

Computação Científica com Python

Computação Científica com Python

Uma introdução à programação
para cientistas

Flávio Codeço Coelho

Petrópolis – RJ
Edição do Autor
2007

© 2007 Flávio Codeço Coelho
Todos os direitos reservados.

ISBN: 978-85-907346-0-4

Capa: Mosaico construído com as figuras deste livro imitando o logotipo da linguagem Python. Concebido e realizado pelo autor, com o auxílio do software livre Metapixel.

Revisão ortográfica: Paulo F. Coelho e Luciene C. Coelho.

Edição do autor. Petrópolis - RJ - Brasil

Primeira Edição: Julho 2007

Este livro é dedicado à minha esposa e meu filho, sem os quais nada disso valeria a pena.

Agradecimentos

Muitas pessoas foram indispensáveis para que este livro se tornasse uma realidade. Seria impossível listar todas elas. Mas algumas figuras fundamentais merecem uma menção especial.

Richard M. Stallman. Sem o Software Livre tudo o que eu sei sobre programação, provavelmente se reduziria a alguns comandos de DOS.

Linus Torvalds. Sem o Linux, nunca teria me aproximado o suficiente da programação para conhecer a linguagem Python.

Guido van Rossum. Muito obrigado por esta bela linguagem, e por acreditar que elegância e clareza são atributos importantes de uma linguagem de programação.

Comunidade Python. Obrigado por todas esta extensões maravilhosas ao Python. À comunidade de desenvolvedores do Numpy e Scipy segue um agradecimento especial por facilitar a adoção do Python por cientistas.

Alem destas pessoas gostaria ainda de agradecer ao Fernando Perez (criador e mantenedor do Ipython) por este incrivelmente útil software e por permitir que eu utilizasse alguns dos exemplos da sua documentação neste livro.

Sumário

Sumário	i
Lista de Figuras	vii
Lista de Tabelas	x
Listagens	xi
Prefácio: Computação Científica	xvii
<i>Da Computação Científica e sua definição pragmática.</i>	
<i>Do porquê esta se diferencia, em metas e ferramentas, da Ciência da Computação.</i>	
I Python	1
1 Fundamentos da Linguagem	3
<i>Breve introdução a conceitos básicos de programação e à linguagem Python. A maioria dos elementos básicos da linguagem são abordados neste capítulo, com exceção de classes, que são discutidas em detalhe no capítulo 2.</i>	
<i>Pré-requisitos: Conhecimentos básicos de programação em qualquer linguagem.</i>	

1.1	Primeiras impressões	3
1.2	Uso Interativo <i>vs.</i> Execução a Partir de Scripts	5
	Operações com Números	8
1.3	Nomes, Objetos e Espaços de Nomes	11
1.4	Estruturas de Dados	13
	Listas	13
	Tuplas	20
	Strings	22
	Dicionários	24
	Conjuntos	27
1.5	Controle de fluxo	28
	Condições	28
	Iteração	29
	Lidando com erros: Exceções	32
1.6	Funções	34
	Funções lambda	37
	Geradores	38
	Decoradores	39
	Strings de Documentação	41
1.7	Módulos e Pacotes	42
	Pacotes Úteis para Computação Científica	45
1.8	Documentando Programas	47
	Pydoc	48
	Epydoc	49
1.9	Exercícios	50
2	Orientação a Objeto	51
	<i>Introdução à programação orientada a objetos e sua implementação na linguagem Python. Pré-requisitos: Ter lido o capítulo 1.</i>	
2.1	Objetos	52
	Definindo Objetos e seus Atributos em Python	53
	Adicionando Funcionalidade a Objetos	55
	Herança	56

Utilizando Classes como Estruturas de Dados Genéricas.	57
2.2 Exercícios	57
3 Criando Gráficos em Python	59
<i>Introdução à produção de figuras de alta qualidade utilizando o pacote matplotlib. Pré-requisitos: Capítulo 1.</i>	
3.1 Introdução ao Matplotlib	59
Configurando o MPL	61
Comandos Básicos	62
3.2 Exemplos Simples	63
O comando <code>plot</code>	63
O Comando <code>subplot</code>	65
Adicionando Texto a Gráficos	68
3.3 Exemplos Avançados	69
Mapas	69
4 Ferramentas de Desenvolvimento	73
<i>Exposição de ferramentas voltadas para o aumento da produtividade em um ambiente de trabalho em computação científica. Pré-requisitos: Capítulos 1 e 2</i>	
4.1 Ipython	73
Primeiros Passos	74
Comandos Mágicos	75
4.2 Editores de Código	80
Editores Genéricos	80
Editores Especializados	83
4.3 Controle de Versões em Software	86
Entendendo o Mercurial	88
5 Interagindo com Outras Linguagens	99
<i>Introdução a vários métodos de integração do Python com outras linguagens. Pré-requisitos: Capítulos 1 e 2.</i>	
5.1 Introdução	99

5.2	Integração com a Linguagem C	100
	Weave	100
	Ctypes	105
	Pyrex	106
5.3	Integração com C++	111
	Shedskin	112
5.4	Integração com a Linguagem Fortran	117
	f2py	118
5.5	A Pítón que sabia Javanês — Integração com Java .	126
	O interpretador Jython	126
	Criando “Applets” em Jython	128
5.6	Exercícios	131
 II Aplicações		 133
6	Modelagem Matemática	135
<i>Introdução à modelagem matemática e sua implementação computacional. Discussão sobre a importância da Análise dimensional na modelagem e apresentação do pacote Unum para lidar com unidades em Python. Pré-requisitos: Conhecimentos básicos de cálculo.</i>		
6.1	Modelos	137
	Lineares	137
	Exponenciais	142
6.2	Construindo Modelos Dinâmicos	145
	Modelando Iterativamente	145
	Integração Numérica	148
6.3	Grandezas, Dimensões e Unidades	155
	Análise Dimensional	158
	O Pacote Unum	159
6.4	Exercícios	167

7 Teoria de Grafos	169
<i>Breve introdução a teoria de grafos e sua representação computacional. Introdução ao Pacote NetworkX, voltado para a manipulação de grafos. Pré-requisitos: Programação orientada a objetos.</i>	
7.1 Introdução	169
7.2 NetworkX	172
Construindo Grafos	173
Manipulando Grafos	174
Criando Grafos a Partir de Outros Grafos	175
Gerando um Grafo Dinamicamente	176
Construindo um Grafo a Partir de Dados	179
7.3 Exercícios	182
8 Interação com Bancos de Dados	185
<i>Apresentação dos módulos de armazenamento de dados Pickle e Sqlite3 que fazem parte da distribuição padrão do Python. Apresentação do pacote SQLAlchemy para comunicação com os principais sistemas de bancos de dados existentes. Pré-requisitos: Conhecimentos básicos de bancos de dados e SQL.</i>	
8.1 O Módulo Pickle	186
8.2 O módulo Sqlite3	187
8.3 O Pacote SQLAlchemy	190
Construindo um aranha digital	190
8.4 Exercícios	197
9 Simulações Estocásticas	199
<i>Seleção de problemas relacionados com a simulação e análise de processos estocásticos. Pré-requisitos: Conhecimentos avançados de estatística.</i>	
9.1 Números Aleatórios	199
Hipercubo Latino - LHS	200
9.2 Inferência Bayesiana	203

9.3	Simulações Estocásticas	205
	Integração Monte Carlo	205
9.4	Amostragem por Rejeição	207
	Método do Envelope	208
	Aplicando ao Teorema de Bayes	210
9.5	Cadeias de Markov	214
9.6	MCMC (Markov Chain Monte Carlo)	217
	O Amostrador de Metropolis-Hastings	217
	O Amostrador de Gibbs	220
9.7	Métodos Monte Carlo Sequenciais	223
	Sampling Importance Resampling – SIR	224
9.8	Funções de Verossimilhança	229
	Definição da função de verossimilhança	229
9.9	Inferência Bayesiana com Objetos.	235
9.10	Exercícios	248
Console Gnu/Linux		253
<i>Guia de sobrevivência no console do Gnu/Linux</i>		
	A linguagem BASH	254
	Alguns Comando Úteis	254
	Entradas e Saídas, redirecionamento e "Pipes".	257
	Redirecionamento	258
	“Pipelines”	258
	Pérolas Científicas do Console Gnu/Linux	259
	Gnu plotutils	259
Índice Remissivo		267

Lista de Figuras

1.1	Versão HTML da documentação gerada pelo Epydoc	50
3.1	Histograma simples a partir da listagem 3.2	61
3.2	Reta simples a partir da listagem 3.3	63
3.3	Gráfico com símbolos circulares a partir da listagem 3.4 .	65
3.4	Figura com dois gráficos utilizando o comando subplot, a partir da listagem 3.5	67
3.5	Formatação de texto em figuras. Gerada pela listagem 3.6	70
3.6	Mapa mundi na projeção de Robinson.	72
4.1	Editor emacs em modo Python e com “Code Browser” ativado	82
4.2	Editor Scite.	83
4.3	Editor Gnu Nano.	84
4.4	Editor “Jython” Jedit	85
4.5	IDE Boa-Constructor	86
4.6	IDE Eric.	87
4.7	IDE Pydev	88
4.8	Diagrama de um repositório do Mercurial.	89
4.9	Operação de “commit”.	90
4.10	Repositório da Ana.	91

4.11	Modificações de Bruno.	92
4.12	Modificações de Ana.	92
4.13	Repositório atualizado de Bruno.	93
4.14	Repositório de Bruno após a fusão.	94
5.1	Comparação da performance entre o Python e o C (wave) para um loop simples. O eixo y está em escala logarítmica para facilitar a visualização das curvas.	103
5.2	Regra trapezoidal para integração aproximada de uma função.	113
5.3	Saída da listagem 5.25.	129
6.1	Diagrama de dispersão do exemplo do taxi. A reta superior descreve o cenário com bandeirada e a inferior, sem bandeirada.	139
6.2	População crescendo de forma linear, devido à chegada de um número constante de novos indivíduos por ano.	140
6.3	População diminui de forma linear, devido à saída de um número constante de indivíduos por ano.	141
6.4	Representação gráfica de um modelo linear. Efeito do parâmetro a .	142
6.5	Representação gráfica de um modelo linear. Efeito do parâmetro b .	143
6.6	Crescimento de uma população de bactérias por bipartição.	144
6.7	Modelos populacionais discretos. Modelo 2 à esquerda e modelo 1 à direita.	146
6.8	Gráfico com o resultado da integração do modelo 6.2.	153
7.1	Um grafo simples.	171
7.2	Diagrama de um grafo	174
7.3	Grafo, produto cartesiano e complemento.	177
7.4	Resultado da simulação do exemplo 7.5.	180
7.5	Rede social a partir de mensagens de email.	183

9.1	Amostragem ($n=20$) de uma distribuição normal com média 0 por LHS (histograma mais escuro) e amostragem padrão do scipy (histograma mais claro).	201
9.2	Distribuição acumulada da variável $x \sim N(0, 1)$, mostrando o particionamento do espaço amostral em segmentos equiprováveis.	202
9.3	Resultado da amostragem por envelope de uma distribuição beta(1,1)	211
9.4	Amostragem por rejeição da posterior Bayesiana.	214
9.5	Cadeia de Markov com kernel de transição exponencial.	216
9.6	Amostrador de Metropolis-Hastings($n=10000$).	220
9.7	Estrutura de dependência circular utilizada na listagem 9.7.	223
9.8	Amostrador de Gibbs aplicado a uma distribuição trivariada.	224
9.9	Processo markoviano e suas observações.	225
9.10	Função de verossimilhança para a comparação de sequências de DNA.	232
9.11	Log-verossimilhança da função descrita anteriormente. .	234
9.12	Resultado da listagem 9.21.	243
1	Konsole: janela contendo uma sessão bash (ou outra) no KDE.	255
2	Usando <code>spline</code>	262
3	Interpolando uma curva em um plano.	263
4	Atrator de Lorenz.	266

Listas de Tabelas

1.1	Métodos de Listas.	16
1.2	Métodos de Dicionários.	26
3.1	Principais comandos de plotagem do MPL	62
3.2	Argumentos que podem ser passados juntamente com a função plot para controlar propriedades de linhas.	66
3.3	Argumentos opcionais dos comandos de inserção de texto.	68
6.1	Valor de corridas de taxi	138
6.2	Dimensões básicas.	156
6.3	Dimensões derivadas.	157
6.4	Prefixos de Unidades	165

Listagens

1.1	O console interativo do Python	3
1.2	Palavras reservadas não podem ser utilizadas como nomes de variáveis	4
1.3	Definição da palavra reservada <code>for</code>	5
1.4	Usando o Python como uma calculadora.	6
1.5	Executando script.py via comando de linha.	6
1.7	Executando um script Python executável.	6
1.6	Tornando um script executável	7
1.8	Operações aritméticas	8
1.9	Atribuindo valores	9
1.10	Atribuindo o mesmo valor a múltiplas variáveis	9
1.11	Números complexos	10
1.12	Explorando números complexos	10
1.13	Atribuindo objetos a nomes (variáveis)	11
1.14	Exemplo de NameError	12
1.15	Criando listas.	14
1.16	Fatiando uma lista	14
1.17	Selecionando o final de uma lista.	15
1.18	selecionando todos os elementos de uma lista.	15
1.19	Adicionando elementos a listas.	16
1.20	Estendendo uma lista.	17
1.21	Efetuando buscas em listas.	17
1.22	Removendo elementos de uma lista.	18

1.23	Operações com listas	18
1.24	Criando listas numéricas sequenciais.	19
1.25	Definindo uma Tupla.	20
1.26	Outras formas de definir tuplas.	20
1.27	Strings.	22
1.28	Formatando strings.	23
1.29	Problemas com a concatenação de strings.	24
1.30	Criando e manipulando dicionários.	24
1.31	Iterando dicionários.	25
1.32	Exemplo do emprego de ramificações.	28
1.33	Implementando a funcionalidade de <code>if</code> e <code>elif</code> por meio de um dicionário.	29
1.34	Comandos de laço no Python.	30
1.35	A função <code>enumerate</code>	30
1.36	Iterando sobre mais de uma sequência.	31
1.37	Iterando sobre uma sequência ordenada.	31
1.38	Exemplo de uma exceção	32
1.39	Contornando uma divisão por zero	33
1.40	Lidando com diferentes tipos de erros.	34
1.41	Definindo uma função com um argumento obrigatório e outro opcional (com valor “ <code>default</code> ”).	35
1.42	lista de argumentos variável	36
1.43	Desempacotando argumentos.	36
1.44	Passando todos os argumentos por meio de um dicionário.	37
1.45	Retornando valores a partir de uma função.	37
1.46	Funções lambda	38
1.47	Geradores	38
1.48	Decorador básico.	39
1.49	Utilizando um decorador	39
1.50	Limpando a geração de decoradores	40
1.51	Usando Docstrings	41
1.52	Módulo exemplo	43
1.53	Executing a module from the console.	44

1.54	importing a package	45
1.55	Calculando e mostrando o determinante de uma matriz.	45
1.56	Listando as opções do Epydoc.	49
2.1	Definindo uma classe	53
2.2	Definindo atributos de uma classe	54
	caption=Instanciando uma classe.,label=ex:classe3	54
2.3	Passando atributos na criação de um objeto	54
2.4	Herança	56
2.5	Classe como uma estrutura de dados	57
3.1	Instalando o matplotlib	60
3.2	Criando um histograma no modo interativo	60
3.3	Gráfico de linha	63
3.4	Gráfico de pontos com valores de x e y especificados.	64
3.5	Figura com dois gráficos utilizando o comando subplot.	66
3.6	Formatando texto e expressões matemáticas	69
3.7	Plotando o globo terrestre	70
5.1	Otimização de loops com o weave	100
5.2	Calculando iterativamente a série de Fibonacci em Python e em C(<code>weave.inline</code>)	102
5.3	Executando o código do exemplo 5.2.	105
5.4	Carregando bibliotecas em C	106
5.5	Chamando funções em bibliotecas importadas com o ctypes	106
5.6	Calculando números primos em Pyrex	107
5.7	Gerando Compilando e linkando	108
5.8	Calculando números primos em Python	109
5.9	Comparando performances dentro do ipython	110
5.10	Automatizando a compilação de extensões em Pyrex por meio do setuptools	110
5.11	Utilizando Setuptools para compilar extensões em Pyrex	111
5.12	implementação da regra trapezoidal(utilizando laço for) conforme especificada na equação 5.3	112

5.13 Executando a listagem 5.12	114
5.14 Verificando o tempo de execução em C++	114
5.15 Código C++ gerado pelo Shed-skin a partir do script trapintloopy.py	115
5.16 implementação em Fortran da regra trapezoidal. label118	
5.17 Compilando e executando o programa da listagem ??	120
5.18 Compilando com f2py	121
5.19 Script para comparação entre Python e Fortran via f2py	121
5.20 Executando trapintloopcomp.py	123
5.21 Implementação vetorizada da regra trapezoidal . . .	123
5.22 setup.py para distribuir programas com extensões em Fortran	125
5.23 Usando o interpretador Jython	127
5.24 Um programa simples em Java usando a classe Swing.	127
5.25 Versão Jython do programa da listagem 5.24.	128
5.26 Criando um applet em Jython	128
5.27 Compilando appletp.py	129
5.28 Documento HTML contendo o “applet”	130
6.1 Dois modelos discretos de crescimento populacional .	147
6.2 Integrando um sistema de equações diferenciais or- dinárias	149
6.3 Integração numérica utilizando a classe ODE.	153
6.4 Criando novas unidades.	162
7.1 Definindo um grafo como um dicionário.	170
7.2 Obtendo a lista de vértices.	171
7.3 Definindo um grafo através de seus vértices	172
7.4 Diagrama de um grafo	172
7.5 Construindo um grafo dinamicamente	176
7.6 Construindo uma rede social a partir de e-mails . .	179
8.1 Exemplo de uso do módulo pickle	186
8.2 Módulos necessários	190
8.3 Especificando o banco de dados.	191
8.4 Especificando a tabela <code>ideia</code> do banco de dados. .	192

8.5	Restante do código da aranha.	193
9.1	Amostragem por Hipercubo latino (LHS)	201
9.2	Integração Monte Carlo	206
9.3	Amostragem por rejeição utilizando o método do envelope	209
9.4	Amostragem da Posterior Bayesiana por rejeição.	212
9.5	Cadeia de Markov com kernel de transição exponencial	215
9.6	Amostrador de Metropolis-Hastings	218
9.7	Amostrador de Gibbs aplicado à uma distribuição tri-variada	221
9.8	Filtro de partículas usando algoritmo SIR	225
9.9	Função de verossimilhança para x igual a 0.	230
9.10	Calculando a verossimilhança da comparação de duas sequências de DNA	231
9.11	Log-verossimilhança	234
9.12	Maximizando a verossimilhança	235
9.13	Classe Variável Aleatória Bayesiana – Requisitos.	236
9.14	Classe Variável Aleatória Bayesiana – Inicialização. .	237
9.15	Classe Variável Aleatória Bayesiana – Herança dinâmica de métodos.	238
9.16	Classe Variável Aleatória Bayesiana – Herança dinâmica de métodos.	238
9.17	Classe Variável Aleatória Bayesiana – Adição de dados e cálculo da Verossimilhança.	239
9.18	Classe Variável Aleatória Bayesiana – Amostras e PDF da <i>a priori</i>	240
9.19	Classe Variável Aleatória Bayesiana – Gerando a posterior.	241
9.20	Classe Variável Aleatória Bayesiana – Código de teste.	242
9.21	Classe Variável Aleatória Bayesiana – Listagem completa	244
22	Redirecionando STDIN e STDOUT	258
23	Lista ordenada dos usuários do sistema.	258
24	Plotando dados em um arquivo.	260

25	Desenhando um quadrado.	261
26	Uso do <code>spline</code>	261
27	Interpolando uma curva em um plano.	261
28	Resolvendo uma equação diferencial simples no con- sole do Linux.	264
29	Sistema de três equações diferenciais acopladas . . .	265

Prefácio: Computação Científica

*Da Computação Científica e sua definição pragmática.
Do porquê esta se diferencia, em metas e ferramentas,
da Ciência da Computação.*

COMPUTAÇÃO científica não é uma área do conhecimento muito bem definida. A definição utilizada neste livro é a de uma área de atividade/conhecimento que envolve a utilização de ferramentas computacionais (software) para a solução de problemas científicos em áreas da ciência não necessariamente ligadas à disciplina da ciência da computação, ou seja, a computação para o restante da comunidade científica.

Nos últimos tempos, a computação em suas várias facetas, tem se tornado uma ferramenta universal para quase todos os segmentos da atividade humana. Em decorrência, podemos encontrar produtos computacionais desenvolvidos para uma enorme variedade de aplicações, sejam elas científicas ou não. No entanto, a diversidade de aplicações científicas da computação é quase tão vasta quanto o próprio conhecimento humano. Por isso, o cientista freqüentemente se encontra com problemas para os quais as ferramentas computacionais adequadas não existem.

No desenvolvimento de softwares científicos, temos dois modos principais de produção de software: o desenvolvimento de softwares comerciais, feito por empresas de software que contratam progra-

madores profissionais para o desenvolvimento de produtos voltados para uma determinada aplicação científica, e o desenvolvimento feito por cientistas (físicos, matemáticos, biólogos, etc., que não são programadores profissionais), geralmente de forma colaborativa através do compartilhamento de códigos fonte.

Algumas disciplinas científicas, como a estatística por exemplo, representam um grande mercado para o desenvolvimento de softwares comerciais genéricos voltados para as suas principais aplicações, enquanto que outras disciplinas científicas carecem de massa crítica (em termos de número de profissionais) para estimular o desenvolvimento de softwares comerciais para a solução dos seus problemas computacionais específicos. Como agravante, o desenvolvimento lento e centralizado de softwares comerciais, tem se mostrado incapaz de acompanhar a demanda da comunidade científica, que precisa ter acesso a métodos que evoluem a passo rápido. Além disso, estão se multiplicando as disciplinas científicas que têm como sua ferramenta principal de trabalho os métodos computacionais, como por exemplo a bio-informática, a modelagem de sistemas complexos, dinâmica molecular e etc.

