
```

O parâmetro `level` registra aonde estamos na árvore. Por 'default', o nível inicialmente é zero. A cada chamada recursiva repassamos `level+1` porque o nível do filho é sempre um a mais do que o nível do pai. Cada item é indentado dois espaços por nível. Para o nosso exemplo obtemos:

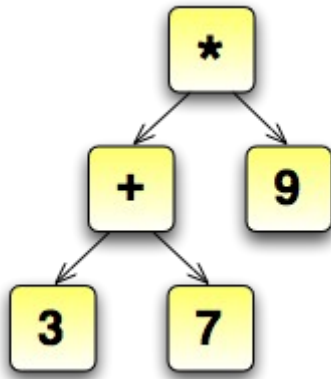
```
>>> printTreeIndented(tree)
      3
     *
    2
   +
  1
```

Se Você deitar a saída acima Você enxerga uma versão simplificada da figura original.

### 20.5 Construindo uma árvore de expressão

Nesta seção analisamos expressões infixas e construímos as árvores de expressão correspondentes. Por exemplo, para a expressão  $(3+7)*9$  resultará a seguinte árvore:

## Como pensar como um cientista da Computação usando Python



Note que simplificamos o diagrama omitindo os nomes dos campos.

O analisador que escreveremos aceitará expressões que incluam números, parêntesis e as operações + e \*. Vamos supor que a cadeia de entrada já foi tokenizada numa lista do Python. A lista de tokens para a expressão (3+7)\*9 é:

```
[ '(', 3, '+', 7, ')', '*', 9, 'end' ]
```

O token final end é prático para prevenir que o analisador tente buscar mais dados após o término da lista.

A título de um exercício, escreva uma função que recebe uma expressão na forma de uma cadeia e devolve a lista de tokens.

A primeira função que escreveremos é getToken que recebe como parâmetros uma lista de tokens e um token esperado. Ela compara o token esperado com o primeiro token da lista: se eles batem a função remove o token da lista e devolve um valor verdadeiro, caso contrário a função devolve um valor falso:

```
def getToken(tokenList, expected) :
    if tokenList[0] == expected :
        tokenList[0:1] = [] # remove the token
        return 1
    else :
        return 0
```

Já que tokenList refere a um objeto mutável, as alterações feitas aqui são visíveis para qualquer outra variável que se refira ao mesmo objeto.

A próxima função, getNumber, trata de operandos. Se o primeiro token na tokenList for um número então getNumber o remove da lista e devolve uma célula folha contendo o número; caso contrário ele devolve None.

```
def getNumber(tokenList) :
    x = tokenList[0]
    if type(x) != type(0) : return None
    del tokenList[0]
    return Tree(x, None, None)
```

Antes de continuar, convém testar getNumber isoladamente. Atribuímos uma lista de números a tokenList, extraímos o primeiro, imprimimos o resultado e imprimimos o que resta na lista de tokens:

```
>>> tokenList = [9, 11, 'end']
>>> x = getNumber(tokenList)
>>> printTreePostorder(x)
9
>>> print tokenList
[11, 'end']
```

Em seguida precisaremos da função getProduct, que constrói uma árvore de expressão para produtos. Os dois operandos de um produto simples são números, como em 3 \* 7.

Segue uma versão de getProduct que trata de produtos simples.

```
def getProduct(tokenList) :
    a = getNumber(tokenList)
    if getToken(tokenList, '*') :
        b = getNumber(tokenList)
        return Tree('*', a, b)
    else :
        return a
```

Supondo que a chamada de getNumber seja bem sucedida e devolva uma árvore de uma só célula atribuímos o primeiro operando a a. Se o próximo caractere for \*, vamos buscar o segundo número e construir a árvore com a, b e o operador.

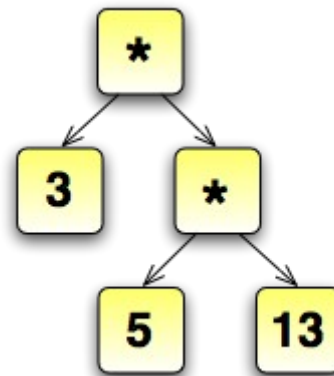
Se o caractere seguinte for qualquer outra coisa, então simplesmente devolvemos uma célula folha com a. Seguem dois exemplos:

```
>>> tokenList = [9, '*', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9 11 *
```

```
>>> tokenList = [9, '+', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9
```

O segundo exemplo sugere que nós consideramos um operando unitário como uma espécie de produto. Esta definição de "produto" talvez não seja intuitiva, mas ela será útil.

Agora tratamos produtos compostos, como 3 \* 5 \* 13. Encaramos esta expressão como um produto de produtos, mais precisamente como 3 \* (5 \* 13). A árvore resultante é:



Com uma pequena alteração em getProduct, podemos acomodar produtos arbitrariamente longos:

```
def getProduct(tokenList) :
    a = getNumber(tokenList)
    if getToken(tokenList, '*') :
        b = getProduct(tokenList) # this line changed
        return Tree('*', a, b)
    else :
```

```
return a
```

Em outras palavras, um produto pode ser um singleton ou uma árvore com \* na raiz, que tem um número como filho esquerdo e um produto como filho direito. Este tipo de definição recursiva devia começar a ficar familiar.

Testemos a nova versão com um produto composto:

```
>>> tokenList = [2, '*', 3, '*', 5, '*', 7, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
2 3 5 7 * * *
```

A seguir adicionamos o tratamento de somas. De novo, usamos uma definição de "soma" que é ligeiramente não intuitiva. Para nós, uma soma pode ser uma árvore com + na raiz, que tem um produto como filho esquerdo e uma soma como filho direito. Ou, uma soma pode ser simplesmente um produto.

Se Você está disposto a brincar com esta definição, ela tem uma propriedade interessante: podemos representar qualquer expressão (sem parêntesis) como uma soma de produtos. Esta propriedade é a base do nosso algoritmo de análise sintática.

getSum tenta construir a árvore com um produto à esquerda e uma soma à direita. Mas, se ele não encontra uma +, ele simplesmente constrói um produto.

```
def getSum(tokenList) :
    a = getProduct(tokenList)
    if getToken(tokenList, '+') :
        b = getSum(tokenList)
        return Tree('+', a, b)
    else :
        return a
```

Vamos testar o algoritmo com  $9 * 11 + 5 * 7$ :

```
>>> tokenList = [9, '*', 11, '+', 5, '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 * 5 7 * +
```

Quase terminamos, mas ainda temos que tratar dos parêntesis. Em qualquer lugar numa expressão onde podemos ter um número, podemos também ter uma soma inteira envolvida entre parêntesis. Precisamos, apenas, modificar getNumber para que ela possa tratar de **subexpressões**:

```
def getNumber(tokenList) :
    if getToken(tokenList, '(') :
        x = getSum(tokenList) # get subexpression
        getToken(tokenList, ')') #eat closing parenthesis
        return x
    else :
        x = tokenList[0]
        if type(x) != type(0) : return None
        tokenList[0:1] = [] # remove the token
        return Tree(x, None, None) # return a leaf
with the number
```

Testemos este código com  $9 * (11 + 5) * 7$ :

```
>>> tokenList = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 5 + 7 * *
```

O analisador tratou os parêntesis corretamente; a adição é feita antes da multiplicação.

Na versão final do programa, seria uma boa

idéia dar a getNumber um nome mais descritivo do seu novo papel.

## 20.6 Manipulando erros

Ao longo do analisador sintático tínhamos suposto que as expressões (de entrada) são bem formadas. Por exemplo, quando atingimos o fim de uma subexpressão, supomos que o próximo caractere é um facha parêntesis. Caso haja um erro e o próximo caractere seja algo diferente, devemos tratar disto.

```
def getNumber(tokenList) :
    if getToken(tokenList, '(') :
        x = getSum(tokenList)
        if not getToken(tokenList, ')') :
            raise 'BadExpressionError',
                'missing parenthesis'
        return x
    else :
        # the rest of the function omitted
```

O comando raise cria uma exceção; neste caso criamos um novo tipo de exceção, chamada de BadExpressionError. Se a função que chamou getNumber, ou uma das outras funções no traceback, manipular a exceção, então o programa pode continuar. caso contrário Python vai imprimir uma mensagem de erro e terminará o processamento em seguida.

A título de exercício, encontre outros locais nas funções criadas onde erros possam ocorrer e adiciona comandos ``raise`` apropriados. Teste seu código com expressões mal formadas.

## 20.7 A árvore dos animais

Nesta seção, desenvolvemos um pequeno programa que usa uma árvore para representar uma base de conhecimento.

O programa interage com o usuário para criar uma árvore de perguntas e de nomes de animais. Segue uma amostra da funcionalidade:

```
Are you thinking of an animal? y
Is it a bird? n
What is the animals name? dog
What question would distinguish a dog from a bird?
Can it fly
If the animal were dog the answer would be? n

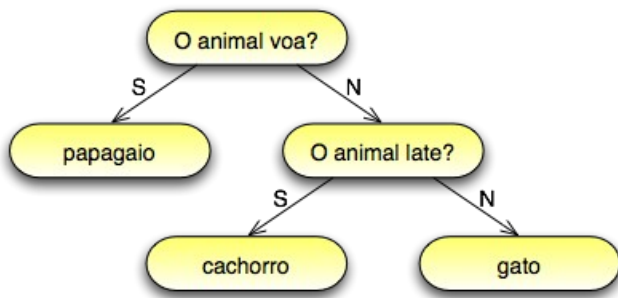
Are you thinking of an animal? y
Can it fly? n
Is it a dog? n
What is the animals name? cat
What question would distinguish a cat from a dog?
Does it bark
If the animal were cat the answer would be? n

Are you thinking of an animal? y
Can it fly? n
Does it bark? y
Is it a dog? y
I rule!