Cientistas que se vêem na situação de terem de desenvolver softwares para poder viabilizar seus projetos de pesquisa, geralmente têm de buscar uma formação improvisada em programação e produzem programas que tem como característica básica serem minimalistas, ou seja, os programas contêm o mínimo número de linhas de código possível para resolver o problema em questão. Isto se deve à conjugação de dois fatos: 1) O cientista raramente possui habilidades como programador para construir programas mais sofisticados e 2) Frequentemente o cientista dispõe de pouco tempo entre suas tarefas científicas para dedicar-se à programação.

Para complicar ainda mais a vida do cientista-programador, as linguagens de programação tradicionais foram projetadas e desenvolvidas por programadores para programadores e voltadas ao desenvolvimento de softwares profissionais com dezenas de milhares de linhas de código. Devido a isso, o número de linhas de código

mínimo para escrever um programa científico nestas linguagens é muitas vezes maior do que o número de linhas de código associado com a resolução do problema em questão.

Quando este problema foi percebido pelas empresas de software científico, surgiu uma nova classe de software, voltado para a demanda de cientistas que precisavam implementar métodos computacionais específicos e que não podiam esperar por soluções comerciais.

Esta nova classe de aplicativos científicos, geralmente inclui uma linguagem de programação de alto nível, por meio da qual os cientistas podem implementar seus próprios algoritmos, sem ter que perder tempo tentando explicar a um programador profissional o que, exatamente, ele deseja. Exemplos destes produtos incluem MATLABTM, MathematicaTM, MapleTM, entre outros. Nestes aplicativos, os programas são escritos e executados dentro do próprio aplicativo, não podendo ser executados fora dos mesmos. Estes ambientes, entretanto, não possuem várias características importantes das linguagens de programação: Não são portáteis, ou seja, não podem ser levados de uma máquina para a outra e executados a menos que a máquina-destino possua o aplicativo gerador do programa (MATLABTM, etc.) que custa milhares de dólares por licença, Os programas não podem ser portados para outra plataforma computacional para a qual não exista uma versão do aplicativo gerador. E, por último e não menos importante, o programa produzido pelo cientista não lhe pertence, pois, para ser executado, necessita de código proprietário do ambiente de desenvolvimento comercial.

Este livro se propõe a apresentar uma alternativa livre (baseada em Software Livre), que combina a facilidade de aprendizado e rapidez de desenvolvimento, características dos ambientes de desenvolvimento comerciais apresentados acima, com toda a flexibilidade das linguagens de programação tradicionais. Programas científicos desenvolvidos inteiramente com ferramentas de código aberto tem a vantagem adicional de serem plenamente escrutináveis pelo sis-

tema de revisão por pares (“peer review”), mecanismo central da ciência para validação de resultados.

A linguagem Python apresenta as mesmas soluções propostas pelos ambientes de programação científica, mantendo as vantagens de ser uma linguagem de programação completa e de alto nível.

Apresentando o Python

O Python é uma linguagem de programação dinâmica e orientada a objetos, que pode ser utilizada no desenvolvimento de qualquer tipo de aplicação, científica ou não. O Python oferece suporte à integração com outras linguagens e ferramentas, e é distribuído com uma vasta biblioteca padrão. Além disso, a linguagem possui uma sintaxe simples e clara, podendo ser aprendida em poucos dias. O uso do Python é frequentemente associado com grandes ganhos de produtividade e ainda, com a produção de programas de alta qualidade e de fácil manutenção.

A linguagem de programação Python¹ começou a ser desenvolvida ao final dos anos 80, na Holanda, por Guido van Rossum. Guido foi o principal autor da linguagem e continua até hoje desempenhando um papel central no direcionamento da evolução. Guido é reconhecido pela comunidade de usuários do Python como “Benevolent Dictator For Life” (BDFL), ou ditador benevolente vitalício da linguagem.

A primeira versão pública da linguagem (0.9.0) foi disponibilizada. Guido continuou avançando o desenvolvimento da linguagem, que alcançou a versão 1.0 em 1994. Em 1995, Guido emigrou para os EUA levando a responsabilidade pelo desenvolvimento do Python, já na versão 1.2, consigo. Durante o período em que Guido trabalhou para o CNRI², o Python atingiu a versão 1.6, que foi rapidamente seguida pela versão 2.0. A partir desta versão, o Python

¹www.python.org

²Corporation for National Research Initiatives

passa a ser distribuído sob a Python License, compatível com a GPL³, tornando-se oficialmente software livre. A linguagem passa a pertencer oficialmente à Python Software Foundation. Apesar da implementação original do Python ser desenvolvida na Linguagem C (CPython), Logo surgiram outras implementações da Linguagem, inicialmente em Java (Jython⁴), e depois na própria linguagem Python (Pypy⁵), e na plataforma .NET (IronPython⁶).

Dentre as várias características da linguagem que a tornam interessante para computação científica, destacam-se:

Multiplataforma: O Python pode ser instalado em qualquer plataforma computacional: Desde PDAs até supercomputadores com processamento paralelo, passando por todas as plataformas de computação pessoal.

Portabilidade: Aplicativos desenvolvidos em Python podem ser facilmente distribuídos para várias plataformas diferentes daquela em que foi desenvolvido, mesmo que estas não possuam o Python instalado.

Software Livre: O Python é software livre, não impondo qualquer limitação à distribuição gratuita ou venda de programas.

Extensibilidade: O Python pode ser extendido através de módulos, escritos em Python ou rotinas escritas em outras linguagens, tais como C ou Fortran (Mais sobre isso no capítulo 5).

Orientação a objeto: Tudo em Python é um objeto: funções, variáveis de todos os tipos e até módulos (programas escritos em Python) são objetos.

³GNU General Public License

⁴www.jython.org

⁵pypy.org

⁶www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython

Tipagem automática: O tipo de uma variável (string, inteiro, float, etc.) é determinado durante a execução do código; desta forma, você não necessita perder tempo definindo tipos de variáveis no seu programa.

Tipagem forte: Variáveis de um determinado tipo não podem ser tratadas como sendo de outro tipo. Assim, você não pode somar a string '123' com o inteiro 3. Isto reduz a chance de erros em seu programa. As variáveis podem, ainda assim, ser convertidas para outros tipos.

Código legível: O Python, por utilizar uma sintaxe simplificada e forçar a divisão de blocos de código por meio de indentação, torna-se bastante legível, mesmo para pessoas que não estejam familiarizadas com o programa.

Flexibilidade: O Python já conta com módulos para diversas aplicações, científicas ou não, incluindo módulos para interação com os protocolos mais comuns da Internet (FTP, HTTP, XMLRPC, etc.). A maior parte destes módulos já faz parte da distribuição básica do Python.

Operação com arquivos: A manipulação de arquivos, tais como a leitura e escrita de dados em arquivos texto e binário, é muito simplificada no Python, facilitando a tarefa de muitos pesquisadores ao acessar dados em diversos formatos.

Uso interativo: O Python pode ser utilizado interativamente, ou invocado para a execução de scripts completos. O uso interativo permite “experimentar” comandos antes de incluí-los em programas mais complexos, ou usar o Python simplesmente como uma calculadora.

etc...

Entretanto, para melhor compreender todas estas vantagens apresentadas, nada melhor do que começar a explorar exemplos

de computação científica na linguagem Python. Mas para inspirar o trabalho técnico, nada melhor do que um poema:

```
1 >>> import this
2 The Zen of Python, by Tim Peters
3
4 Beautiful is better than ugly.
5 Explicit is better than implicit.
6 Simple is better than complex.
7 Complex is better than complicated.
8 Flat is better than nested.
9 Sparse is better than dense.
10 Readability counts.
11 Special cases aren't special enough to break
    the rules.
12 Although practicality beats purity.
13 Errors should never pass silently.
14 Unless explicitly silenced.
15 In the face of ambiguity, refuse the
    temptation to guess.
16 There should be one—and preferably only
    one—obvious way to do it.
17 Although that way may not be obvious at
    first unless you're Dutch.
18 Now is better than never.
19 Although never is often better than *right*
    now.
20 If the implementation is hard to explain, it
    's a bad idea.
21 If the implementation is easy to explain, it
    may be a good idea.
22 Namespaces are one honking great idea — let
    's do more of those!
23 >>>
```

Usando este Livro

Este livro foi planejado visando a versatilidade de uso. Sendo assim, ele pode ser utilizado como livro didático (em cursos formais) ou como referência pessoal para auto-aprendizagem ou consulta.

Como livro didático, apresenta, pelo menos, dois níveis de aplicação possíveis:

1. Um curso introdutório à linguagem Python, no qual se faria uso dos capítulos da primeira parte. O único pré-requisito seria uma exposição prévia dos alunos a conceitos básicos de programação (que poderia ser condensada em uma única aula).
2. Um curso combinado de Python e computação científica. O autor tem ministrado este tipo de curso com grande sucesso. Este curso faria uso da maior parte do conteúdo do livro, o instrutor pode selecionar capítulos de acordo com o interesse dos alunos.

Como referência pessoal, este livro atende a um público bastante amplo, de leigos a cientistas. No início de cada capítulo encontram-se os pré-requisitos para se entender o seu conteúdo. Mas não se deixe inibir; as aplicações científicas são apresentadas juntamente com uma breve introdução à teoria que as inspira.

Recomendo aos auto-didatas que explorem cada exemplo contido no livro; eles ajudarão enormemente na compreensão dos tópicos apresentados⁷. Para os leitores sem sorte, que não dispõem de um computador com o sistema operacional GNU/Linux instalado, sugiro que o instalem, facilitará muito o acompanhamento dos exemplos. Para os que ainda não estão prontos para abandonar o

⁷O código fonte do exemplos está disponível na seguinte URL: http://fccoelho.googlepages.com/CCP_code.zip

WindowsTM, instalem o Linux em uma máquina virtual⁸! A distribuição que recomendo para iniciantes é o Ubuntu (www.ubuntu.com).

Enfim, este livro foi concebido para ser uma leitura prazeirosa para indivíduos curiosos como eu, que estão sempre interessados em aprender coisas novas!

Bom Proveito! Flávio Codeço Coelho

Petrópolis, 2007

⁸Recomendo o VirtualBox (www.virtualbox.org), é software livre e fantástico!

Parte I

Introdução ao Python e Ferramentas de Desenvolvimento

Capítulo 1

Fundamentos da Linguagem

Breve introdução a conceitos básicos de programação e à linguagem Python. A maioria dos elementos básicos da linguagem são abordados neste capítulo, com exceção de classes, que são discutidas em detalhe no capítulo 2.
Pré-requisitos: Conhecimentos básicos de programação em qualquer linguagem.

NESTE Capítulo, faremos uma breve introdução à linguagem Python. Esta introdução servirá de base para os exemplos dos capítulos subsequentes. Para uma introdução mais completa à linguagem, recomendamos ao leitor a consulta a livros e outros documentos voltados especificamente para programação em Python.

1.1 Primeiras impressões

Para uma primeira aproximação à linguagem, vamos examinar suas características básicas. Façamos isso interativamente, a partir do console Python. Vejamos como invocá-lo:

Listagem 1.1: O console interativo do Python

```
1 $ python
```

```
2 Python 2.5.1 (r251:54863, May 2 2007,  
16:56:35)  
3 [GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on  
linux2  
4 Type "help", "copyright", "credits" or "  
license" for more information.  
5 >>>
```

Toda linguagem, seja ela de programação ou linguagem natural, possui um conjunto de palavras que a caracteriza. As linguagens de programação tendem a ser muito mais compactas do que as linguagens naturais. O Python pode ser considerado uma linguagem compacta, mesmo em comparação com outras linguagens de programação.

As palavras que compõem uma linguagem de programação são ditas reservadas, ou seja, não podem ser utilizadas para nomear variáveis. Se o programador tentar utilizar uma das palavras reservadas como variável, incorrerá em um erro de sintaxe.

Listagem 1.2: Palavras reservadas não podem ser utilizadas como nomes de variáveis

```
1 >>> for=1  
2     File "<stdin>", line 1  
3         for=1  
4             ^  
5 SyntaxError: invalid syntax
```

A linguagem Python em sua versão atual (2.5), possui 30 palavras reservadas. São elas: `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while` e `yield`. Além destas palavras, existem constantes, tipos e funções internas ao Python, que estão disponíveis para a construção de programas. Estes elementos podem ser inspecionados através do comando `dir(__builtins__)`. Nenhum dos ele-

mentos do módulo `__builtins__` deve ser redefinido¹. O console interativo do Python possui um sistema de ajuda integrado que pode ser usado para acessar a documentação de qualquer elemento da linguagem. O comando `help()`, inicia a ajuda interativa. A partir daí, podemos por exemplo, digitar `keywords` para acessar a ajuda das palavras reservadas listadas acima. Se digitarmos `for` em seguida, obteremos a seguinte ajuda:

Listagem 1.3: Definição da palavra reservada `for`.

```
1 7.3 The for statement
2
3 The for statement is used to iterate over
4      the elements of a sequence
5      (such as a string, tuple or list) or other
          iterable object:
6 . . .
```

1.2 Uso Interativo *vs.* Execução a Partir de Scripts

Usuários familiarizados com ambientes de programação científicos tais como Matlab, R e similares, ficarão satisfeitos em saber que o Python também pode ser utilizado de forma interativa. Para isso, basta invocar o interpretador na linha de comando (Python shell, em Unix) ou invocar um shell mais sofisticado como o Idle, que vem com a distribuição padrão do Python, ou o Ipython (ver 4.1).

Tanto no uso interativo, como na execução a partir de scripts, o interpretador espera encontrar apenas uma expressão por linha do programa. Caso se deseje inserir mais de uma expressão em uma linha, as expressões devem ser separadas por `;`. Mas esta prática

¹Atenção, os componentes de `__builtins__`, não geram erros de sintaxe ao ser redefinidos.

deve ser evitada. Expressões podem continuar em outra linha se algum de seus parênteses, colchetes, chaves ou aspas ainda não tiver sido fechado. Alternativamente, linhas podem ser quebradas pela aposição do caractere \ ao final da linha.

Listagem 1.4: Usando o Python como uma calculadora.

```
1 >>> 1+1  
2 2  
3 >>>
```

No cabeçalho da shell do Python, acima (listagem 1.1), o interpretador identifica a versão instalada, data e hora em que foi compilada, o compilador C utilizado, detalhes sobre o sistema operacional e uma linhazinha de ajuda para situar o novato.

Para executar um programa, a maneira usual (em Unix) é digitar: `python script.py`. No Windows basta um duplo clique sobre arquivos `*.py`.

Listagem 1.5: Executando `script.py` via comando de linha.

```
1 $ python script.py
```

No Linux e em vários UNIXes, podemos criar scripts que são executáveis diretamente, sem precisar invocar o interpretador antes. Para isso, basta incluir a seguinte linha no topo do nosso script:

```
1 #! /usr/bin/env python
```

Note que os caracteres `#!` devem ser os dois primeiros caracteres do arquivo (como na listagem 1.6).

Depois, resta apenas ajustar as permissões do arquivo para que possamos executá-lo: `chmod +x script.py` (listagem 1.7).

Listagem 1.7: Executando um script Python executável.

```
1 $ chmod +x script.py
```

1.2. USO INTERATIVO VS. EXECUÇÃO A PARTIR DE SCRIPTS

7

Listagem 1.6: Tornando um script executável

```
1 #! /usr/bin/env python  
2  
3 print "Alô Mundo!"
```

```
2 $ ./script.py  
3 sys:1: DeprecationWarning: Non-ASCII  
    character '\xf4' in file ./teste on line  
    3, but no encoding declared; see http  
    ://www.python.org/peps/pep-0263.html for  
    details  
4 Alô Mundo!
```

Mas que lixo é aquele antes do nosso “**Alô mundo**”? Trata-se do interpretador reclamando do acento circunflexo em “**Alô**”. Para que o Python não reclame de acentos e outros caracteres da língua portuguesa não contidos na tabela ASCII, precisamos adicionar a seguinte linha ao script: `# -*- coding: latin-1 -*-`. Experimente editar o script acima e veja o resultado.

No exemplo da listagem 1.6, utilizamos o comando `print` para fazer com que nosso script produzisse uma string como saída, ou seja, para escrever no `stdout`². Como podemos receber informações pelo `stdin`? O Python nos oferece duas funções para isso: `input('texto')`, que executa o que o usuário digitar, sendo portanto perigoso, e `raw_input('texto')`, que retorna uma string com a resposta do usuário.

Nas listagens que se seguem, alternaremos entre a utilização de scripts e a utilização do Python no modo interativo (como na listagem 1.4). A presença do símbolo “`>>>`”, característico da shell

²Todos os processos no Linux e outros sistemas operacionais possuem vias de entrada e saída de dados denominados de `stdin` e `stdout`, respectivamente.

do Python será suficiente para diferenciar os dois casos. Exemplos de scripts virão dentro de caixas.

Operações com Números

Noventa e nove por cento das aplicações científicas envolvem algum tipo de processamento numérico. Vamos iniciar nosso contato com o Python através dos números (Listagem 1.8).

Listagem 1.8: Operações aritméticas

```
1 >>> 2+2
2 4
3 >>> # Comentário
4 ... 2*2
5 4
6 >>> 2**2 # Comentário na mesma linha
7 4
8 >>> (50-5*6)/4
9 5
10 >>> # Divisão de inteiros retorna "floor":
11 ... 7/3
12 2
13 >>> 7/-3
14 -3
15 >>> 7/3.
16 2.333333333333335
```

Nosso primeiro exemplo numérico (Listagem 1.8)³, trata números em sua representação mais simples: como constantes. É desta forma que utilizamos uma calculadora comum. Em programação é mais comum termos números associados a quantidades, a que

³Repare como o Python trata a divisão de dois inteiros. Ela retorna o resultado arredondado para baixo

1.2. USO INTERATIVO VS. EXECUÇÃO A PARTIR DE SCRIPTS

9

precisamos nos referenciar e que podem se modificar. Esta representação de números chama-se variável.

O sinal de = é utilizado para atribuir valores a variáveis(Listagem 1.9).

Listagem 1.9: Atribuindo valores

```
1 >>> largura = 20
2 >>> altura = 5*9
3 >>> largura * altura
4 900
```

Um valor pode ser atribuído a diversas variáveis com uma única operação de atribuição, ou múltiplos valores a múltiplas variáveis (Listagem 1.10). Note que no exemplo de atribuição de múltiplos valores a múltiplas variáveis (Listagem 1.10, linha 9) os valores poderiam estar em uma tupla.

Listagem 1.10: Atribuindo o mesmo valor a múltiplas variáveis

```
1 >>> x = y = z = 0 # Zero x, y e z
2 >>> x
3 0
4 >>> y
5 0
6 >>> z
7 0
8 >>> a,b,c=1,2,3
9 >>> a
10 1
11 >>> b
12 2
13 >>> c
14 3
```

O Python também reconhece números reais (ponto-flutuante) e complexos naturalmente. Em operações entre números reais e

inteiros o resultado será sempre real. Da mesma forma, operações entre números reais e complexos resultam sempre em um número complexo. Números complexos são sempre representados por dois números ponto-flutuante: a parte real e a parte imaginária. A parte imaginária é representada com um sufixo “j” ou “J”.

Listagem 1.11: Números complexos

```

1 >>> 1j * 1J
2 (-1+0j)
3 >>> 1j * complex(0,1)
4 (-1+0j)
5 >>> 3+1j*3
6 (3+3j)
7 >>> (3+1j)*3
8 (9+3j)
9 >>> (1+2j)/(1+1j)
10 (1.5+0.5j)
```

Um Número complexo para o Python, é um objeto⁴. Podemos extrair as partes componentes de um número complexo `c` utilizando atributos do tipo complexo: `c.real` e `c.imag`. A função `abs`, que retorna o módulo de um numero inteiro ou real, retorna o comprimento do vetor no plano complexo, quando aplicada a um número complexo. O módulo de um número complexo é também denominado magnitude (listagem 1.12).

Listagem 1.12: Explorando números complexos

```

1 >>> a=3.0+3.0j
2 >>> a.real
3 3.0
4 >>> a.imag
5 3.0
6 >>> abs(a) # sqrt(a.real**2 + a.imag**2)
```

⁴ Assim como os outros tipos de números.

```
7 4.2426406871192848
```

1.3 Nomes, Objetos e Espaços de Nomes

Nomes em Python são identificadores de objetos, e também são chamados de variáveis. Nomes devem ser iniciados por letras maiúsculas ou minúsculas e podem conter algarismos, desde que não sejam o primeiro caractere. O Python faz distinção entre maiúsculas e minúsculas portanto, `nome != Nome`.

No Python, todos os dados são objetos tipados, que são associados dinamicamente a nomes. O sinal de igual (`=`), liga o resultado da avaliação da expressão do seu lado direito a um nome situado à sua esquerda. A esta operação damos o nome de atribuição (ver exemplo na listagem 1.13).

Listagem 1.13: Atribuindo objetos a nomes (variáveis)

```
1 >>> a=3*2**7
2 >>> a , b = ( 'laranja' , 'banana' )
```

As variáveis criadas por atribuição ficam guardadas na memória do computador. Para evitar preenchimento total da memória, assim que um objeto deixa de ser referenciado por um nome (deixa de existir no espaço de nomes corrente), ele é imediatamente apagado da memória pelo interpretador.

O conceito de espaço de nomes é uma característica da linguagem Python que contribui para sua robustez e eficiência. Espaços de nomes são dicionários (ver 1.4) contendo as variáveis, objetos e funções disponíveis durante a execução de um programa. A um dado ponto da execução de um programa, existem sempre dois dicionários disponíveis para a resolução de nomes: um local e um global. Estes dicionários podem ser acessados para leitura através das funções `locals()` e `globals()`, respectivamente. Sempre que o interpretador Python encontra uma palavra que não pertence ao

conjunto de palavras reservadas da linguagem, ele a procura, primeiro no espaço de nomes local e depois no global. Se a palavra não é encontrada, um erro do tipo `NameError` é acionado (exemplo 1.14).

Listagem 1.14: Exemplo de `NameError`

```

1 >>> maria
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in ?
4     NameError: name 'maria' is not defined

```

O espaço de nomes local, muda ao longo da execução de um programa. Toda a vez que a execução adentra uma função, o espaço de nomes local passa a refletir apenas as variáveis definidas dentro daquela função⁵. Ao sair da função, o dicionário local torna-se igual ao global.

```

1 >>> a=1
2 >>> len(globals().items())
3 4
4 >>> len(locals().items())
5 4
6 >>> def fun():
7     ...     a='novo valor'
8     ...     print len(locals().items())
9     ...     print a
10 ...
11 >>> fun()
12 1
13 novo valor
14 >>> print a
15 1
16 >>> len(locals().items())

```

⁵Mais quaisquer variáveis explicitamente definidas como globais

```
17 5
18 >>> locals()
19 { '__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'fun': <function fun at 0xb7c18ed4>, '__doc__': None, 'a': 1}
```

Também é importante lembrar que o espaço de nomes local sempre inclui os `__builtins__` como vemos na listagem 1.3.

1.4 Estruturas de Dados

Qualquer linguagem de programação pode ser simplisticamente descrita como uma ferramenta, através da qual, dados e algoritmos são implementados e interagem para a solução de um dado problema. Nesta seção vamos conhecer os tipos e estruturas de dados do Python para que possamos, mais adiante, utilizar toda a sua flexibilidade em nossos programas.

No Python, uma grande ênfase é dada à simplicidade e à flexibilidade de forma a maximizar a produtividade do programador. No tocante aos tipos e estruturas de dados, esta filosofia se apresenta na forma de uma tipagem dinâmica, porém forte. Isto quer dizer que os tipos das variáveis não precisam ser declarados pelo programador, como é obrigatório em linguagens de tipagem estática como o C, FORTRAN, Visual Basic, etc. Os tipos das variáveis são inferidos pelo interpretador. As principais estruturas de dados como **Listas** e **Dicionários**, podem ter suas dimensões alteradas, dinamicamente durante a execução do Programa , o que facilita enormemente a vida do programador, como veremos mais adiante.

Listas

As listas formam o tipo de dados mais utilizado e versátil do Python. Listas são definidas como uma sequência de valores se-

parados por vírgulas e delimitada por colchetes:

Listagem 1.15: Criando listas.

```
1 >>> lista =[1 , 'a' , 'pe' ]
2 >>> lista
3 [1 , 'a' , 'pe' ]
4 >>> lista [0]
5 1
6 >>> lista [2]
7 'pe'
8 >>> lista [-1]
9 'pe'
```

Na listagem 1.15, criamos uma lista de três elementos. Uma lista é uma sequência ordenada de elementos, de forma que podemos selecionar elementos de uma lista por meio de sua posição. Note que o primeiro elemento da lista é `lista[0]`. Todas as contagens em Python começam em 0.

Uma lista também pode possuir elementos de tipos diferentes. Na listagem 1.15, o elemento 0 é um inteiro enquanto que os outros elementos são strings. Para verificar isso, digite o comando `type(lista[0])`.

Uma característica muito interessante das listas do Python, é que elas podem ser indexadas de trás para frente, ou seja, `lista[-1]` é o último elemento da lista. Como listas são sequências de tamanho variável, podemos assessorar os últimos **n** elementos, sem ter que contar os elementos da lista.

Listas podem ser “fatiadas”, ou seja, podemos selecionar uma porção de uma lista que contenha mais de um elemento.

Listagem 1.16: Fatiando uma lista

```
1 >>> lista =[ 'a' , 'pe' , 'que' , 1]
2 >>> lista [1:3]
3 [ 'pe' , 'que' ]
```

```
4 >>> lista[-1]
5 1
6 >>> lista[3]
7 1
8 >>>
```

O comando `lista[1:3]`, delimita uma “fatia” que vai do elemento 1 (o segundo elemento) ao elemento imediatamente anterior ao elemento 3. Note que esta seleção inclui o elemento correspondente ao limite inferior do intervalo, mas não o limite superior. Isto pode gerar alguma confusão, mas tem suas utilidades. Índices negativos também podem ser utilizados nestas expressões.

Para retirar uma fatia que inclua o último elemento, temos que usar uma variação deste comando seletor de intervalos:

Listagem 1.17: Selecionando o final de uma lista.

```
1 >>> lista[2:]
2 [ 'que' , 1]
3 >>>
```

Este comando significa todos os elementos a partir do elemento 2 (o terceiro), até o final da lista. Este comando poderia ser utilizado para selecionar elementos do início da lista: `lista[:3]`, só que desta vez não incluindo o elemento 3 (o quarto elemento).

Se os dois elementos forem deixados de fora, são selecionados todos os elementos da lista:

Listagem 1.18: selecionando todos os elementos de uma lista.

```
1 >>> lista[:]
2 [ 'a' , 'pe' , 'que' , 1]
3 >>>
```

Só que não é a mesma lista, é uma nova lista com os mesmos elementos. Desta forma, `lista[:]` é uma maneira de fazer uma cópia completa de uma lista. Normalmente este recurso é utilizado junto com uma atribuição `a = lista[:]`.

Listagem 1.19: Adicionando elementos a listas.

```

1 >>> lista [:]
2 [ 'a' , 'pe' , 'que' , 1]
3 >>> lista.append(2) #adiciona 2 ao final
4 [ 'a' , 'pe' , 'que' , 1, 2]
5 >>> lista.insert(2,[ 'a' , 'b' ])
6 >>> lista
7 [ 'a' , 'pe' , [ 'a' , 'b' ] , 'que' , 1, 2]
8 >>>

```

As listas são conjuntos mutáveis, ao contrário de tuplas e strings, portanto pode-se adicionar(listagem 1.19), modificar ou remover (tabela 1.1) elementos de uma lista.

Tabela 1.1: Métodos de Listas.

Método	Efeito
L.append(objeto)	Adiciona um elemento ao final da lista.
L.count(elemento)	Retorna o número de ocorrências do elemento.
L.extend(list)	Concatena listas.
L.index(value)	Retorna índice da primeira ocorrência do valor.
L.insert (i, o)	Insere objeto (o) antes de índice (i).
L.pop([índice])	Remove e retorna objeto no índice. ou o último elemento.
L.remove(value)	remove primeira ocorrência do valor.
L.reverse()	Inverte lista. <i>In situ</i>
L.sort()	Ordena lista. <i>In loco</i>

Note que as operações *in situ* não alocam memória extra para a operação, ou seja, a inversão ou a ordenação descritas na tabela 1.1, são realizadas no mesmo espaço de memória da lista original. Operações *in situ* alteram a variável em si sem fazer uma cópia da mesma e, portanto não retornam nada.

O método L.insert insere um objeto antes da posição indicada pelo índice. Repare, na listagem 1.19, que o objeto em questão era

uma lista, e o método `insert` não a fundiu com a lista original. Este exemplo nos mostra mais um aspecto da versatilidade do objeto lista, que pode ser composto por objetos de qualquer tipo.

Listagem 1.20: Estendendo uma lista.

```

1 >>> lista2=[ 'a' , 'b' ]
2 >>> lista . extend ( lista2 )
3 >>> lista
4 [ 'a' , 'pe' , [ 'a' , 'b' ] , 'que' , 1 , 2 , 'a' , 'b'
   ]

```

Já na listagem 1.20, os elementos da segunda lista são adicionados, individualmente, ao final da lista original.

Listagem 1.21: Efetuando buscas em listas.

```

1 >>> lista . index ( 'que' )
2 3
3 >>> lista . index ( 'a' )
4 0
5 >>> lista . index ( 'z' )
6 Traceback (most recent call last):
7   File "<input>", line 1, in ?
8     ValueError: list . index (x): x not in list
9 >>> 'z' in lista
10 0

```

Conforme ilustrado na listagem 1.21, o método `L.index` retorna o índice da primeira ocorrência do valor dado. Se o valor não existir, o interpretador retorna um `ValueError`. Para testar se um elemento está presente em uma lista, pode-se utilizar o comando `in`⁶ como ilustrado na listagem 1.21. Caso o elemento faça parte da lista, este comando retornará 1, caso contrário retornará 0⁷.

⁶O inverso do operador `in`, é o operador `not in` e também é válido para todas as sequências.

⁷**Verdadeiro e falso:** Em Python, quase qualquer coisa pode ser utilizada em um contexto booleano, ou seja, como verdadeiro ou falso. Por exemplo 0 é

Existem dois métodos básicos para remover elementos de uma lista: `L.remove` e `L.pop` — listagem 1.22. O primeiro remove o elemento nomeado sem nada retornar, o segundo elimina e retorna o último ou o elemento da lista (se chamado sem argumentos), ou o determinado pelo índice, passado como argumento (Listagem 1.22).

Listagem 1.22: Removendo elementos de uma lista.

```

1 >>> lista .remove("que")
2 >>> lista
3 [ 'a' , 'pe' , [ 'a' , 'b' ] , 1 , 2 , 'a' , 'b' ]
4 >>> lista .pop(2)
5 [ 'a' , 'b' ]
6 >>> lista
7 [ 'a' , 'pe' , 1 , 2 , 'a' , 'b' ]
8 >>>

```

Operadores aritméticos também podem ser utilizados para operações com listas. O operador de soma, “+”, concatena duas listas. O operador “+=” é um atalho para o método `L.extend` conforme mostrado na listagem 1.23.

Listagem 1.23: Operações com listas

```

1 >>> lista =[ 'a' , 'pe' , 1 , 2 , 'a' , 'b' ]
2 >>> lista = lista + [ 'novo' , 'elemento' ]
3 >>> lista
4 [ 'a' , 'pe' , 1 , 2 , 'a' , 'b' , 'novo' , ' '
   elemento ]
5 >>> lista += 'dois'
6 >>> lista
7 [ 'a' , 'pe' , 1 , 2 , 'a' , 'b' , 'd' , 'o' , 'i' , ' '
   s ]
8 >>> lista += [ 'dois' ]

```

falso enquanto que todos os outros números são verdadeiros. Uma string, lista, dicionário ou tupla vazias são falsas enquanto que as demais são verdadeiras.

Listagem 1.24: Criando listas numéricas sequenciais.

```
1 >>> range(10)
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 >>> range(2,20,2)#números pares
4 [2, 4, 6, 8, 10, 12, 14, 16, 18]
5 >>> range(1,20,2)#números ímpares
6 [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
9 >>> lista
10 ['a', 'pe', 1, 2, 'a', 'b', 'd', 'o', 'i',
     's', 'dois']
11 >>> li=[1,2]
12 >>> li*3
13 [1, 2, 1, 2, 1, 2]
14 >>>
```

Note que a operação `lista = lista + lista2` cria uma nova `lista` enquanto que o comando `+=` aproveita a lista original e a extende. Esta diferença faz com que o operador `+=` seja muito mais rápido, especialmente para grandes listas. O operador de multiplicação, “`*`”, é um repetidor/concatenador de listas conforme mostrado ao final da listagem 1.23. A operação de multiplicação *in situ* (`*=`) também é válida.

Um tipo de lista muito útil em aplicações científicas, é lista numérica sequencial. Para construir estas listas podemos utilizar o comando `range` (exemplo 1.24). O comando `range` aceita 1, 2 ou três argumentos: início, fim e passo, respectivamente (ver exemplo 1.24).

Tuplas

Uma tupla, é uma lista imutável, ou seja, ao contrário de uma lista, após a sua criação, ela não pode ser alterada. Uma tupla é definida de maneira similar a uma lista, com exceção dos delimitadores do conjunto de elementos que no caso de uma tupla são parênteses (listagem 1.25).

Listagem 1.25: Definindo uma Tupla.

```

1 >>> tu = ('Genero', 'especie', 'peso', ,
2           'estagio')
3 >>> tu[0]
4   'Genero'
5 >>> tu[1:3]
6   ('especie', 'peso')
7 >>>

```

Os elementos de uma tupla podem ser referenciados através de índices, (posição) de forma idêntica a como é feito em listas. Tuplas também podem ser fatiadas, gerando outras tuplas.

As tuplas não possuem métodos. Isto se deve ao fato de as tuplas serem imutáveis. Os métodos `append`, `extend`, e `pop` naturalmente não se aplicam a tuplas, uma vez que não se pode adicionar ou remover elementos de uma tupla. Não podemos fazer busca em tuplas, visto que não dispomos do método `index`. No entanto, podemos usar `in` para determinar se um elemento existe em uma tupla, como se faz em listas.

Listagem 1.26: Outras formas de definir tuplas.

```

1 >>> tu=()
2 >>> tu
3   ()
4 >>> tu='casa', #—Repare na vírgula ao
5           final!
6 >>> tu

```

```
6 ('casa',)
7 >>> tu=1,2,3,4
8 >>> tu
9 (1, 2, 3, 4)
10 >>> var =w,x,y,z
11 >>> var
12 (w,x,y,z)
13 >>> var = tu
14 >>> w
15 1
16 >>> x
17 2
18 >>> y
19 3
20 >>> z
21 4
```

Conforme exemplificado em 1.26, uma tupla vazia, é definida pela expressão (), já no caso de uma tupla unitária, isto é, com apenas um elemento, fazemos a atribuição com uma vírgula após o elemento, caso contrário (`tu=('casa')`), o interpretador não poderá distinguir se os parênteses estão sendo utilizados como delimitadores normais ou delimitadores de tupla. O comando `tu=('casa',)` é equivalente ao apresentado na quarta linha da listagem 1.26, apenas mais longo.

Na sétima linha da listagem 1.26, temos uma extensão do conceito apresentado na linha anterior: a definição de uma tupla sem a necessidade de parênteses. A este processo, se dá o nome de *empacotamento de sequência*. O empacotamento de vários elementos sempre gera uma tupla.

As tuplas, apesar de não serem tão versáteis quanto as listas, são mais rápidas. Portanto, sempre que se precisar de uma sequência de elementos para servir apenas de referência, sem a necessidade de edição, deve-se utilizar uma tupla. Tuplas também são úteis na

formatação de strings como veremos na listagem 1.28.

Apesar das tuplas serem imutáveis, pode-se contornar esta limitação fatiando e concatenando tuplas. Listas também podem ser convertidas em tuplas, com a função `tuple(lista)`, assim como tuplas podem ser convertidas em listas através da função `list(tupla)`.

Uma outra aplicação interessante para tuplas, mostrada na listagem 1.26, é a atribuição múltipla, em que uma tupla de valores, é atribuída a uma lista de nomes de variáveis armazenados em uma tupla. Neste caso, as duas sequências devem ter, exatamente, o mesmo número de elementos.

Strings

Strings são um terceiro tipo de sequências em Python. Strings são sequências de caracteres delimitados por aspas simples, '`string345`', ou duplas "`string`". Todos os operadores discutidos até agora para outras sequências, tais como `+`, `*`, `in`, `not in`, `s[i]` e `s[i:j]`, também são válidos para strings. Strings também podem ser definidas com três aspas (duplas ou simples). Esta última forma é utilizada para definir strings contendo quebras de linha.

Listagem 1.27: Strings.

```
1 >>> st='123 de oliveira4'
2 >>> len(st)
3 16
4 >>> min(st)
5   ,
6 >>> max(st)
7   'v'
8 >>> texto = """primeira linha
9 segunda linha
10 terceira linha"""
11 >>> print(texto)
```

```
12 primeira linha  
13 segunda linha  
14 terceira linha
```

Conforme ilustrado na listagem 1.27, uma string é uma sequência de quaisquer caracteres alfanuméricos, incluindo espaços. A função `len()`, retorna o comprimento da string, ou de uma lista ou tupla. As funções `min()` e `max()` retornam o valor mínimo e o máximo de uma sequência, respectivamente. Neste caso, como a sequência é uma string, os valores são os códigos ASCII de cada caractere. Estes comandos também são válidos para listas e tuplas.

O tipo String possui 33 métodos distintos (na versão 2.2.1 do Python). Seria por demais enfadonho listar e descrever cada um destes métodos neste capítulo. Nesta seção vamos ver alguns métodos de strings em ação no contexto de alguns exemplos. Outros métodos aparecerão em outros exemplos nos demais capítulos.

O uso mais comum dado a strings é a manipulação de textos que fazem parte da entrada ou saída de um programa. Nestes casos, é interessante poder montar strings, facilmente, a partir de outras estruturas de dados. Em Python, a inserção de valores em strings envolve o marcador `%s`.

Listagem 1.28: Formatando strings.

```
1 >>> animal='Hamster 1'  
2 >>> peso=98  
3 >>> '%s: %s gramas' % (animal, peso)  
4 'Hamster 1: 98 gramas'
```

Na listagem 1.28, temos uma expressão de sintaxe não tão óbvia mas de grande valor na geração de strings. O operador `%` (módulo), indica que os elementos da tupla seguinte serão mapeados, em sequência, nas posições indicadas pelos marcadores `%s` na string.

Esta expressão pode parecer uma complicação desnecessária para uma simples concatenação de strings. Mas não é. Vejamos porquê:

Listagem 1.29: Problemas com a concatenação de strings.

```

1 >>> animal='Hamster 1'
2 >>> peso=98
3 >>> '%s: %s gramas' % (animal, peso)
4 'Hamster 1: 98 gramas'
5 >>> animal+': '+peso+' gramas'
6 Traceback (most recent call last):
7   File "<input>", line 1, in ?
8     TypeError: cannot concatenate 'str' and 'int'
      objects
9 >>>

```

Pelo erro apresentado na listagem 1.29, vemos que a formatação da string utilizando o operador módulo e os marcadores **%s**, faz mais do que apenas concatenar strings, também converte a variável **peso** (inteiro) em uma string.

Dicionários

O dicionário é um tipo de dado muito interessante do Python: É uma estrutura que funciona como um banco de dados em miniatura, no sentido de que seus elementos consistem de pares “**chave : valor**”, armazenados sem ordenação. Isto significa que não existem índices para os elementos de um dicionário, a informação é acessada através das chaves.

Listagem 1.30: Criando e manipulando dicionários.

```

1 >>> Z={'C':12, 'O':16, 'N':12, 'Na':40}
2 >>> Z[ 'O']
3 16
4 >>> Z[ 'H']=1
5 >>> Z
6 { 'Na': 40, 'C': 12, 'H': 1, 'O': 16, 'N': 12}

```

```
7 >>> Z.keys()
8 [ 'Na' , 'C' , 'H' , 'O' , 'N' ]
9 >>> Z.has_key( 'N' )
10 1
```

As chaves podem ser de qualquer tipo imutável: números, strings, tuplas (que contenham apenas tipos imutáveis). Dicionários possuem os métodos listados na tabela 1.2.

Os conjuntos (chave:valor) são chamados de ítems do dicionários. Esta terminologia é importante pois podemos acessar, separadamente, chaves, valores ou ítems de um dicionário.

Os valores de um dicionário podem ser de qualquer tipo, números, strings, listas, tuplas e até mesmo outros dicionários. Também não há qualquer restrição para o armazenamento de diferentes tipos de dados em um mesmo dicionário.

Conforme exemplificado em 1.30, pode-se adicionar novos ítems a um dicionário, a qualquer momento, bastando atribuir um valor a uma chave. Contudo, é preciso ter cuidado. Se você tentar criar um ítem com uma chave que já existe, o novo ítem substituirá o antigo.

Os métodos `D.iteritems()`, `D.iterkeys()` e `D.itervalues()` criam iteradores. Iteradores permitem iterar através dos ítems, chaves ou valores de um dicionário. Veja a listagem 1.31:

Listagem 1.31: Iterando dicionários.

```
1 >>> Z.items()
2 [ ( 'Na' , 40) , ( 'C' , 12) , ( 'H' , 1) , ( 'O' , 16) ,
   ( 'N' , 12) ]
3 >>> i=Z.iteritems()
4 >>> i
5 <dictionary-iterator object at 0x8985d00>
6 >>> i.next()
7 ( 'Na' , 40)
8 >>> i.next()
```

Tabela 1.2: Métodos de Dicionários.

Método	Efeito
D.clear()	Remove todos os itens do dicionário.
D.copy()	Cria uma cópia de um dicionário.
D.get(k[, d])	Retorna D[k], se a chave k existir. Senão, d.
D.has_key(k)	Retorna 1 se D possuir a chave k.
D.items()	Retorna lista de tuplas (chave:valor).
D.iteritems()	Retorna objeto iterador para D.
D.iterkeys()	Idem para chaves.
D.itervalues()	Idem para valores.
D.keys()	Retorna lista com todas as chaves.
D.popitem()	Remove e retorna um ítem (chave:valor).
D.update(E)	Copia itens de E para D.
D.values()	Retorna lista com todas os valores.

```

9  ('C', 12)
10 >>> # e assim por diante ...
11 >>> k=Z.iterkeys()
12 >>> k.next()
13 'Na'
14 >>> k.next()
15 'C'
16 >>> k.next()
17 'H'
18 >>> k.next()
19 'O'
20 >>> k.next()
21 'N'
22 >>> k.next()
23 Traceback (most recent call last):
24   File "<input>", line 1, in ?
25 StopIteration

```

O uso de iteradores é interessante quando se precisa acessar o conteúdo de um dicionário, elemento-a-elemento, sem repetição. Ao final da iteração, o iterador retorna um aviso: `StopIteration`.

Conjuntos

Reafirmando sua vocação científica, a partir da versão 2.4, uma estrutura de dados para representar o conceito matemático de conjunto foi introduzida na linguagem Python. Um conjunto no Python é uma coleção de elementos sem ordenação e sem repetições. O objeto conjunto em Python aceita operações matemáticas de conjuntos tais como união, interseção, diferença e diferença simétrica (exemplo 1.4).

```

1 >>> a = set('pirapora')
2 >>> b = set('paranapanema')
3 >>> a #letras em a
4 set(['i', 'p', 'r', 'a', 'o'])
5 >>> a - b #Letras em a mas não em b
6 set(['i', 'o'])
7 >>> a | b #letras em a ou b
8 set(['a', 'e', 'i', 'm', 'o', 'n', 'p', 'r'])
9 >>> a & b #letras em a e b
10 set(['a', 'p', 'r'])
11 >>> a ^ b #letras em a ou b mas não em ambos
12 set(['i', 'm', 'e', 'o', 'n'])
```

No exemplo 1.4 pode-se observar as seguintes correspondências entre a notação do Python e a notação matemática convencional:

a - b: $A - B$ ⁸

a | b: $A \cup B$

⁸Por convenção representa-se conjuntos por letras maiúsculas.

a & b: $A \cap B$

a ^ b: $(A \cup B) - (A \cap B)$

1.5 Controle de fluxo

Em condições normais o interpretador executa as linhas de um programa uma a uma. As exceções a este caso são linhas pertencentes à definição de função e classe, que são executadas apenas quando a respectiva função ou classe é chamada. Entretanto algumas palavras reservadas tem o poder de alterar a direção do fluxo de execução das linhas de um programa.