Are you thinking of an animal? n
```

Aqui está a árvore que este diálogo constrói:





No começo de cada rodada, o programa parte do topo da árvore e faz a primeira pergunta. Dependendo da resposta, ele segue pelo filho esquerdo ou direito e continua até chegar numa folha. Neste ponto ele arrisca um palpite. Se o palpite não for correto, ele pergunta ao usuário o nome de um novo animal e uma pergunta que distingue o palpite errado do novo animal. A seguir, adiciona uma célula à árvore contendo a nova pergunta e o novo animal.

Aqui está o código:

```

def animal() :
    # start with a singleton
    root = Tree("bird")

    # loop until the user quits
    while 1 :
        print
        if not yes("Are you thinking of an animal? ") :
            break

        # walk the tree
        tree = root
        while tree.getLeft() != None :
            prompt = tree.getCargo() + "? "
            if yes(prompt):
                tree = tree.getRight()
            else:
                tree = tree.getLeft()

        # make a guess
        guess = tree.getCargo()
        prompt = "Is it a " + guess + "? "
        if yes(prompt) :
            print "I rule!"
            continue

        # get new information
        prompt = "What is the animal's name? "
        animal = raw_input(prompt)
        prompt = "What question would distinguish a %s
from a %s? "
        question = raw_input(prompt % (animal,guess))

        # add new information to the tree
        tree.setCargo(question)
        prompt = "If the animal were %s the answer would
be? "
        if yes(prompt % animal) :
            tree.setLeft(Tree(guess))
            tree.setRight(Tree(animal))
        else :
    
```

```

tree.setLeft(Tree(animal))
tree.setRight(Tree(guess))
    
```

A função `yes` é um auxiliar; ele imprime um prompt e em seguida solicita do usuário uma entrada. Se a resposta começar com `y` ou `Y`, a função devolve um valor verdadeiro:

```

def yes(ques) :
    from string import lower
    ans = lower(raw_input(ques))
    return (ans[0:1] == 'y')
    
```

A condição do laço externo é `1``, que significa que ele continuará até a execução de um comando `break`, caso o usuário não pense num animal.

O laço `while` interno caminha na árvore de cima para baixo, guiado pelas respostas do usuário.

Quando uma nova célula é adicionada à árvore, a nova pergunta substitui a carga e os dois filhos são o novo animal e a carga original.

Uma falha do programa é que ao sair ele esquece tudo que lhe foi cuidadosamente ensinado!

A título de exercício, pense de várias jeitos para salvar a árvore do conhecimento acumulado num arquivo. Implemente aquele que Você pensa ser o mais fácil.

### 20.8 Glossário

<b>árvore binária (binary tree)</b>	Uma árvore em que cada célula tem zero, um ou dois descendentes.
<b>raiz (root)</b>	A célula mais alta de uma árvore, a (única) célula de uma árvore que não tem pai.
<b>folha (leaf)</b>	Uma célula mais baixa numa árvore; uma célula que não tem descendentes.
<b>pai (parent)</b>	A célula que aponta para uma célula dada.
<b>filho (child)</b>	Uma célula apontada por uma célula dada.
<b>irmãos (siebling)</b>	Células que tem o mesmo pai.
<b>nível (level)</b>	Um conjunto de células equidistantes da raiz.
<b>operador binário (binary operator)</b>	Um operador sobre dois operandos.
<b>subexpressão (subexpression)</b>	Uma expressão entre parêntesis que se comporta como um operando simples numa expressão maior.
<b>pré-ordem (preorder)</b>	Uma forma de percorrer uma árvore visitando cada célula antes dos seus filhos.
<b>notação prefixa (prefix notation)</b>	Uma forma de escrever uma expressão matemática em que cada operador aparece antes dos seus operandos.
<b>pós-ordem (postorder)</b>	Uma forma de percorrer uma árvore visitando os filhos de cada célula antes da própria célula.
<b>in-ordem (inorder)</b>	Uma forma de percorrer uma árvore visitando a subárvore esquerda, seguida da raiz e finalmente da subárvore direita.

## GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related

matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from

those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.



You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

---

## 5. COMBINING DOCUMENTS

---

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

---

## 6. COLLECTIONS OF DOCUMENTS

---

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

---

## 7. AGGREGATION WITH INDEPENDENT WORKS

---

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

---

## 8. TRANSLATION

---

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

---

## 9. TERMINATION

---

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

---

## 10. FUTURE REVISIONS OF THIS LICENSE

---

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a

version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

### How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or
modify this document
under the terms of the GNU Free Documentation
License, Version 1.2
or any later version published by the Free Software
Foundation;
with no Invariant Sections, no Front-Cover Texts,
and no Back-Cover
Texts. A copy of the license is included in the
section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR
TITLES, with the
Front-Cover Texts being LIST, and with the Back-
Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.