Condições

Toda linguagem de programação possui estruturas condicionais que nos permitem representar decisões: “se isso, faça isso, caso contrário faça aquilo”. Estas estruturas também são conhecidas por ramificações. O Python nos disponibiliza três palavras reservadas para este fim: `if`, `elif` e `else`. O seu uso é melhor demonstrado através de um exemplo (Listagem 1.32).

Listagem 1.32: Exemplo do emprego de ramificações.

```
1 if a == 1:  
2     #este bloco é executado se a for 1  
3     pass  
4 elif a == 2:  
5     #este bloco é executado se a for 2  
6     pass  
7 else:  
8     #este bloco é executado se  
9     #se se nenhum dos blocos  
10    #anteriores tiver sido executado  
11    pass
```

No exemplo 1.32, vemos também emprego da palavra reservada `pass`, que apesar de não fazer nada é muito útil quando ainda não sabemos quais devem ser as consequências de determinada condição.

Uma outra forma elegante e compacta de implementar uma ramificação condicional da execução de um programa é através de dicionários (Listagem 1.33). As condições são as chaves de um dicionário cujos valores são funções. Esta solução não contempla o `else`, porém.

Listagem 1.33: Implementando a funcionalidade de `if` e `elif` por meio de um dicionário.

```
1 desfechos = {1:fun1 ,2:fun2}
2 desfechos [a]
```

Iteeração

Muitas vezes em problemas computacionais precisamos executar uma tarefa, repetidas vezes. Entretanto não desejamos ter que escrever os mesmos comandos em sequência, pois além de ser uma tarefa tediosa, iria transformar nosso “belo” programa em algo similar a uma lista telefônica. A solução tradicional para resolver este problema é a utilização de laços (loops) que indicam ao interpretador que ele deve executar um ou mais comandos um número arbitrário de vezes. Existem vários tipos de laços disponíveis no Python.

O laço while: O laço `while` repete uma tarefa enquanto uma condição for verdadeira (Listagem 1.34). Esta tarefa consiste em um ou mais comandos indentados em relação ao comando que inicia o laço. O fim da indentação indica o fim do bloco de instruções que deve ser executado pelo laço.

Listagem 1.34: Comandos de laço no Python.

```
1 >>> while True:  
2         pass#repete indefinidamente  
3 >>> i=0  
4 >>> while i < 10:  
5         i +=1  
6         print i  
7 # saida omitida  
8 >>> for i in range(1):  
9         print i
```

O laço for: O laço `for` nos permite iterar sobre uma sequência atribuindo os elementos da mesma a uma variável, sequencialmente, à medida que prossegue. Este laço se interrompe automaticamente ao final da sequência.

Iteração avançada: O Python nos oferece outras técnicas de iteração sobre sequências que podem ser bastante úteis na redução da complexidade do código. No exemplo 1.31 nós vimos que dicionários possuem métodos específicos para iterar sobre seus componentes. Agora suponhamos que desejássemos iterar sobre uma lista e seu índice?

Listagem 1.35: A função enumerate.

```
1 >>> for n,e in enumerate(['a','b','c','d','e']):  
2         print "%s: %s"%(n,e)  
3  
4 0: a  
5 1: b  
6 2: c  
7 3: d  
8 4: e
```

A função `enumerate` (exemplo 1.35) gera um iterador similar ao visto no exemplo 1.31. O laço `for` chama o método `next` deste iterador repetidas vezes, até que receba a mensagem `StopIteration` (ver exemplo 1.31).

O comando `zip` nos permite iterar sobre um conjunto de seqüências pareando sequencialmente os elementos das múltiplas listas (exemplo 1.36).

Listagem 1.36: Iterando sobre mais de uma sequência.

```
1 >>> perguntas = [ 'nome' , 'cargo' , 'partido' ]
2 >>> respostas = [ 'Lula' , 'Presidente' , 'PT' ]
3 >>> for p, r in zip (perguntas , respostas):
4         print "qual o seu %s? %s"%(p , r)
5
6
7 qual o seu nome? Lula
8 qual o seu cargo? Presidente
9 qual o seu partido? PT
```

Podemos ainda desejar iterar sobre uma sequência em ordem reversa (exemplo 1.5), ou iterar sobre uma sequência ordenada sem alterar a sequência original (exemplo 1.37). Note que no exemplo 1.37, a lista original foi convertida em um conjunto (`set`) para eliminar as repetições.

```
1 >>> for i in reversed (range (5)):
2         print i
3 4
4 3
5 2
6 1
7 0
```

Listagem 1.37: Iterando sobre uma sequência ordenada.

```
1 >>> for i in sorted (set (1)):
```

```
2         print i
3 laranja
4 leite
5 manga
6 ovos
7 uva
```

Iterações podem ser interrompidas por meio da palavra reservada `break`. Esta pode ser invocada quando alguma condição se concretiza. Podemos também saltar para a próxima iteração (sem completar todas as instruções do bloco) por meio da palavra reservada `continue`. A palavra reservada `else` também pode ser aplicada ao final de um bloco iterativo. Neste caso o bloco definido por `else` só será executado se a iteração se completar normalmente, isto é, sem a ocorrência de `break`.

Lidando com erros: Exceções

O método da tentativa e erro não é exatamente aceito na ortodoxia científica mas, frequentemente, é utilizado no dia a dia do trabalho científico. No contexto de um programa, muitas vezes somos forçados a lidar com possibilidades de erros e precisamos de ferramentas para lidar com eles.

Muitas vezes queremos apenas continuar nossa análise, mesmo quando certos erros de menor importância ocorrem; outras vezes, o erro é justamente o que nos interessa, pois nos permite examinar casos particulares onde nossa lógica não se aplica.

Como de costume o Python nos oferece ferramentas bastante intuitivas para interação com erros⁹.

Listagem 1.38: Exemplo de uma exceção

```
1 >>> 1/0
```

⁹Os erros tratados nesta seção não são erros de sintaxe mas erros que ocorrem durante a execução de programas sintaticamente corretos. Estes erros serão denominados **exceções**

```
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in ?
4 ZeroDivisionError: integer division or
      modulo by zero
```

Suponhamos que você escreva um programa que realiza divisões em algum ponto, e dependendo dos dados fornecidos ao programa, o denominador torna-se zero. Como a divisão por zero não é possível, o seu programa para, retornando uma mensagem similar a da listagem 1.38. Caso você queira continuar com a execução do programa apesar do erro, poderíamos solucionar o problema conforme o exposto na listagem 1.39

Listagem 1.39: Contornando uma divisão por zero

```
1 >>> for i in range(5):
2     ...     try:
3     ...         q=1./i
4     ...         print q
5     ...     except ZeroDivisionError:
6     ...         print "Divisão por zero!"
7 ...
8 Divisão por zero!
9 1.0
10 0.5
11 0.333333333333
12 0.25
```

A construção `try:...except:` nos permite verificar a ocorrência de erros em partes de nossos programas e responder adequadamente a ele. o Python reconhece um grande número de tipos de exceções, chamadas “built-in exceptions”. Mas não precisamos sabê-las de cor, basta causar o erro e anotar o seu nome.

Certas situações podem estar sujeitas à ocorrência de mais de um tipo de erro. neste caso, podemos passar uma tupla de exceções para a palavra-chave `except: except (NameError, ValueError, IOError):pass,`

ou simplesmente não passar nada: `except: pass`. Pode acontecer ainda que queiramos lidar de forma diferente com cada tipo de erro (listagem 1.40).

Listagem 1.40: Lidando com diferentes tipos de erros.

```
1 >>> try:
2         f = open( 'arq.txt' )
3         s = f.readline()
4         i = int(s.strip())
5     except IOError, (errno, strerror):
6         print "Erro de I/O (%s): %s" % (
7             errno, strerror)
7     except ValueError:
8         print "Não foi possível converter o
9             dado em Inteiro."
10    except:
11        print "Erro desconhecido."
```

A construção `try:...except:` acomoda ainda uma cláusula `else` opcional, que será executada sempre que o erro esperado não ocorrer, ou seja, caso ocorra um erro imprevisto a cláusula `else` será executada (ao contrário de linhas adicionais dentro da cláusula `try`).

Finalmente, `try` permite uma outra cláusula opcional, `finally`, que é sempre executada (quer haja erros quer não). Ela é útil para tarefas que precisam ser executadas de qualquer forma, como fechar arquivos ou conexões de rede.

1.6 Funções

No Python, uma função é um bloco de código definido por um cabeçalho específico e um conjunto de linhas indentadas, abaixo deste. Funções, uma vez definidas, podem ser chamadas de qualquer ponto do programa (desde que pertençam ao espaço de no-

mes). Na verdade, uma diferença fundamental entre uma função e outros objetos é o fato de ser “chamável”. Isto decorre do fato de todas as funções possuirem um método¹⁰ chamado `__call__`. Todos os objetos que possuam este método poderão ser chamados¹¹.

O ato de chamar um objeto, em Python, é caracterizado pela aposição de parênteses ao nome do objeto. Por exemplo: `func()`. Estes parênteses podem ou não conter “argumentos”. Continue lendo para uma explicação do que são argumentos.

Funções também possuem seu próprio espaço de nomes, ou seja, todas as variáveis definidas no escopo de uma função só existem dentro desta. Funções são definidas pelo seguinte cabeçalho:

```
1 def nome(par1, par2, par3=valor default, *  
         args, **kwargs):
```

A palavra reservada `def` indica a definição de uma função; em seguida deve vir o nome da função que deve seguir as regras de formação de qualquer nome em Python. Entre parênteses vem, opcionalmente, uma lista de argumentos que serão ser passados para a função quando ela for chamada. Argumentos podem ter valores “default” se listados da forma `a=1`. Argumentos com valores default devem vir necessariamente após todos os argumentos sem valores default (Listagem 1.41).

Listagem 1.41: Definindo uma função com um argumento obrigatório e outro opcional (com valor “default”).

```
1 >>> def fun(a,b=1):  
2     ...     print a,b  
3     ...  
4 >>> fun(2)  
5 2 1
```

¹⁰Veja o capítulo 2 para uma explicação do que são métodos.

¹¹O leitor, neste ponto deve estar imaginando todo tipo de coisas interessantes que podem advir de se adicionar um método `__call__` a objetos normalmente não “chamáveis”.

```

6 >>> fun(2,3)
7 2 3
8 fun(b=5,2)
9 SyntaxError: non-keyword arg after keyword
      arg

```

Por fim, um número variável de argumentos adicionais pode ser previsto através de argumentos precedidos por * ou **. No exemplo acima, argumentos passados anonimamente (não associados a um nome) serão colocados em uma tupla de nome t, e argumentos passados de forma nominal (z=2,q='asd') serão adicionados a um dicionário, chamado d(Listagem 1.42).

Listagem 1.42: lista de argumentos variável

```

1 >>> def fun(*t, **d):
2         print t, d
3 >>> fun(1,2,c=2,d=4)
4 (1,2) {'c':3,'d':4}

```

Funções são chamadas conforme ilustrado na linha 3 da listagem 1.42. Argumentos obrigatórios, sem valor “default”, devem ser passados primeiro. Argumentos opcionais podem ser passados fora de ordem, desde que após os argumentos obrigatórios, que serão atribuídos sequencialmente aos primeiros nomes da lista definida no cabeçalho da função(Listagem 1.41).

Muitas vezes é conveniente “desempacotar” os argumentos passados para uma função a partir de uma tupla ou dicionário.

Listagem 1.43: Desempacotando argumentos.

```

1 >>> def fun(a,b,c,d):
2         print a,b,c,d
3 >>> t=(1,2); di = { 'd': 3, 'c': 4}
4 >>> fun(*t,**di)
5 1 2 4 3

```

Argumentos passados dentro de um dicionário podem ser utilizados simultaneamente para argumentos de passagem obrigatória (declarados no cabeçalho da função sem valor “default”) e para argumentos opcionais, declarados ou não(Listagem 1.44).

Listagem 1.44: Passando todos os argumentos por meio de um dicionário.

```
1 >>> def fun2(a, b=1,**outros):
2     ...     print a, b, outros
3     ...
4 >>> dic = { 'a':1, 'b':2, 'c':3, 'd':4}
5 >>> fun2(**dic)
6 1 2 { 'c': 3, 'd': 4}
```

Note que no exemplo 1.44, os valores cujas chaves correspondem a argumentos declarados, são atribuídos a estes e retirados do dicionário, que fica apenas com os ítems restantes.

Funções podem retornar valores por meio da palavra reservada `return`.

Listagem 1.45: Retornando valores a partir de uma função.

```
1 >>> def soma(a,b):
2         return a+b
3         print "ignorado!"
4 >>> soma (3,4)
5 7
```

A palavra `return` indica saída imediata do bloco da função levando consigo o resultado da expressão à sua direita.

Funções lambda

Funções lambda são pequenas funções anônimas que podem ser definidas em apenas uma linha. Por definição, podem conter uma única expressão.

Listagem 1.46: Funções lambda

```

1 >>> def raiz(n):#definindo uma raiz de ordem
2     n
3         return lambda(x):x**(1./n)
4 >>> r4 = raiz(4)#r4 calcula a raiz de ordem
5     4
6 >>> r4(16) #utilizando
7     2

```

Observe no exemplo (1.46), que lambda lembra a definição de variáveis do espaço de nome em que foi criada. Assim, r4 passa a ser uma função que calcula a raiz quarta de um número. Este exemplo nos mostra que podemos modificar o funcionamento de uma função durante a execução do programa: a função raiz retorna uma função raiz de qualquer ordem, dependendo do argumento que receba.

Geradores

Geradores são um tipo especial de função que retém o seu estado de uma chamada para outra. São muito convenientes para criar iteradores, ou seja, objetos que possuem o método `next()`.

Listagem 1.47: Geradores

```

1 >>> def letras(palavra):
2         for i in palavra:
3             yield i
4 >>> for L in letras('gato'): print L
5 g
6 a
7 t
8 o

```

Como vemos na listagem 1.47 um gerador é uma função sobre a qual podemos iterar.

Decoradores

Decoradores são uma alteração da sintaxe do Python, introduzida a partir da versão 2.4, para facilitar a modificação de funções (sem alterá-las), adicionando funcionalidade. Nesta seção vamos ilustrar o uso básico de decoradores. Usos mais avançados podem ser encontrados nesta url: <http://wiki.python.org/moin/PythonDecoratorLibrary>.

Listagem 1.48: Decorador básico.

```
1 def faznada(f):
2     def novaf(*args,**kwargs):
3         print "chamando...",args,kwags
4         return f(*args,**kwags)
5     novaf.__name__ = f.__name__
6     novaf.__doc__ = f.__doc__
7     novaf.__dict__.update(f.__dict__)
8     return novaf
```

Na listagem 1.48, vemos um decorador muito simples. Como seu nome diz, não faz nada, além de ilustrar a mecânica de um decorador. Decoradores esperam um único argumento: uma função. A listagem 1.49, nos mostra como utilizar o decorador.

Listagem 1.49: Utilizando um decorador

```
1 >>>@faznada
2     def soma(a,b):
3         return a+b
4
5 >>> soma(1,2)
6 chamando... (1, 2) {}
7 Out[5]: 3
```

O decorador da listagem 1.48, na verdade adiciona uma linha de código à função que decora: `print "chamando...",args,kwags`.

Repare que o decorador da listagem 1.48, passa alguns atributos básicos da função original para a nova função, de forma que a função decorada possua o mesmo nome, docstring, etc. que a função original. No entanto, esta passagem de atributos “polui” o código da função decoradora. Podemos evitar a “poluição” e o trabalho extra utilizando a funcionalidade do módulo `functools`.

Listagem 1.50: Limpando a geração de decoradores

```
1 >>> from functools import wraps
2 >>> def meuDecorador(f):
3     ...     @wraps(f)
4     ...     def novaf(*args, **kwds):
5         ...         print 'Chamando funcao
decorada'
6         ...         return f(*args, **kwds)
7         ...         return novaf
8
9
9     >>> @meuDecorador
10    ... def exemplo():
11        ...     """Docstring"""
12        ...     print 'funcao exemplo
executada!'
13
14    >>> exemplo()
15    Chamando funcao decorada
16    funcao exemplo executada!
17    >>> exemplo.__name__
18    'exemplo'
19    >>> exemplo.__doc__
20    'Docstring'
```

Decoradores não adicionam nenhuma funcionalidade nova ao que já é possível fazer com funções, mas ajudam a organizar o código e reduzir a necessidade de duplicação. Aplicações científicas

de decoradores são raras, mas a sua presença em pacotes e módulos de utilização genérica vem se tornando cada vez mais comum. Portanto, familiaridade com sua sintaxe é aconselhada.

Strings de Documentação

Strings posicionadas na primeira linha de uma função, ou seja, diretamente abaixo do cabeçalho, são denominadas strings de documentação, ou simplesmente **docstrings**.

Estas strings devem ser utilizadas para documentar a função explicitando sua funcionalidade e seus argumentos. O conteúdo de uma docstring está disponível no atributo `__doc__` da função.

Ferramentas de documentação de programas em Python extraem estas strings para montar uma documentação automática de um programa. A função `help(nome_da_função)` também retorna a docstring. Portanto a inclusão de docstrings auxilia tanto o programador quanto o usuário.

Listagem 1.51: Usando Docstrings

```
1 >>> def soma(a,b):
2         """
3             Esta funcao soma dois numeros:
4             >>> soma(2,3)
5                 5
6             """
7             return a+b
8 >>> help(soma)
9 Help on function soma in module __main__:
10
11 soma(a, b)
12     Esta funcao soma dois numeros:
13     >>> soma(2,3)
14     5
```

No exemplo 1.51, adicionamos uma docstring explicando a finalidade da função soma e ainda incluímos um exemplo. Incluir um exemplo de uso da função cortado e colado diretamente do console Python (incluindo o resultado), nos permitirá utilizar o módulo `doctest` para testar funções, como veremos mais adiante.

1.7 Módulos e Pacotes

Para escrever programas de maior porte ou agregar coleções de funções e/ou objetos criados pelo usuário, o código Python pode ser escrito em um arquivo de texto, salvo com a terminação `.py`, facilitando a re-utilização daquele código. Arquivos com código Python contruídos para serem importados, são denominados “módulo”. Existem algumas variações na forma de se importar módulos. O comando `import meumodulo` cria no espaço de nomes um objeto com o mesmo nome do módulo importado. Funções, classes (ver capítulo 2) e variáveis definidas no módulo são acessíveis como atributos deste objeto. O comando `from modulo import *` importa todas as funções e classes definidas pelo módulo diretamente para o espaço de nomes global¹² do nosso script. Deve ser utilizado com cuidado pois nomes iguais pré-existentes no espaço de nomes global serão redefinidos. Para evitar este risco, podemos substituir o `*` por uma sequência de nomes correspondente aos objetos que desejamos importar: `from modulo import nome1, nome2`. Podemos ainda renomear um objeto ao importá-lo: `import numpy as N` ou ainda `from numpy import det as D`.

Seja um pequeno módulo como o do exemplo 1.52. Podemos importar este módulo em uma sessão do interpretador iniciada no mesmo diretório que contém o módulo (exemplo 1.7).

```
1 >>> import fibo
```

¹²Dicionário de nomes de variáveis e funções válidos durante a execução de um script

Listagem 1.52: Módulo exemplo

```
1 # Disponvel no pacote de programas como:  
2     fibo.py  
3 #modulo para calcular a serie de fibonacci  
4     ate o numero n.  
5  
6 def fib(n):  
7     a, b = 0, 1  
8     while b < n:  
9         print b,  
10        a, b = b, a+b  
11  
12 if __name__=="__main__":  
13     import sys  
14     print __name__  
15     print sys.argv  
16     fib(int(sys.argv[1]))
```

```
2 >>> fibo.fib(50)  
3 1 1 2 3 5 8 13 21 34  
4 >>> fibo.__name__  
5 'fibo'
```

Note que a função declarada em `fibo.py` é chamada como um método de `fibo`. Isto é porque módulos importados são objetos (como tudo o mais em Python).

Quando um módulo é importado ou executado diretamente , torna-se um objeto com um atributo `__name__`. O conteúdo deste atributo depende de como o módulo foi executado. Se foi executado por meio de importação, `__name__` é igual ao nome do módulo (sem a terminação “.py”). Se foi executado diretamente (`python modulo.py`), `__name__` é igual a “`__main__`”.

Durante a importação de um módulo, todo o código contido no mesmo é executado, entretanto como o `__name__` de `fibo` é “`fibo`” e não “`__main__`”, as linhas abaixo do `if` não são executadas. Qual então a função destas linhas de código? Módulos podem ser executados diretamente pelo interpretador, sem serem importados primeiro. Vejamos isso no exemplo 1.53. Podemos ver que agora o `__name__` do módulo é “`__main__`” e, portanto, as linhas de código dentro do bloco `if` são executadas. Note que neste caso importamos o módulo `sys`, cujo atributo `argv` nos retorna uma lista com os argumentos passados para o módulo a partir da posição 1. A posição 0 é sempre o nome do módulo.

Listagem 1.53: Executing a module from the console.

```
1 $ python fibo.py 60
2 __main__
3 ['fibo.py', '60']
4 1 1 2 3 5 8 13 21 34 55
```

Qualquer arquivo com terminação `.py` é considerado um módulo Python pelo interpretador Python. Módulos podem ser executados diretamente ou “importados” por outros módulos.

A linguagem Python tem como uma de suas principais vantagens uma biblioteca bastante ampla de módulos, incluída com a distribuição básica da linguagem. Nesta seção vamos explorar alguns módulos da biblioteca padrão do Python, assim como outros, módulos que podem ser obtidos e adicionados à sua instalação do Python.

Para simplicidade de distribuição e utilização, módulos podem ser agrupados em “pacotes”. Um pacote nada mais é do que um diretório contendo um arquivo denominado `__init__.py` (este arquivo não precisa conter nada). Portanto, pode-se criar um pacote simplesmente criando um diretório chamado, por exemplo, “pacote” contendo os seguintes módulos: `modulo1.py` e `modulo2.py`¹³. Um

¹³ Além de `__init__.py`, naturalmente.

pacote pode conter um número arbitrário de módulos, assim como outros pacotes.

Como tudo o mais em Python, um pacote também é um objeto. Portanto, ao importar o pacote “pacote” em uma sessão Python, modulo1 e modulo2 aparecerão como seus atributos (listagem 1.54).

Listagem 1.54: importing a package

```
1 >>> import pacote
2 >>> dir(pacote)
3 [ 'modulo1' , 'modulo2' ]
```

Pacotes Úteis para Computação Científica

Numpy

Um dos pacotes mais importantes, senão o mais importante para quem deseja utilizar o Python em computação científica, é o `numpy`. Este pacote contém uma grande variedade de módulos voltados para resolução de problemas numéricos de forma eficiente.

Exemplos de objetos e funções pertencentes ao pacote `numpy` aparecerão regularmente na maioria dos exemplos deste livro. Uma lista extensiva de exemplos de Utilização do Numpy pode ser consultada neste endereço: http://www.scipy.org/Numpy_Example_List

Na listagem 1.55, vemos um exemplo de uso típico do `numpy`. O `numpy` nos oferece um objeto matriz, que visa representar o conceito matemático de matriz. Operações matriciais derivadas da álgebra linear, são ainda oferecidas como funções através do subpacote `linalg` (Listagem 1.55).

Listagem 1.55: Calculando e mostrando o determinante de uma matriz.

```
1 >>> from numpy import *
2 >>> a = arange(9)
```

```
3 >>> print a
4 [0 1 2 3 4 5 6 7 8]
5 >>> a.shape =(3,3)
6 >>> print a
7 [[0 1 2]
8 [3 4 5]
9 [6 7 8]]
10 >>> from numpy.linalg import det
11 >>> det(a)
12 0.0
13 >>>
```

Na primeira linha do exemplo 1.55, importamos todas as funções e classes definidas no módulo `numpy`.

Na segunda linha, usamos o comando `arange(9)` para criar um vetor `a` de 9 elementos. Este comando é equivalente ao `range` para criar listas, só que retorna um vetor (matriz unidimensional). Note que este vetor é composto de números inteiros sucessivos começando em zero. Todas as enumerações em Python começam em zero. Como em uma lista, `a[0]` é o primeiro elemento do vetor `a`. O objeto que criamos, é do tipo `array`, definido no módulo `numpy`. Uma outra forma de criar o mesmo objeto seria: `a = array([0,1,2,3,4,5,6,7,8])`.

Na terceira linha, nós mostramos o conteúdo da variável `a` com o comando `print`. Este comando imprime na tela o valor de uma variável.

Como tudo em Python é um objeto, o objeto `array` apresenta diversos métodos e atributos. O atributo chamado `shape` contém o formato da matriz como uma tupla, que pode ser multi-dimensional ou não. Portanto, para converter vetor `a` em uma matriz 3×3 , basta atribuir o valor `(3,3)` a `shape`. Conforme já vimos, atributos e métodos de objetos são referenciados usando-se esta notação de ponto¹⁴.

¹⁴ nome_da_variável.atributo

Na quinta linha, usamos o comando `print` para mostrar a alteração na forma da variável `a`.

Na sexta linha importamos a função `det` do módulo `numpy.linalg` para calcular o determinante da nossa matriz. A função `det(a)` nos informa, então, que o determinante da matriz `a` é 0.0.

Scipy

Outro módulo muito útil para quem faz computação numérica com Python, é o `scipy`. O `scipy` depende do `numpy` e provê uma grande coleção de rotinas numéricas voltadas para aplicações em matemática, engenharia e estatística.

Diversos exemplos da segunda parte deste livro se utilizarão do `scipy`, portanto, não nos extenderemos em exemplos de uso do `scipy`.

Uma lista extensa de exemplos de utilização do `scipy` pode ser encontrada no seguinte endereço:<http://www.scipy.org/Documentation>.

1.8 Documentando Programas

Parte importante de um bom estilo de trabalho em computação científica é a documentação do código produzido. Apesar do Python ser uma linguagem bastante clara e de fácil leitura por humanos, uma boa dose de documentação é sempre positiva.

O Python facilita muito a tarefa tanto do documentador quanto do usuário da documentação de um programa. Naturalmente, o trabalho de documentar o código deve ser feito pelo programador, mas todo o resto é feito pela própria linguagem.

A principal maneira de documentar programas em Python é através da adição de strings de documentação (“docstrings”) a funções e classes ao redigir o código. Módulos também podem possuir “docstrings” contendo uma sinopse da sua funcionalidade. Estas strings servem não somente como referência para o próprio programador durante o desenvolvimento, como também como material

para ferramentas de documentação automática. A principal ferramenta de documentação disponível para desenvolvedores é o `pydoc`, que vem junto com a distribuição da linguagem.

Pydoc

O `pydoc` é uma ferramenta que extrai e formata a documentação de programas Python. Ela pode ser utilizada de dentro do console do interpretador Python, ou diretamente do console do Linux.

```
1 $ pydoc pydoc
```

No exemplo acima, utilizamos o `pydoc` para examinar a documentação do próprio módulo `pydoc`. Podemos fazer o mesmo para acessar qualquer módulo disponível no `PYTHONPATH`.

O `pydoc` possui algumas opções de comando muito úteis:

- k <**palavra**> Procura por palavras na documentação de todos os módulos.
- p <**porta**> <**nome**> Gera a documentação em html iniciando um servidor HTTP na porta especificada da máquina local.
- g Útil para sistemas sem fácil acesso ao console, inicia um servidor HTTP e abre uma pequena janela para busca.
- w <**nome**> escreve a documentação requisitada em formato HTML, no arquivo <**nome**>.html, onde <**nome**> pode ser um módulo instalado na biblioteca local do Python ou um módulo ou pacote em outra parte do sistema de arquivos. Muito útil para gerar documentação para programas que criamos.

Além do `pydoc`, outras ferramentas mais sofisticadas, desenvolvidas por terceiros, estão disponíveis para automatizar a documentação de programas Python. Exploraremos uma alternativa a seguir.

Epydoc

O Epydoc é uma ferramenta consideravelmente mais sofisticada do que o módulos `pydoc`. Além de prover a funcionalidade já demonstrada para o `pydoc`, oferece outras facilidades como a geração da documentação em formato PDF ou HTML e suporte à formatação das “*docstrings*”.

O uso do Epydoc é similar ao do `pydoc`. Entretanto, devido à sua maior versatilidade, suas opções são bem mais numerosas (1.56).

Listagem 1.56: Listando as opções do Epydoc.

```
1 $ epydoc -h
```

Não vamos discutir em detalhes as várias opções do Epydoc pois estas encontram-se bem descritas na página `man` do programa. Ainda assim, vamos comentar algumas funcionalidades interessantes.

A capacidade de gerar a documentação em L^AT_EX, facilita a customização da mesma pelo usuário e a exportação para outros formatos. A opção `-url`, nos permite adicionar um link para o website de nosso projeto ao cabeçalho da documentação. O Epydoc também verifica o quão bem nosso programa ou pacote encontra-se documentado. Usando-se a opção `-check` somos avisados sobre todos os objetos não documentados.

A partir da versão 3.0, o Epydoc adiciona links para o código fonte na íntegra, de cada elemento de nosso módulo ou pacote. A opção `-graph` pode gerar três tipos de gráficos sobre nosso programa, incluindo um diagrama UML(Figura 1.1).

Dada toda esta funcionalidade, vale apena conferir o Epydoc¹⁵.

¹⁵<http://epydoc.sourceforge.net>

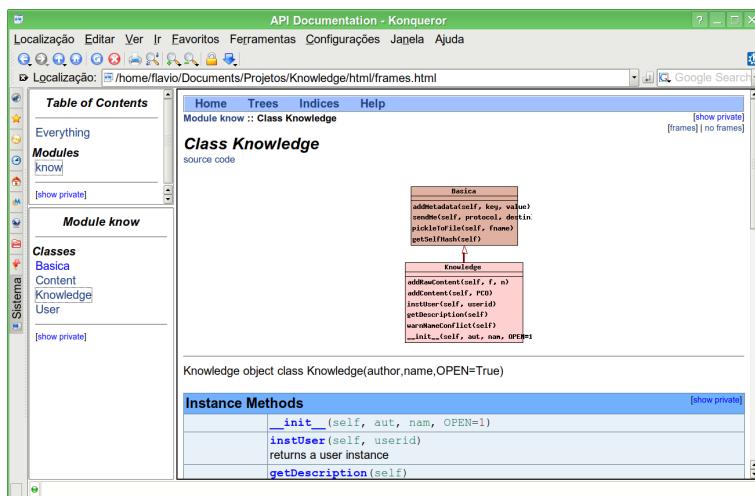


Figura 1.1: Versão HTML da documentação gerada pelo Epydoc.

1.9 Exercícios

1. Repita a iteração do exemplo 1.35 sem utilizar a função `enumerate`. Execute a iteração do objeto gerado por `enumerate` manualmente, sem o auxílio do laço `for` e observe o seu resultado.
2. Adicione a funcionalidade `else` à listagem 1.33 utilizando exceções.
3. Escreva um exemplo de iteração empregando `break`, `continue` e `else`(ao final).

Capítulo 2

Programação Orientada a Objetos

Introdução à programação orientada a objetos e sua implementação na linguagem Python. Pré-requisitos: Ter lido o capítulo 1.

PROGRAMAÇÃO orientada a objetos é um tema vasto na literatura computacional. Neste capítulo introduziremos os recursos presentes na linguagem Python para criar objetos e, através de exemplos, nos familiarizaremos com o paradigma da programação orientada a objetos.

Historicamente, a elaboração de programas de computador passou por diversos paradigmas. Programas de computador começaram como uma simples lista de instruções a serem executadas, em sequência, pela CPU. Este paradigma de programação foi mais tarde denominado de programação não-estruturada. Sua principal característica é a presença de comandos para desviar a execução para pontos específicos do programa (goto, jump, etc.) Exemplos de linguagens não-estruturadas são Basic, Assembly e Fortran. Mais tarde surgiram as linguagens estruturadas, que permitiam a organização do programa em blocos que podiam ser executados em qualquer ordem. As Linguagens C e Pascal ganham grande popu-

laridade, e linguagens até então não estruturadas (Basic, Fortran, etc.) ganham versões estruturadas. Outros exemplos de linguagens não estruturadas incluem Ada, D, Forth,PL/1, Perl, maple, Matlab, Mathematica, etc.

A estruturação de programas deu origem a diversos paradigmas de programação, tais como a programação funcional, na qual a computação é vista como a avaliação sequencial de funções matemáticas, e cujos principais exemplos atuais são as linguagens Lisp e Haskell.

A programação estruturada atingiu seu pico em versatilidade e popularidade com o paradigma da programação orientada a objetos. Na programação orientada a objetos, o programa é dividido em unidades (objetos) contendo dados (estado) e funcionalidade (métodos) própria. Objetos são capazes de receber mensagens, processar dados (de acordo com seus métodos) e enviar mensagens a outros objetos. Cada objeto pode ser visto como uma máquina independente ou um ator que desempenha um papel específico.

2.1 Objetos

Um tema frequente em computação científica, é a simulação de sistemas naturais de vários tipos, físicos, químicos, biológicos, etc. A orientação a objetos é uma ferramenta natural na construção de simulações, pois nos permite replicar a arquitetura do sistema natural em nossos programas, representando componentes de sistemas naturais como objetos computacionais.

A orientação a objeto pode ser compreendida em analogia ao conceito grammatical de objeto. Os componentes principais de uma frase são: sujeito, verbo e objeto. Na programação orientada a objeto, a ação está sempre associada ao objeto e não ao sujeito, como em outros paradigmas de programação.

Um dos pontos altos da linguagem Python que facilita sua assimilação por cientistas com experiência prévia em outras linguagens

Listagem 2.1: Definindo uma classe

```
1 class Objeto:  
2     pass
```

de programação, é que a linguagem não impõe nenhum estilo de programação ao usuário. Em Python pode-se programar de forma não estruturada, estruturada, procedural, funcional ou orientada a objeto. Além de acomodar as preferências de cada usuário, permite acomodar as conveniências do problema a ser resolvido pelo programa.

Neste capítulo, introduziremos as técnicas básicas de programação orientada a objetos em Python. Em exemplos de outros capítulos, outros estilos de programação aparecerão, justificados pelo tipo de aplicação a que se propõe.

Definindo Objetos e seus Atributos em Python

Ao se construir um modelo de um sistema natural, uma das características desejáveis deste modelo, é um certo grau de generalidade. Por exemplo, ao construir um modelo computacional de um automóvel, desejamos que ele (o modelo) represente uma categoria de automóveis e não apenas nosso automóvel particular. Ao mesmo tempo, queremos ser capazes de ajustar este modelo para que ele possa representar nosso automóvel ou o de nosso vizinho sem precisar re-escrever o modelo inteiramente do zero. A estes modelos de objetos dá-se o nome de classes.

A definição de classes em Python pode ser feita de forma mais ou menos genérica. À partir das classes, podemos construir instâncias ajustadas para representar exemplares específicos de objetos representados pela classe. Na listagem 2.1, temos uma definição mínima de uma classe de objetos. Criamos uma classe chamada

Listagem 2.2: Definindo atributos de uma classe

```
1 class pessoa:
2     idade=20
3     altura=170
4     sexo='masculino'
5     peso=70
```

Objeto, inteiramente em branco. Como uma classe completamente vazia não é possível em Python, adicionamos o comando `pass` que não tem qualquer efeito.

Para criar uma classe mais útil de objetos, precisamos definir alguns de seus **atributos**. Como exemplo vamos criar uma classe que represente pessoas. Na listagem 2.2, definimos alguns atributos para a classe `pessoa`. Agora, podemos criar instâncias do objeto `pessoa` e estas instâncias herdarão estes atributos.

```
1 >>> maria = pessoa()
2 >>> maria.peso
3 70
4 >>> maria.sexo
5 'masculino'
6 >>> maria
7 <__main__.pessoa instance at 0x402f196c>
```

Entretanto, os atributos definidos para o objeto `pessoa` (listagem 2.2), são atributos que não se espera que permaneçam os mesmos para todas as possíveis instâncias (`pessoas`). O mais comum é que os atributos específicos das instâncias sejam fornecidos no momento da sua criação. Para isso, podemos definir quais as informações necessárias para criar uma instância do objeto `pessoa`.

Listagem 2.3: Passando atributos na criação de um objeto

```
1 >>> class pessoa:
```

```
2 ...     def __init__(self, idade, altura, sexo,
3 ...                         peso):
4 ...         self.idade=idade
5 ...         self.altura=altura
6 ...         self.sexo=sexo
7 ...         self.peso=70
7 >>> maria = pessoa()
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in ?
10 TypeError: __init__() takes exactly 5
11           arguments (1 given)
11 maria=pessoa(35,155,'feminino',50)
12 >>> maria.sexo
13 'feminino'
```

A função `__init__` que definimos na nova versão da classe `pessoa` (listagem 2.3), é uma função padrão de classes, que é executada automaticamente, sempre que uma nova instância é criada. Assim, se não passarmos as informações requeridas como argumentos pela função `__init__` (listagem 2.3, linha 7), recebemos uma mensagem de erro. Na linha 11 da listagem 2.3 vemos como instanciar a nova versão da classe `pessoa`.

Adicionando Funcionalidade a Objetos

Continuando com a analogia com objetos reais, os objetos computacionais também podem possuir funcionalidades, além de atributos. Estas funcionalidades são denominadas **métodos** de objeto. Métodos são definidos como funções pertencentes ao objeto. A função `__init__` que vimos há pouco é um método presente em todos os objetos, ainda que não seja definida pelo programador. Métodos são sempre definidos com, pelo menos, um argumento: `self`, que pode ser omitido ao se invocar o método em uma instância do objeto (veja linha 11 da listagem 2.3). O argumento `self` também

deve ser o primeiro argumento a ser declarado na lista de argumentos de um método.

Herança

Para simplificar a definição de classes complexas, classes podem herdar atributos e métodos de outras classes. Por exemplo, uma classe Felino, poderia herdar de uma classe mamífero, que por sua vez herdaria de outra classe, vertebrados. Esta cadeia de herança pode ser extendida, conforme necessário (Listagem 2.4).

Listagem 2.4: Herança

```

1 >>> class Vertebrado:
2         vertebra = True
3 >>> class Mamifero(Vertebrado):
4         mamas = True
5 >>> class Carnivoro(Mamifero):
6         longos_caninos = True
7 >>> bicho = Carnivoro()
8 >>> dir(bicho)
9 [ '__doc__', '__module__', 'longos_caninos',
  'mamas', 'vertebra']
10 >>> issubclass(Carnivoro, Vertebrado)
11 True
12 >>> bicho.__class__
13 <class '__main__.Carnivoro' at 0xb7a1d17c>
14 >>> isinstance(bicho, Mamifero)
15 True

```

Na listagem 2.4, vemos um exemplo de criação de um objeto, instância da classe Carnivoro, herdando os atributos dos ancestrais desta. Vemos também que é possível testar a pertinência de um objeto a uma dada classe, através da função `isinstance`. A função `issubclass`, de forma análoga, nos permite verificar as relações parentais de duas classes.

Utilizando Classes como Estruturas de Dados Genéricas.

Devido à natureza dinâmica do Python, podemos utilizar uma classe como um compartimento para quaisquer tipos de dados. Tal construto seria equivalente ao tipo `struct` da linguagem C. Para exemplificar, vamos definir uma classe vazia:

Listagem 2.5: Classe como uma estrutura de dados

```
1 >>> class Cachorro:  
2         pass  
3 >>> rex=Cachorro()  
4 >>> rex.dono = 'Pedro'  
5 >>> rex.raca = 'Pastor'  
6 >>> rex.peso=25  
7 >>> rex.dono  
8 'Pedro'  
9 >>> laika = Cachorro()  
10 >>> laika.dono  
11 AttributeError: Cachorro instance has no  
attribute 'dono'
```

No exemplo 2.5, a classe `Cachorro` é criada vazia, mas ainda assim, atributos podem ser atribuídos a suas instâncias, sem alterar a estrutura da classe.

2.2 Exercícios

1. Utilizando os conceitos de herança e os exemplos de classes apresentados, construa uma classe `Cachorro` que herde atributos das classes `Carnívoro` e `Mamífero` e crie instâncias que possuam donos, raças, etc.
2. No Python, o que define um objeto como “chamável” (funções, por exemplo) é a presença do método `__call__`. Crie uma

classe, cujas instâncias podem ser “chamadas”, por possuírem o método `__call__`.

Capítulo 3

Criando Gráficos em Python

Introdução à produção de figuras de alta qualidade utilizando o pacote matplotlib. Pré-requisitos: Capítulo 1.

EXISTE um número crescente de módulos para a criação de gráficos científicos com Python. Entretanto, até o momento da publicação deste livro, nenhum deles fazia parte da distribuição oficial do Python.

Para manter este livro prático e conciso, foi necessário escolher apenas um dos módulos de gráficos disponíveis, para apresentação neste capítulo.

O critério de escolha levou em consideração os principais valores da filosofia da linguagem Python (ver listagem): simplicidade, elegância, versatilidade, etc. À época, a aplicação destes critérios nos deixou apenas uma opção: o módulo matplotlib¹.

3.1 Introdução ao Matplotlib

O módulo matplotlib (MPL) é voltado para a geração de gráficos bi-dimensionais de vários tipos, e se presta para utilização tanto in-

¹<http://matplotlib.sourceforge.net>

terativa quanto em scripts, aplicações web ou integrada a interfaces gráficas (GUIs) de vários tipos.

A instalação do MPL também segue o padrão de simplicidade do Python (listagem 3.1). Basta baixar o pacote **tar.gz** do sítio, descompactar e executar o comando de instalação.

Listagem 3.1: Instalando o matplotlib

```
1 $ python setup.py install
```

O MPL procura tornar simples tarefas de plotagem, simples e tarefas complexas, possíveis (listagem 3.2, figura 3.1). Os gráficos gerados podem ser salvos em diversos formatos: jpg, png, ps, eps e svg. Ou seja, o MPL exporta em formatos raster e vetoriais (svg) o que torna sua saída adequada para inserção em diversos tipos de documentos.

Listagem 3.2: Criando um histograma no modo interativo

```
1 >>> from pylab import *
2 >>> from numpy.random import *
3 >>> x=normal(0,1,1000)
4 >>> hist(x,30)
5 ...
6 >>> show()
```

Podemos também embutir a saída gráfica do MPL em diversas GUIs: GTK, WX e TKinter. Naturalmente a utilização do MPL dentro de uma GUI requer que os módulos adequados para o desenvolvimento com a GUI em questão estejam instalados².

Para gerar os gráficos e se integrar a interfaces gráficas, o MPL se utiliza de diferentes “backends” de acordo com nossa escolha (Wx, GTK, Tk, etc.).

²Veja no sitio do MPL os pré-requisitos para cada uma das GUIs

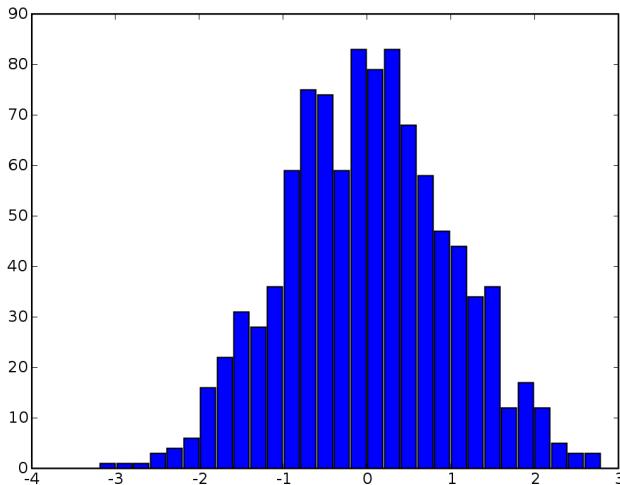


Figura 3.1: Histograma simples a partir da listagem 3.2

Configurando o MPL

O MPL possui valores *default* para propriedades genéricas dos gráficos gerados. Estas configurações ficam em um arquivo texto chamado **matplotlibrc**, que deve ser copiado da distribuição do MPL, editado conforme as preferências do usuário e renomeado para `~/.matplotlibrc`, ou seja, deve ser colocado como um arquivo oculto no diretório **home** do usuário.

A utilização de configurações padrão a partir do **matplotlibrc** é mais útil na utilização interativa do MPL, pois evita a necessidade de configurar cada figura de acordo com as nossas preferências, a

<code>bar</code>	Gráfico de barras
<code>cohere</code>	Gráfico da função de coerência
<code>csd</code>	Densidade espectral cruzada
<code>errorbar</code>	Gráfico com barras de erro
<code>hist</code>	Histograma
<code>imshow</code>	Plota imagens
<code>pcolor</code>	Gráfico de pseudocores
<code>plot</code>	Gráfico de linha
<code>psd</code>	Densidade espectral de potência
<code>scatter</code>	Diagrama de espalhamento
<code>specgram</code>	Espectrograma
<code>stem</code>	Pontos com linhas verticais

Tabela 3.1: Principais comandos de plotagem do MPL

cada vez que usamos o MPL³.

Comandos Básicos

Os comandos relacionados diretamente à geração de gráficos são bastante numerosos (tabela 3.1); mas, além destes, existe um outro conjunto ainda maior de comandos, voltados para o ajuste fino de detalhes dos gráficos (ver tabela 3.2, para uma amostra), tais como tipos de linha, símbolos, cores, etc. Uma explicação mais detalhada dos comandos apresentados na tabela 3.1, será dada nas próximas seções no contexto de exemplos.

³Para uma descrição completa das características de gráficos que podem ser configuradas, veja o exemplo de `matplotlibrc` que é fornecido com a distribuição do MPL.

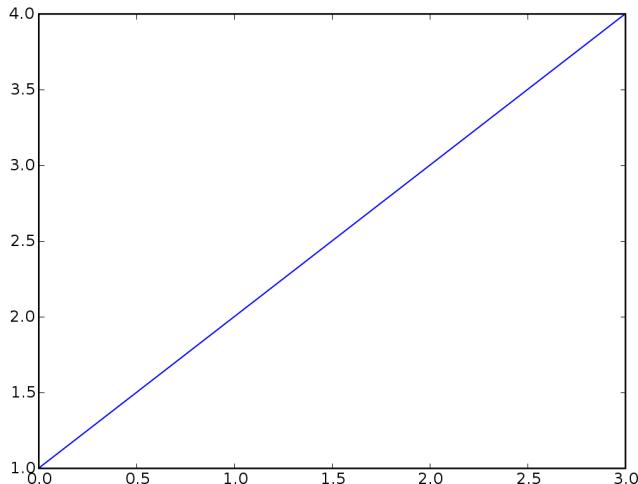


Figura 3.2: Reta simples a partir da listagem 3.3

3.2 Exemplos Simples

O comando `plot`

O comando `plot` é um comando muito versátil, pode receber um número variável de argumentos, com diferentes saídas.

Listagem 3.3: Gráfico de linha

```
1 from pylab import *
2 plot ([1 ,2 ,3 ,4])
3 show()
```

Quando `plot` recebe apenas uma sequência de números (lista, tupla ou array), ele gera um gráfico (listagem 3.3) utilizando os valores recebidos como valores de `y` enquanto que os valores de `x` são as posições destes valores na sequência.

Caso duas sequências de valores sejam passadas para `plot` (listagem 3.4), a primeira é atribuída a `x` e a segunda a `y`. Note que, neste exemplo, ilustra-se também a especificação do tipo de saída gráfica como uma sequência de pontos. O parâmetro '`ro`' indica que o símbolo a ser usado é um círculo vermelho.

Listagem 3.4: Gráfico de pontos com valores de `x` e `y` especificados.

```
1 from pylab import *
2 plot([1,2,3,4], [1,4,9,16], 'ro')
3 axis([0, 6, 0, 20])
4 savefig('ponto.png')
5 show()
```

Na linha 3 da listagem 3.4 especifica-se também os limites dos eixos como uma lista de quatro elementos: os valores mínimo e máximo dos eixos `x` e `y`, respectivamente. Na linha 4, vemos o comando `savefig` que nos permite salvar a figura gerada no arquivo cujo nome é dado pela string recebida. O tipo de arquivo é determinado pela extensão (.png, .ps, .eps, .svg, etc).

O MPL nos permite controlar as propriedades da linha que forma o gráfico. Existe mais de uma maneira de determinar as propriedades das linhas geradas pelo comando `plot`. Uma das maneiras mais diretas é através dos argumentos listados na tabela 3.2. Nos diversos exemplos apresentados neste capítulo, alguns outros métodos serão apresentados e explicados⁴.

Vale a pena ressaltar que o comando `plot` aceita, tanto listas, quanto arrays dos módulos Numpy, Numeric ou numarray. Na ver-

⁴Para maiores detalhes consulte a documentação do MPL (<http://matplotlib.sourceforge.net>).

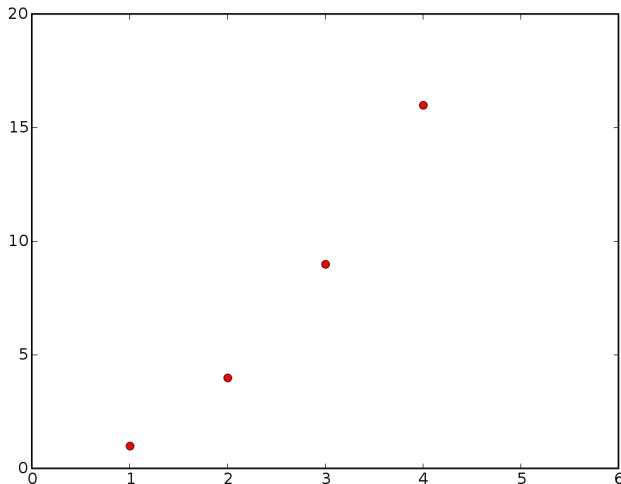


Figura 3.3: Gráfico com símbolos circulares a partir da listagem 3.4

dade todas a sequências de números passadas para o comando `plot` são convertidas internamente para `arrays`.

O Comando `subplot`

O MPL trabalha com o conceito de *figura* independente do de *eixos*. O comando `gcf()` retorna a figura atual, e o comando `gca()` retorna os eixos atuais. Este detalhe nos permite posicionar os eixos de um gráfico em posições arbitrárias dentro da figura. Todos os comandos de plotagem são realizados nos eixos atuais. Mas, para a maioria dos usuários, estes detalhes são transparentes, ou

Tabela 3.2: Argumentos que podem ser passados juntamente com a função plot para controlar propriedades de linhas.

Propriedade	Valores
alpha	transparência (0-1)
antialiased	true false
color	Cor: b,g,r,c,m,y,k,w
label	legenda
linestyle	-- : - . -
linewidth	Espessura da linha (pontos)
marker	+ o . s v x > < ^
markeredgewidth	Espessura da margem do símbolo
markeredgecolor	Cor da margem do símbolo
markerfacecolor	Cor do símbolo
markersize	Tamanho do símbolo (pontos)

seja, o usuário não precisa tomar conhecimento deles. A listagem 3.5 apresenta uma figura com dois eixos feita de maneira bastante simples.

Listagem 3.5: Figura com dois gráficos utilizando o comando subplot.

```

1 # Disponível no pacote de programas como:
2   subplot.py
3 from pylab import *
4
5 def f(t):
6     s1 = cos(2*pi*t)
7     e1 = exp(-t)
8     return multiply(s1, e1)
9
10 t1 = arange(0.0, 5.0, 0.1)
11 t2 = arange(0.0, 5.0, 0.02)

```

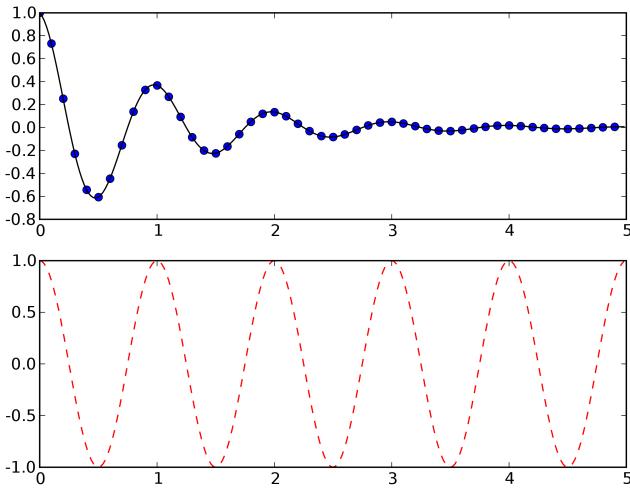


Figura 3.4: Figura com dois gráficos utilizando o comando subplot, a partir da listagem 3.5

```
11||figure(1)
12||subplot(211)
13||plot(t1, f(t1), 'bo', t2, f(t2), 'k')
14|
15|
16||subplot(212)
17||plot(t2, cos(2*pi*t2), 'r—')
18||savefig('subplot.png', dpi=400)
19||show()
```

Tabela 3.3: Argumentos opcionais dos comandos de inserção de texto.

Propriedades	Valores
alpha	Transparência (0-1)
color	Cor
fontangle	italic normal oblique
fontname	Nome da fonte
fontsize	Tamanho da fonte
fontweight	normal bold light4
horizontalalignment	left center right
rotation	horizontal vertical
verticalalignment	bottom center top

O comando `figure(1)`, na linha 11 da listagem 3.5, é opcional, mas pode vir a ser importante quando se deseja criar múltiplas figuras, antes de dar o comando `show()`. Note pelo primeiro comando `plot` da listagem 3.5, que o comando `plot` aceita mais de um par `(x,y)`, cada qual com seu tipo de linha especificado independentemente.

Adicionando Texto a Gráficos

O MPL nos oferece quatro comandos para a adição de texto a figuras: `title`, `xlabel`, `ylabel`, e `text`. O três primeiros adicionam título e nomes para os eixos `x` e `y`, respectivamente.

Todos os comandos de inserção de texto aceitam argumentos (tabela 3.3) adicionais para formatação do texto. O MPL também nos permite utilizar um subconjunto da linguagem `TeX` para formatar expressões matemáticas (Listagem 3.6 e figura 3.5). Para inserir expressões em `TeX`, é necessário que as strings contendo as expressões matemáticas sejam “raw strings”⁵, e delimitadas por

⁵ exemplo: `r'raw string'`

cifrões(\$).

Listagem 3.6: Formatando texto e expressões matemáticas

```
1 # Disponivel no pacote de programas como:  
2     mathtext.py  
3 from pylab import *  
4 t = arange(0.0, 2.0, 0.01)  
5 s = sin(2*pi*t)  
6 plot(t,s)  
7 title(r'$\alpha_i > \beta_i$', fontsize=20)  
8 text(1, -0.6, r'$\sum_{i=0}^{\infty} x_i$',  
9       fontsize=20)  
10 text(0.6, 0.6, r'$\mathcal{A}\sin(2\omega t)$',  
11        fontsize=20)  
12 xlabel('time (s)')  
13 ylabel(r'$\mathrm{Voltagem}(\mu V)$')  
14 savefig('mathtext.png', dpi=400)  
15 show()
```

3.3 Exemplos Avançados

O MPL é capaz produzir uma grande variedade gráficos mais sofisticados do que os apresentados até agora. Explorar todas as possibilidades do MPL, foge ao escopo deste texto, mas diversos exemplos de outros tipos de gráficos serão apresentados junto com os exemplos da segunda parte deste livro.

Mapas

O matplotlib pode ser extendido para plotar mapas. Para isso precisamos instalar o Basemap toolkit. Se você já instalou o matplot-

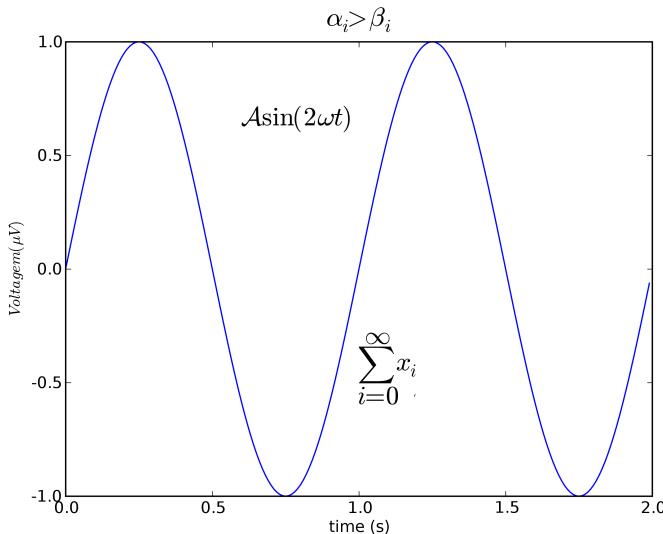


Figura 3.5: Formatação de texto em figuras. Gerada pela listagem 3.6

tlib, basta baixar o arquivo tar.gz do Basemap, descompactar para um diretório e executar o já conhecido `python setup.py install`.

O Basemap já vem com um mapa mundi incluído para demonstração. Vamos utilizar este mapa em nosso exemplo (Listagem 3.7).

Listagem 3.7: Plotando o globo terrestre

```

1 # Disponível no pacote de programas como:
   mapa.py
2 from matplotlib.toolkits.basemap import
   Basemap

```

```
3 import pylab as p
4
5 map = Basemap(projection='robin',lat_0=-23,
6                 lon_0=-46,
7                 resolution='l',area_thresh
8                 =1000.)
9 map.drawcoastlines()
10 map.drawcountries()
11 map.fillcontinents(color='coral')
12 map.drawmapboundary()
13 map.drawmeridians(p.arange(0,360,30))
14 map.drawparallels(p.arange(-90,90,30))
15 p.show()
```

Na listagem 3.6, criamos um objeto `map`, que é uma instância da classe `Basemap` (linha 4). A classe `Basemap` possui diversos atributos, mas neste exemplo estamos definindo apenas alguns como a projeção (Robinson), coordenadas do centro do mapa, `lat_0` e `lon_0`, resolução dos contornos, ajustada para baixa, e tamanho mínimo de detalhes a serem desenhados, `area_thresh`, definido como $1000km^2$.

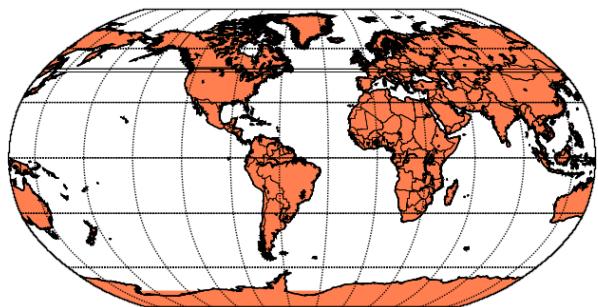


Figura 3.6: Mapa mundi na projeção de Robinson.

Capítulo 4

Ferramentas de Desenvolvimento

Exposição de ferramentas voltadas para o aumento da produtividade em um ambiente de trabalho em computação científica. Pré-requisitos: Capítulos 1 e 2

Como em todo ambiente de trabalho, para aumentar a nossa produtividade em Computação Científica, existem várias ferramentas além da linguagem de programação. Neste capítulo falaremos das ferramentas mais importantes, na opinião do autor.

4.1 Ipython

A utilização interativa do Python é de extrema valia. Outros ambientes voltados para computação científica, tais como MatlabTM, R, MathematicaTM dentre outros, usam o modo interativo como seu principal modo de operação. Os que desejam fazer o mesmo com o Python, podem se beneficiar imensamente do Ipython.

O Ipython é uma versão muito sofisticada da shell do Python voltada para tornar mais eficiente a utilização interativa da linguagem Python.

Primeiros Passos

Para iniciar o Ipython, digitamos o seguinte comando:

```
1 $ ipython [opções] arquivos
```

Muitas das opções que controlam o funcionamento do Ipython não são passadas na linha de comando, estão especificadas no arquivo `ipythonrc` dentro do diretório `./ipython`.

Quatro opções do Ipython são consideradas especiais e devem aparecer em primeiro lugar, antes de qualquer outra opção: `-gthread`, `-qthread`, `-wthread`, `-pylab`. As três primeiras opções são voltadas para o uso interativo de módulos na construção de GUIs (interfaces gráficas), respectivamente GTK, Qt, WxPython. Estas opções iniciam o Ipython em um “thread” separado, de forma a permitir o controle interativo de elementos gráficos. A opção `-pylab` permite o uso interativo do pacote matplotlib (Ver capítulo 3). Esta opção executará `from pylab import *` ao iniciar, e permite que gráficos sejam exibidos sem necessidade de invocar o comando `show()`, mas executará scripts que contém `show()` ao final, corretamente.

Após uma das quatro opções acima terem sido especificadas, as opções regulares podem seguir em qualquer ordem. Todas as opções podem ser abreviadas à forma mais curta não-ambígua, mas devem respeitar maiúsculas e minúsculas (como nas linguagens Python e Bash, por sinal). Um ou dois hífens podem ser utilizados na especificação de opções.

Todas as opções podem ser prefixadas por “no” para serem desligadas (no caso de serem ativas por default).

Devido ao grande número de opções existentes, não iremos listá-las aqui. consulte a documentação do Ipython para aprender sobre elas. Entretanto, algumas opções poderão aparecer ao longo desta seção e serão explicadas à medida em que surgirem.

Comandos Mágicos

Uma das características mais úteis do Ipython é o conceito de comandos mágicos. No console do Ipython, qualquer linha começada pelo caractere %, é considerada uma chamada a um comando mágico. Por exemplo, `%autoindent` liga a indentação automática dentro do Ipython.

Existe uma opção que vem ativada por default no `ipythonrc`, denominada `automagic`. Com esta função, os comandos mágicos podem ser chamados sem o %, ou seja `autoindent` é entendido como `%autoindent`. Variáveis definidas pelo usuário podem mascarar comandos mágicos. Portanto, se eu definir uma variável `autoindent = 1`, a palavra `autoindent` não é mais reconhecida como um comando mágico e sim como o nome da variável criada por mim. Porém, ainda posso chamar o comando mágico colocando o caractere % no início.

O usuário pode extender o conjunto de comandos mágicos com suas próprias criações. Veja a documentação do Ipython sobre como fazer isso.

O comando mágico `%magic` retorna um explicação dos comandos mágicos existentes.

`%Exit` Sai do console Ipython.

`%Pprint` Liga/desliga formatação do texto.

`%Quit` Sai do Ipython sem pedir confirmação.

`%alias` Define um sinônimo para um comando.

Você pode usar `%1` para representar a linha em que o comando `alias` foi chamado, por exemplo:

1 In [2]: alias all echo "Entrada entre parênteses: (%1)"

2 In [3]: all Ola mundo

3 Entrada entre parênteses: (Ola mundo)

76 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

%autocall Liga/desliga modo que permite chamar funções sem os parênteses. Por exemplo: `fun 1` vira `fun(1)`.

%autoindent Liga/desliga auto-indentação.

%automagic Liga/desliga auto-mágica.

%bg Executa um comando em segundo plano, em um thread separado. Por exemplo: `%bg func(x,y,z=1)`. Assim que a execução se inicia, uma mensagem é impressa no console informando o número da tarefa. Assim, pode-se ter acesso ao resultado da tarefa número 5 por meio do comando `jobs.results[5]`

O Ipython possui um gerenciador de tarefas acessível através do objeto `jobs`. Para maiores informações sobre este objeto digite `jobs?`. O Ipython permite completar automaticamente um comando digitado parcialmente. Para ver todos os métodos do objeto `jobs` experimente digitar `jobs.`seguido da tecla <TAB>.

%bookmark Gerencia o sistema de marcadores do Ipython. Para saber mais sobre marcadores digite `%bookmark?`.

%cd Muda de diretório.

%colors Troca o esquema de cores.

%cpaste Cola e executa um bloco pré-formatado da área de transferência (clipboard). O bloco tem que ser terminado por uma linha contendo `--`.

%dhist Imprime o histórico de diretórios.

%ed Sinônimo para `%edit`

%edit Abre um editor e executa o código editado ao sair. Este comando aceita diversas opções, veja a documentação.

O editor a ser aberto pelo comando `%edit` é o que estiver definido na variável de ambiente `$EDITOR`. Se esta variável não estiver definida, o Ipython abrirá o `vi`. Se não for especificado o nome de um arquivo, o Ipython abrirá um arquivo temporário para a edição.

O comando `%edit` apresenta algumas conveniências. Por exemplo: se definirmos uma função `fun` em uma sessão de edição ao sair e executar o código, esta função permanecerá definida no espaço de nomes corrente. Então podemos digitar apenas `%edit fun` e o Ipython abrirá o arquivo que a contém, posicionando o cursor, automaticamente, na linha que a define. Ao sair desta sessão de edição, a função editada será atualizada.

```
1 In [6]:%ed
2 IPython will make a temporary file named: /
   tmp/ipython_edit_GuUWr_.py
3 done. Executing edited code...
4 Out[6]:"def fun():\n    print 'fun'\n\n    def funa():\n        print 'funa'\n"
5
6 In [7]:fun()
7 fun
8
9 In [8]:funa()
10 funa
11
12 In [9]:%ed fun
13 done. Executing edited code... 
```

`%hist` Sinônimo para `%history`.

`%history` Imprime o histórico de comandos. Comandos anteriores também podem ser acessados através da variável `_i<n>`, que é o n -ésimo comando do histórico.

78 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

```
1 In [1]:% hist
2 1: _ip.magic("%hist ")
3
4 In [2]:% hist
5 1: _ip.magic("%hist ")
6 2: _ip.magic("%hist ")
```

O Ipython possui um sofisticado sistema de registro das sessões. Este sistema é controlado pelos seguintes comandos mágicos: `%logon`, `%logoff`, `%logstart` e `%logstate`. Para maiores informações consulte a documentação.

`%lsmagic` Lista os comandos mágicos disponíveis.

`%macro` Define um conjunto de linhas de comando como uma macro para uso posterior: `%macro teste 1 2` ou `%macro macro2 44-47 49.`

`%p` Sinônimo para `print`.

`%pdb` liga/desliga depurador interativo.

`%pdef` Imprime o cabeçalho de qualquer objeto chamável. Se o objeto for uma classe, retorna informação sobre o construtor da classe.

`%pdoc` Imprime a docstring de um objeto.

`%pfile` Imprime o arquivo onde o objeto encontra-se definido.

`%psearch` Busca por objetos em espaços de nomes.

`%psource` Imprime o código fonte de um objeto. O objeto tem que ter sido importado a partir de um arquivo.

`%quickref` Mostra um guia de referência rápida

`%quit` Sai do Ipython.

%r Repete o comando anterior.

%rehash Atualiza a tabela de sinônimos com todas as entradas em \$PATH. Este comando não verifica permissões de execução e se as entradas são mesmo arquivos. **%rehashx** faz isso, mas é mais lento.

%rehashdir Adiciona os executáveis dos diretórios especificados à tabela de sinônimos.

%rehashx Atualiza a tabela de sinônimos com todos os arquivos executáveis em \$PATH.

%reset Re-inicializa o espaço de nomes removendo todos os nomes definidos pelo usuário.

%run Executa o arquivo especificado dentro do Ipython como um programa.

%runlog Executa arquivos como logs.

%save Salva um conjunto de linhas em um arquivo.

%sx Executa um comando no console do Linux e captura sua saída.

%store Armazena variáveis para que estejam disponíveis em uma sessão futura.

%time Cronometra a execução de um comando ou expressão.

%timeit Cronometra a execução de um comando ou expressão utilizando o módulo `timeit`.

%unalias Remove um sinônimo.

%upgrade Atualiza a instalação do Ipython.

%who Imprime todas as variáveis interativas com um mínimo de formatação.

`%who_ls` Retorna uma lista de todas as variáveis interativas.

`%whos` Similar ao `%who`, com mais informação sobre cada variável.

Para finalizar, o Ipython é um excelente ambiente de trabalho interativo para computação científica, especialmente quando invocado coma opção `-pylab`. O modo `pylab` além de gráficos, também oferece uma série de comandos de compatibilidade com o MATLABTM(veja capítulo 3). O pacote principal do `numpy` também fica exposto no modo `pylab`. Subpacotes do `numpy` precisam ser importados manualmente.

4.2 Editores de Código

Na edição de programas em Python, um bom editor de código pode fazer uma grande diferença em produtividade. Devido a significância dos espaços em branco para a linguagem, um editor que mantém a indentação do código consistente, é muito importante para evitar bugs. Também é desejável que o editor conheça as regras de indentação do Python, por exemplo: indentar após “:”, indentar com espaços ao invés de tabulações. Outra característica desejável é a colorização do código de forma a ressaltar a sintaxe da linguagem. Esta característica aumenta, em muito, a legibilidade do código.

Os editores que podem ser utilizados com sucesso para a edição de programas em Python, se dividem em duas categorias básicas: editores genéricos e editores especializados na linguagem Python. Nesta seção, vamos examinar as principais características de alguns editores de cada categoria.

Editores Genéricos

Existe um sem-número de editores de texto disponíveis para o Ambiente Gnu/Linux. A grande maioria deles cumpre nossos requisitos básicos de indentação automática e colorização. Selecionei al-

guns que se destacam na minha preferência, quanto a usabilidade e versatilidade.

Emacs: Editor incrivelmente completo e versátil, funciona como ambiente integrado de desenvolvimento (figura 4.1). Precisa ter “python-mode” instalado. Para quem não tem experiência prévia com o Emacs, recomendo que o pacote **Easymacs**¹ seja também instalado. Este pacote facilita muito a interface do Emacs, principalmente para adição de atalhos de teclado padrão CUA. Pode-se ainda utilizar o Ipython dentro do Emacs.

Scite: Editor leve e eficiente, suporta bem o Python (executa o script com <F5>) assim como diversas outras linguagens. Permite configurar comando de compilação de C e Fortran, o que facilita o desenvolvimento de extensões. Completamente configurável (figura 4.2).

Gnu Nano: Levíssimo editor para ambientes de console, possui suporte a auto indentação e colorização em diversas linguagens, incluindo o Python (figura 4.3). Ideal para utilizar em conjunção com o Ipython (comando %edit).

Jedit: Incluí o Jedit nesta lista, pois oferece suporte ao desenvolvimento em Jython (ver Seção 5.5). Afora isso, é um editor bastante poderoso para Java e não tão pesado quanto o Eclipse (figura 4.4).

Kate/Gedit Editores padrão do KDE e Gnome respectivamente. Bons para uso casual, o Kate tem a vantagem de um console embutido.

¹<http://www.dur.ac.uk/p.j.heslin/Software/Emacs/Easymacs/>

82 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

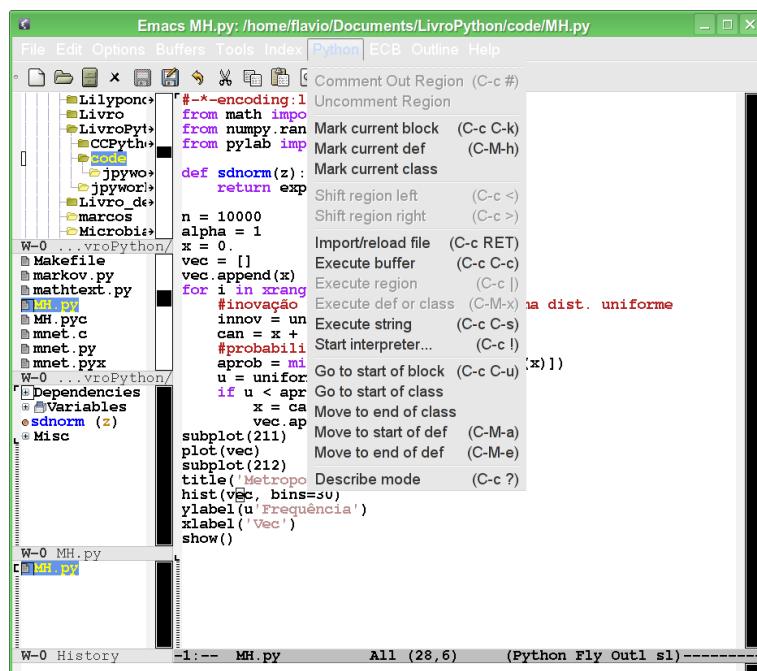


Figura 4.1: Editor emacs em modo Python e com “Code Browser” ativado

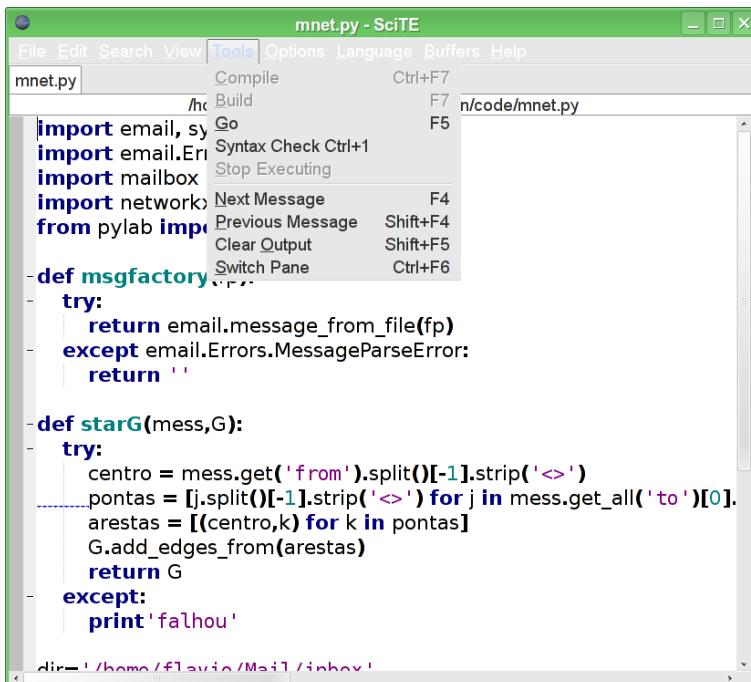


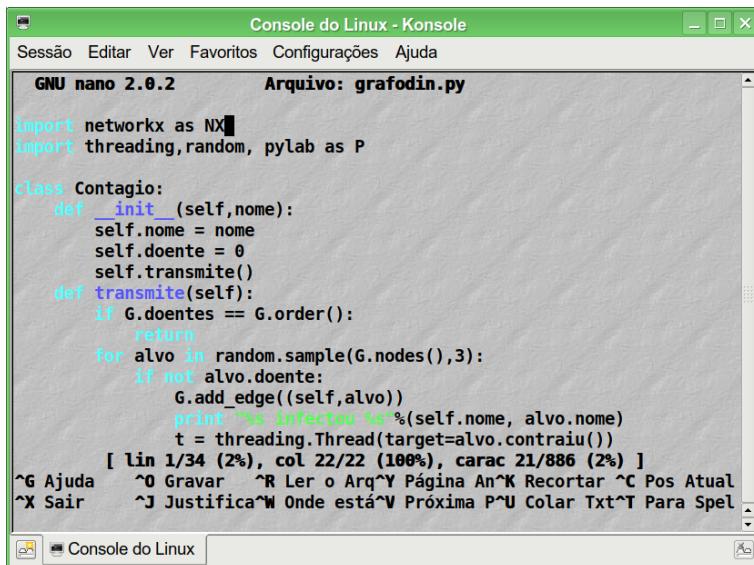
Figura 4.2: Editor Scite.

Editores Especializados

Editores especializados em Python tendem a ser mais do tipo IDE (ambiente integrado de desenvolvimento), oferecendo funcionalidades que só fazem sentido para gerenciar projetos de médio a grande porte, sendo “demais” para se editar um simples Script.

Boa-Constructor: O Boa-constructor é um IDE, voltado para o projetos que pretendam utilizar o WxPython como interface

84 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO



The screenshot shows a window titled "Console do Linux - Konsole". Inside, a terminal window is open with the title "GNU nano 2.0.2" and the file name "Arquivo: grafodin.py". The code in the editor is as follows:

```
import networkx as NX
import threading,random, pylab as P

class Contagio:
    def __init__(self,nome):
        self.nome = nome
        self.doente = 0
        self.transmite()
    def transmite(self):
        if G.doentes == G.order():
            return
        for alvo in random.sample(G.nodes(),3):
            if not alvo.doente:
                G.add_edge((self,alvo))
                print "%s infectou %s"%(self.nome, alvo.nome)
                t = threading.Thread(target=alvo.contraiu())
                [ lin 1/34 (2%), col 22/22 (100%), carac 21/886 (2%) ]
^G Ajuda ^O Gravar ^R Ler o Arq^Y Página An^K Recortar ^C Pos Atual
^X Sair ^J Justifica^W Onde está^V Próxima P^U Colar Txt^T Para Spel
```

The status bar at the bottom of the terminal window displays command-line navigation information.

Figura 4.3: Editor Gnu Nano.

gráfica. Neste aspecto ele é muito bom, permitindo construção visual da interface, gerando todo o código associado com a interface. Também traz um excelente depurador para programas em Python e dá suporte a módulos de extensão escritos em outras linguagens, como Pyrex ou C (figura 4.5).

Eric: O Eric também é um IDE desenvolvido em Python com a interface em PyQt. Possui boa integração com o gerador de interfaces Qt Designer, tornando muito fácil o desenvolvimento de interfaces gráficas com esta ferramenta. Também dispõe de ótimo depurador. Além disso o Eric oferece muitas outras funções, tais como integração com sistemas de controle

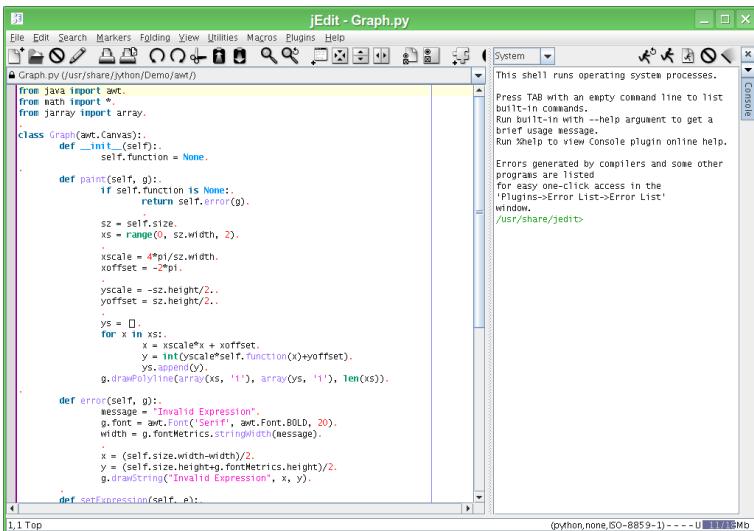


Figura 4.4: Editor “Jython” Jedit

de versão, geradores de documentação, etc.(Figura 4.6).

Pydev (Eclipse): O Pydev, é um IDE para Python e Jython desenvolvido como um plugin para Eclipse. Para quem já tem experiência com a plataforma Eclipse, pode ser uma boa alternativa, caso contrário, pode ser bem mais complicado de operar do que as alternativas mencionadas acima (Figura 4.7). Em termos de funcionalidade, equipara-se ao Eric e ao Boa-constructor.

86 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

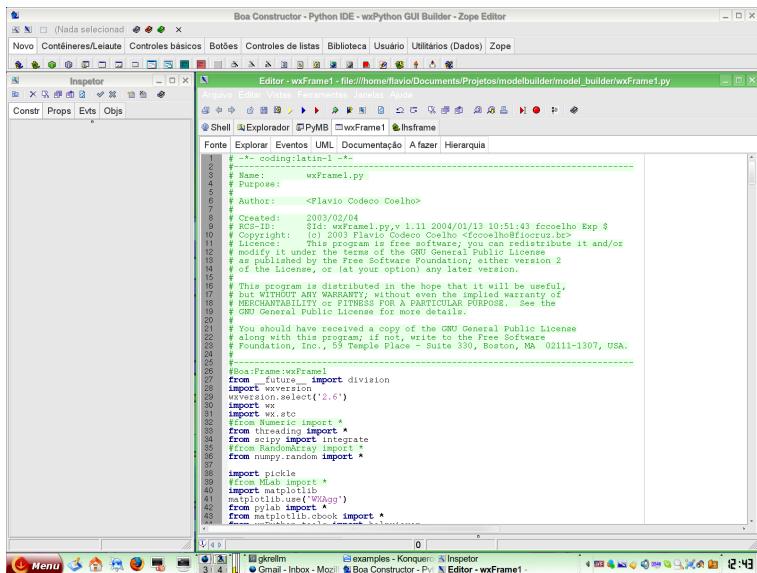


Figura 4.5: IDE Boa-Constructor

4.3 Controle de Versões em Software

Ao se desenvolver software, em qualquer escala, experimentamos um processo de aperfeiçoamento progressivo no qual o software passa por várias versões. Neste processo é muito comum, a um certo estágio, recuperar alguma funcionalidade que estava presente em uma versão anterior, e que, por alguma razão, foi eliminada do código.

Outro desafio do desenvolvimento de produtos científicos (software ou outros) é o trabalho em equipe em torno do mesmo objeto (frequentemente um programa). Normalmente cada membro da equipe trabalha individualmente e apresenta os seus resultados

The screenshot shows the Eric 4 IDE interface. The main window displays Python code for the `epigrass.py` file. The code handles initialization, translation settings, and various signal connections for UI components like buttons and dialogs. A local variable viewer on the right shows the state of variables at line 19. The bottom terminal window shows build logs and warnings related to Python 2 compatibility.

```

1 epigrass-CVS - /home/flavio/Documents/Projetos/Epigrass/Epigrass-devel/epigrass.py - Eric
File Edit View Debug Unitest Project Refactoring Extras Settings Window Bookmarks Help
epigrass.py qtgraph.py manager.py cpanel.py about.py dataObject.py dgraph.py
15
16     def __init__(self, parent = None, name = None, fl = 0):
17
18         # set Translation
19         self.parent = os.getcwd()
20         os.chdir(os.getcwd())
21         if not os.path.exists('epigrassrc'):
22             try:
23                 self.conf = self.loadConfigFile('epigrassrc')
24                 lang = self.conf['settings.language']
25             except:
26                 lang = ''
27             #print lang
28         else:
29             tr = QTranslator(app)
30             loadLang(app,tr,lang)
31
32         MainPanel.__init__(self,parent,name,fl)
33
34         # Overload connections
35         self.connect(self.editCancelButton,SIGNAL("released()"),self.onEditScript)
36         self.connect(self.chooseButton,SIGNAL("released()"),self.chooseScript)
37         self.connect(self.buttonExit,SIGNAL("released()"),self.onExit)
38         self.connect(self.buttonHelp,SIGNAL("released()"),self.onHelp)
39         self.connect(self.buttonHelpAbout,SIGNAL("released()"),self.onHelpAbout)
40         self.connect(self.databaseBackup,SIGNAL("released()"),self.onDbBackup)
41         self.connect(self.repOpen,SIGNAL("released()"),self.onRepOpen)
42         self.connect(self.playButton,SIGNAL("released()"),self.onPlayButton)
43         self.connect(self.aboutButton,SIGNAL("released()"),self.onVisual)
44
45
1 Python 2.7 (No. 1, Nov 10 2006, 14:34:40)
2 [GCC 4.1.2 (Gentoo 4.1.2-r1)] on Sabayon, No Optimized, threaded
3 >>> /usr/lib/python2.7/site-packages/mstplotlib/_init_.pyc[15]: UserWarning: Module Epigrass was already imported from /home/flavio/Documents/Projetos/Epigrass/Epigrass-devel/Epigrass/
4 _import_.pyc[1]: reusing module _mstplotlib_namepatch_.name ...
5 /usr/lib/python2.7/site-packages/numpy/1.6.1-py2.7-unicore-utf8-egghumpmptypeslib.egg[12]: UserWarning: All features of ctypes interface may not work with ctypes < 1.0.1
6 warnings.warn(Fall features of ctypes interface may not work with ... )
7
8
rw File: /home/flavio/.../Projetos/Epigrass/Epigrass-devel/epigrass.py Line: 19 Pos: 0

```

Figura 4.6: IDE Eric.

para a equipe em reuniões regulares. O que fazer quando modificações desenvolvidas por diferentes membros de uma mesma equipe se tornam incompatíveis? Ou mesmo, quando dois ou mais colaboradores estão trabalhando em partes diferentes de um programa, mas que precisam uma da outra para funcionar?

O tipo de ferramenta que vamos introduzir nesta seção, busca resolver ou minimizar os problemas supracitados e pode ser aplicado também ao desenvolvimento colaborativo de outros tipos de documentos, não somente programas.

Como este é um livro baseado na linguagem Python, vamos utilizar um sistema de controle de versões desenvolvido inteiramente

88 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

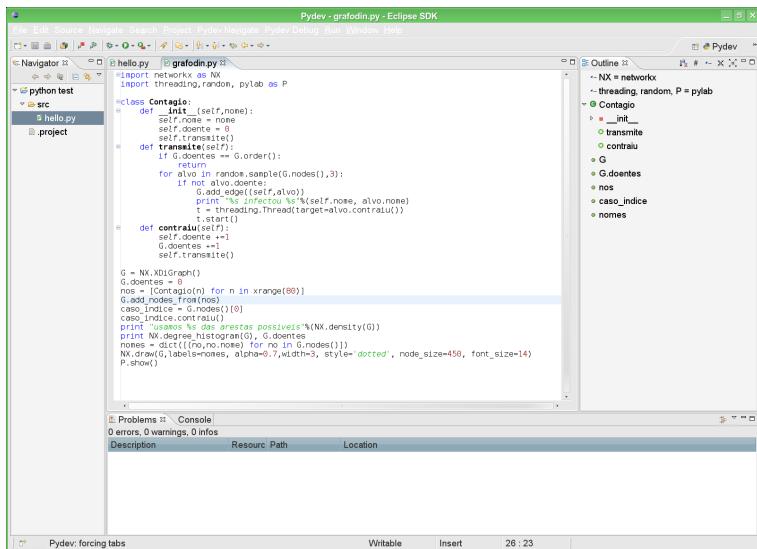


Figura 4.7: IDE Pydev

em Python: [Mercurial](#)². Na prática o mecanismo por trás de todos os sistemas de controle de versão é muito similar. Migrar de um para outro é uma questão de aprender novos nomes para as mesmas operações. Além do mais, o uso diário de sistema de controle de versões envolve apenas dois ou três comandos.

Entendendo o Mercurial

O Mercurial é um sistema de controle de versões descentralizado, ou seja, não há nenhuma noção de um servidor central onde fica

²<http://www.selenic.com/mercurial>

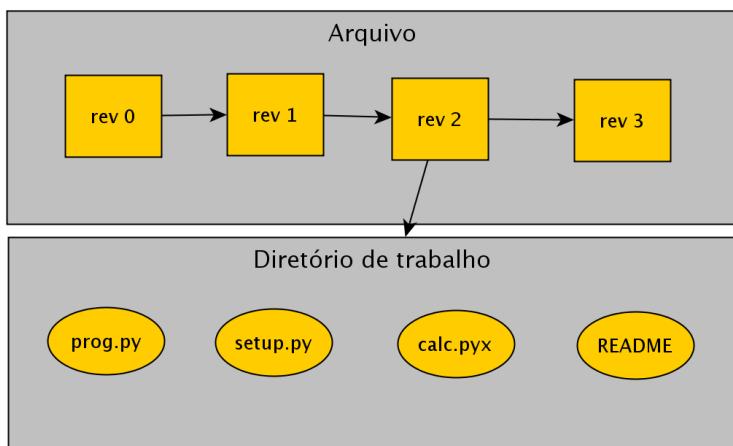


Figura 4.8: Diagrama de um repositório do Mercurial.

depositado o código. Repositórios de códigos são diretórios que podem ser “clonados” de uma máquina para outra.

Então, em que consiste um repositório? A figura 4.8 é uma representação diagramática de um repositório. Para simplificar nossa explanação, consideremos que o repositório já foi criado ou clonado de alguém que o criou. Veremos como criar um repositório a partir do zero, mais adiante.

De acordo com a figura 4.8, um repositório é composto por um Arquivo³ e por um diretório de trabalho. O Arquivo contém a história completa do projeto. O diretório de trabalho contém uma cópia dos arquivos do projeto em um determinado ponto no tempo (por exemplo, na revisão 2). É no diretório de trabalho que o pesquisador trabalha e atualiza os arquivos.

³Doravante grafado com “A” maiúsculo para diferenciar de arquivos comuns(files).

90 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

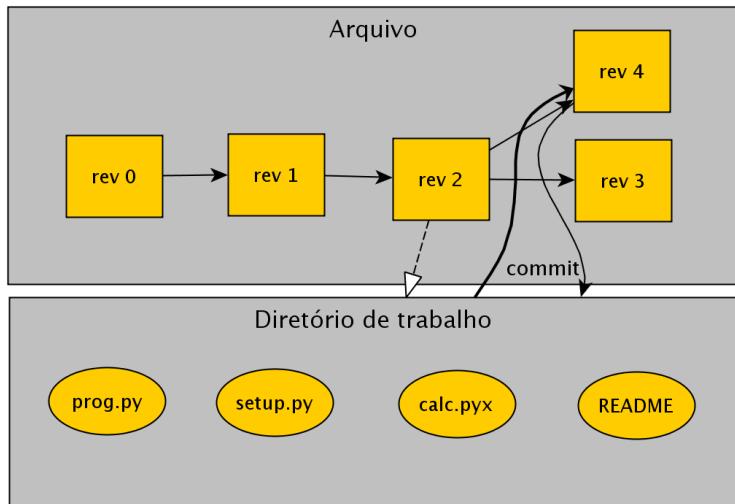


Figura 4.9: Operação de “commit”.

Ao final de cada ciclo de trabalho, o pesquisador envia suas modificações para o arquivo numa operação denominada “commit”(figura 4.9)⁴.

Após um **commit**, como as fontes do diretório de trabalho não correspondiam à última revisão do projeto, o **Mercurial** automaticamente cria uma ramificação no arquivo. Com isso passamos a ter duas linhas de desenvolvimento seguindo em paralelo, com o nosso diretório de trabalho pertencendo ao ramo iniciado pela revisão 4.

O **Mercurial** agrupa as mudanças enviadas por um usuário (via **commit**), em um conjunto de mudanças atômico, que constitui

⁴Vou adotar o uso da palavra **commit** para me referir a esta operação daqui em diante. Optei por não tentar uma tradução pois este termo é um jargão dos sistemas de controle de versão.

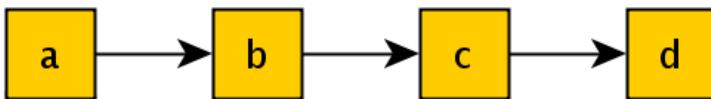


Figura 4.10: Repositório da Ana.

uma revisão. Estas revisões recebem uma numeração sequencial (figura 4.9). Mas como o `Mercurial` permite desenvolvimento de um mesmo projeto em paralelo, os números de revisão para diferentes desenvolvedores poderiam diferir. Por isso cada revisão também recebe um identificador global, consistindo de um número hexadecimal de quarenta dígitos.

Além de ramificações, fusões (“merge”) entre ramos podem ocorrer a qualquer momento. Sempre que houver mais de um ramo em desenvolvimento, o `Mercurial` denominará as revisões mais recentes de cada ramo(`heads`, `cabeças`). Dentre estas, a que tiver maior número de revisão será considerada a ponta (`tip`) do repositório.

Exemplo de uso:

Nestes exemplos, exploraremos as operações mais comuns num ambiente de desenvolvimento em colaboração utilizando o `Mercurial`.

Vamos começar com nossa primeira desenvolvedora, chamada Ana. Ana possui um arquivo como mostrado na figura 4.10.

Nosso segundo desenvolvedor, Bruno, acabou de se juntar ao time e clona o repositório Ana⁵.

```

1 $ hg clone ssh://maquinadana/projeto
      meuprojeto
2 requesting all changes
  
```

⁵Assumimos aqui que a máquina da ana está executando um servidor `ssh`

92 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

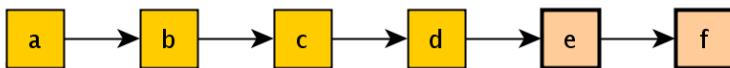


Figura 4.11: Modificações de Bruno.

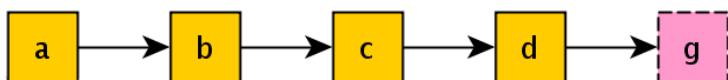


Figura 4.12: Modificações de Ana.

```
3 adding changesets
4 adding manifests
5 adding file changes
6 added 4 changesets with 4 changes to 2 files
```

URLs válidas:

file:///
http:///
https:///
ssh:///
static-http://

Após o comando acima, Bruno receberá uma cópia completa do arquivo de Ana, mas seu diretório de trabalho, `meu projeto`, permanecerá independente. Bruno está ansioso para começar a trabalhar e logo faz dois `commits` (figura 4.11).

Enquanto isso, em paralelo, Ana também faz suas modificações (figura 4.12).

Bruno então decide “puxar” o repositório de Ana para sincronizá-lo com o seu.

```
1 $ hg pull
```

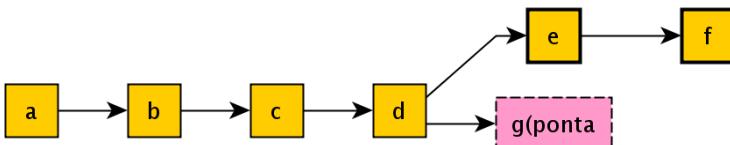


Figura 4.13: Repositório atualizado de Bruno.

```

2 pulling from ssh://maquinadaana/projeto
3 searching for changes
4 adding changesets
5 adding manifests
6 adding file changes
7 added 1 changesets with 1 changes to 1 files
8 (run 'hg heads' to see heads, 'hg merge' to
   merge)
  
```

O comando `hg pull`, se não especificada a fonte, irá “puxar” da fonte de onde o repositório local foi clonado. Este comando atualizará o Arquivo local, mas não o diretório de trabalho.

Após esta operação o repositório de Bruno fica como mostrado na figura 4.13. Como as mudanças feitas por Ana, foram as últimas adicionadas ao repositório de Bruno, esta revisão passa a ser a ponta do Arquivo.

Bruno agora deseja fundir seu ramo de desenvolvimento, com a ponta do seu Arquivo que corresponde às modificações feitas por Ana. Normalmente, após puxar modificações, executamos `hg update` para sincronizar nosso diretório de trabalho com o Arquivo recém atualizado. Então Bruno faz isso.

```

1 $ hg update
2 this update spans a branch affecting the
   following files:
3 hello.py (resolve)
  
```

94 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

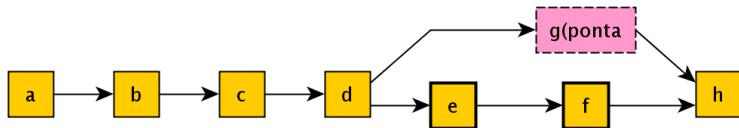


Figura 4.14: Repositório de Bruno após a fusão.

```
4 aborting update spanning branches!
5 (use 'hg merge' to merge across branches or
   'hg update -C' to lose changes)
```

Devido à ramificação no Arquivo de Bruno, o comando `update` não sabe a que ramo fundir as modificações existentes no diretório de trabalho de Bruno. Para resolver isto, Bruno precisará fundir os dois ramos. Felizmente esta é uma tarefa trivial.

```
1 $ hg merge tip
2 merging hello.py
```

No comando `merge`, se nenhuma revisão é especificada, o diretório de trabalho é cabeça de um ramo e existe apenas uma outra cabeça, as duas cabeças serão fundidas. Caso contrário uma revisão deve ser especificada.

Pronto! agora o repositório de Bruno ficou como a figura 4.14.

Agora, se Ana puxar de Bruno, receberá todas as modificações de Bruno e seus repositórios estarão plenamente sincronizados, como a figura 4.14.

Criando um Re却itório

Para criar um repositório do zero, é preciso apenas um comando:

```
1 $ hg init
```

Quando o diretório é criado, um diretório chamado `.hg` é criado dentro do diretório de trabalho. O Mercurial irá armazenar todas as informações sobre o repositório no diretório `.hg`. O conteúdo deste diretório não deve ser alterado pelo usuário.

Para saber mais

Naturalmente, muitas outras coisas podem ser feitas com um sistema de controle de versões. O leitor é encorajado a consultar a documentação do Mercurial para descobri-las. Para servir de referência rápida, use o comando `hg help -v <comando>` com qualquer comando da lista abaixo.

add Adiciona o(s) arquivo(s) especificado(s) no próximo commit.

addrmove Adiciona todos os arquivos novos, removendo os faltantes.

annotate Mostra informação sobre modificações por linha de arquivo.

archive Cria um arquivo (compactado) não versionado, de uma revisão especificada.

backout Reverte os efeitos de uma modificação anterior.

branch Altera ou mostra o nome do ramo atual.

branches Lista todas os ramos do repositório.

bundle Cria um arquivo compactado contendo todas as modificações não presentes em um outro repositório.

cat Retorna o arquivo especificado, na forma em que ele era em dada revisão.

clone Replica um repositório.

96 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

- commit** Arquiva todas as modificações ou os arquivos especificados.
- copy** Copia os arquivos especificados, para outro diretório no próximo **commit**.
- diff** Mostra diferenças entre revisões ou entre os arquivos especificados.
- export** Imprime o cabeçalho e as diferenças para um ou mais conjuntos de modificações.
- grep** Busca por palavras em arquivos e revisões específicas.
- heads** Mostra cabeças atuais.
- help** Mostra ajuda para um comando, extensão ou lista de comandos.
- identify** Imprime informações sobre a cópia de trabalho atual.
- import** Importa um conjunto ordenado de atualizações (patches).
Este comando é a contrapartida de Export.
- incoming** Mostra novos conjuntos de modificações existentes em um dado repositório.
- init** Cria um novo repositório no diretório especificado. Se o diretório não existir, ele será criado.
- locate** Localiza arquivos.
- log** Mostra histórico de revisões para o repositório como um todo ou para alguns arquivos.
- manifest** Retorna o manifesto (lista de arquivos controlados) da revisão atual ou outra.
- merge** Funde o diretório de trabalho com outra revisão.

outgoing Mostra conjunto de modificações não presentes no repositório de destino.

parents Mostra os “pais” do diretório de trabalho ou revisão.

paths Mostra definição de nomes simbólicos de caminho.

pull “Puxa” atualizações da fonte especificada.

push Envia modificações para o repositório destino especificado.
É a contra-partida de **pull**.

recover Desfaz uma transação interrompida.

remove Remove os arquivos especificados no próximo commit.

rename Renomeia arquivos; Equivalente a **copy** + **remove**.

revert Reverte arquivos ao estado em que estavam em uma dada revisão.

rollback Desfaz a última transação neste repositório.

root Imprime a raiz do diretório de trabalho corrente.

serve Exporta o diretório via HTTP.

showconfig Mostra a configuração combinada de todos os arquivos **hgrc**.

status Mostra arquivos modificados no diretório de trabalho.

tag Adiciona um marcador para a revisão corrente ou outra.

tags Lista marcadores do repositório.

tip Mostra a revisão “ponta”.

unbundle Aplica um arquivo de modificações.

98 CAPÍTULO 4. FERRAMENTAS DE DESENVOLVIMENTO

update Atualiza ou funde o diretório de trabalho.

verify Verifica a integridade do repositório.

version Retorna versão e informação de copyright.

Capítulo 5

Interagindo com Outras Linguagens

Introdução a vários métodos de integração do Python com outras linguagens. Pré-requisitos: Capítulos 1 e 2.

5.1 Introdução

O Python é uma linguagem extremamente poderosa e versátil, perfeitamente apta a ser, não somente a primeira, como a última linguagem de programação que um cientista precisará aprender. Entretanto, existem várias situações nas quais torna-se interessante combinar o seu código escrito em Python com códigos escritos em outras linguagens. Uma das situações mais comuns, é a necessidade de obter maior performance em certos algoritmos através da re-implementação em uma linguagem compilada. Outra Situação comum é possibilidade de se utilizar de bibliotecas desenvolvidas em outras linguagens e assim evitar ter que reimplementá-las em Python.

O Python é uma linguagem que se presta, extremamente bem, a estas tarefas existindo diversos métodos para se alcançar os obje-

tivos descritos no parágrafo acima. Neste capítulo, vamos explorar apenas os mais práticos e eficientes, do ponto de vista do tempo de implementação.

5.2 Integração com a Linguagem C

A linguagem C é uma das linguagens mais utilizadas no desenvolvimento de softwares que requerem alta performance. Um bom exemplo é o Linux (kernel) e a própria linguagem Python. Este fato torna o C um candidato natural para melhorar a performance de programas em Python.

Vários pacotes científicos para Python como o *Numpy* e *Scipy*, por exemplo, tem uma grande porção do seu código escrito em C para máxima performance. Coincidemente, o primeiro método que vamos explorar para incorporar código C em programas Python, é oferecido como parte do pacote *Scipy*.

Weave

O `weave` é um módulo do pacote `scipy`, que permite inserir trechos de código escrito em C ou C++ dentro de programas em Python. Existem várias formas de se utilizar o `weave` dependendo do tipo de aplicação que se tem. Nesta seção, vamos explorar apenas a aplicação do módulo `inline` do `weave`, por ser mais simples e cobrir uma ampla gama de aplicações. Além disso, utilizações mais avançadas do `weave`, exigem um conhecimento mais profundo da linguagem C, o que está fora do escopo deste livro. Caso os exemplos incluídos não satisfaçam os anseios do leitor, existe uma farta documentação no site www.scipy.org.

Vamos começar a explorar o `weave` com um exemplo trivial (computacionalmente) um simples loop com uma única operação (exemplo 5.1).

Listagem 5.1: Otimização de loops com o `weave`

```
1  #-*-encoding: latin-1-*-
2 # Disponível no pacote de programas como:
3   weaveloop.py
4 from scipy.weave import inline, converters
5 import time
6 from pylab import *
7
8 code=""""
9 int i=0;
10 while (i < n)
11     {
12         i ^ 3;
13         i++;
14     }"""
15 def loopp(n):
16     i=0
17     while i < n:
18         i**3
19         i+=1
20
21 def loopc(n):
22     return inline(code,[ 'n' ],
23                  type_converters=converters.blitz
24                  , compiler='gcc')
25
26 def bench(max):
27     ptim = []
28     ctim = []
29     for n in xrange(100,max,100):
30         t = time.time()
31         loopp(n)
32         t1=time.time()-t
33         ptim.append(t1)
34         ctim.append(t1)
35
36 print "C loop took %f seconds" % (sum(ctim)/len(ctim))
37 print "Python loop took %f seconds" % (sum(ptim)/len(ptim))
```

```

30         ptim.append(t1)
31         t = time.time()
32         loopc(n)
33         t1=time.time()-t
34         ctim.append(t1)
35     return ptim,ctim,max
36
37 ptim,ctim,max = bench(10000)
38 semilogy(xrange(100,max,100),ptim,'b-o',
39           xrange(100,max,100),ctim,'g-o')
40 ylabel('tempo em segundos'); xlabel(u'Número
41       de operações'); grid()
42 legend(['Python','C'])
43 savefig('weaveloop.png',dpi=400)
44 show()

```

No exemplo 5.1 podemos ver como funciona o `weave`. Uma string contém o código C a ser compilado. A função `inline` compila o código em questão, passando para o mesmo as variáveis necessárias.

Note que, na primeira execução do loop, o `weave` é mais lento que o Python, devido à compilação do código; mas em seguida, com a rotina já compilada e carregada na memória, este atraso não existe mais.

O `weave.inline` tem uma performance inferior à de um programa em C equivalente, executado fora do Python. Mas a sua simplicidade de utilização, compensa sempre que se puder obter um ganho de performance sobre o Python puro.

Listagem 5.2: Calculando iterativamente a série de Fibonacci em Python e em C(`weave.inline`)

```

1 # Disponível no pacote de programas como:
2   weavefib.py
2 from scipy import weave

```

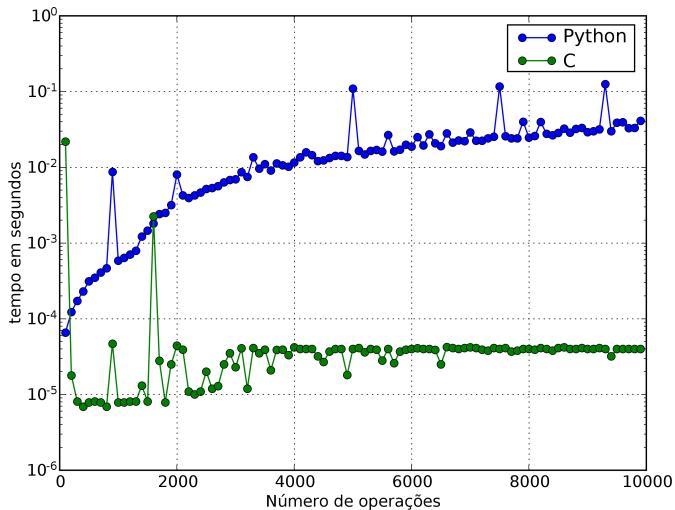


Figura 5.1: Comparação da performance entre o Python e o C (weave) para um loop simples. O eixo y está em escala logarítmica para facilitar a visualização das curvas.

```

3| from scipy.weave import converters
4| import time
5| from scipy.stats import bayes_mvs
6| from pylab import *
7|
8| code=\
9| """
10|     while (b < n)
11|         {
12|             int savea = a;

```

```
13         a=b;
14         b=savea+b;
15     }
16 return _val = a;
17 """
18 def fibp(n):
19     """
20         Calcula série de Fibonacci em Python
21     """
22     a,b = 0,1
23     while b < n:
24         a,b = b,a+b
25     return a
26
27 def fibw(n):
28     """
29         Versão weave.inline
30     """
31     a = 0
32     b = 1
33     return weave.inline(code,[ 'n' , 'a' , 'b'
34                           ],
35                           type_converters=converters.blitz ,
36                           compiler='gcc ')
37 ptim=[];ctim=[]
38 for i in range(100):
39     t = time.time()
40     pn = fibp(1000000000)
41     pt = time.time()-t;ptim.append(pt)
42     fibw(10)#to compile before timing
43     t = time.time()
44     cn = fibw(1000000000)
45     ct = time.time()-t; ctim.append(ct)
46 print 'Tempo médio no python: %g segundos '%
```

```
45     pt
45 print 'Tempo médio no weave: %g segundos '%
45     ct
46 Stats = bayes_mvs(array(ptim)/array(ctim))
47 print "Aceleração média: %g ± %g "%(Stats
47     [0][0],
48 Stats[2][0])
```

No exemplo 5.2, o ganho de performance do `weave.inline` já não é tão acentuado.

Listagem 5.3: Executando o código do exemplo 5.2.

```
1 $ python weavefib.py
2 Tempo médio no python: 1.69277e-05 segundos
3 Tempo médio no weave: 1.3113e-05 segundos
4 Aceleração média: 1.49554 ± 0.764275
```

Ctypes

O pacote `ctypes`, parte integrante do Python a partir da versão 2.5, é um módulo que nos permite invocar funções de bibliotecas em C pré-compiladas, como se fossem funções em Python. Apesar da aparente facilidade de uso, ainda é necessário ter consciência do tipo de dados a função, que se deseja utilizar, requer. Por conseguinte, é necessário que o usuário tenha um bom conhecimento da linguagem C.

Apenas alguns objetos do python podem ser passados para funções através do `ctypes`: `None`, `inteiros`, `inteiros longos`, `strings`, e `strings unicode`. Outros tipos de dados devem ser convertidos, utilizando tipos fornecidos pelo `ctypes`, compatíveis com C.

Dado seu estágio atual de desenvolvimento, o `ctypes` não é a ferramenta mais indicada ao cientista que deseja fazer uso das

conveniências do C em seus programas Python. Portanto, vamos apenas dar dois exemplos básicos para que o leitor tenha uma ideia de como funciona o `ctypes`. Para maiores informações recomendamos o tutorial do `ctypes` (<http://python.net/crew/theller/ctypes/tutorial.html>)

Nos exemplos abaixo assumimos que o leitor está utilizando Linux pois o uso do `ctypes` no Windows não é idêntico.

Listagem 5.4: Carregando bibliotecas em C

```
1 >>> from ctypes import *
2 >>> libc = cdll.LoadLibrary("libc.so.6")
3 >>> libc
4 <CDLL 'libc.so.6', handle ... at ... >
```

Uma vez importada uma biblioteca (listagem 5.4), podemos chamar funções como atributos das mesmas.

Listagem 5.5: Chamando funções em bibliotecas importadas com o `ctypes`

```
1 >>> libc.printf
2 <_FuncPtr object at 0x...>
3 >>> print libc.time(None)
4 1150640792
5 >>> printf = libc.printf
6 >>> printf("Ola, %s\n", "Mundo!")
7 Ola, Mundo!
```

Pyrex

O Pyrex é uma linguagem muito similar ao Python feita para gerar módulos em C para o Python. Desta forma, envolve um pouco mais de trabalho por parte do usuário, mas é de grande valor para acelerar código escrito em Python com pouquíssimas modificações.

O Pyrex não inclui todas as possibilidades da linguagem Python. As principais modificações são as que se seguem:

- Não é permitido definir funções dentro de outras funções;
- definições de classe devem ocorrer apenas no espaço de nomes global do módulo, nunca dentro de funções ou de outras classes;
- Não é permitido `import *`. As outras formas de `import` são permitidas;
- Geradores não podem ser definidos em Pyrex;
- As funções `globals()` e `locals()` não podem ser utilizadas.

Além das limitações acima, existe um outro conjunto de limitações que é considerado temporário pelos desenvolvedores do Pyrex. São as seguintes:

- Definições de classes e funções não podem ocorrer dentro de estruturas de controle (if, elif, etc.);
- Operadores *in situ* (`+=`, `*=`, etc.) não são suportados pelo Pyrex;
- List comprehensions não são suportadas;
- Não há suporte a Unicode.

Para exemplificar o uso do Pyrex, vamos implementar uma função geradora de números primos em Pyrex (listagem 5.6).

Listagem 5.6: Calculando números primos em Pyrex

```
1 # Calcula numeros primos
2 def primes( int kmax ):
3     cdef int n, k, i
4     cdef int p[1000]
```

```
5     result = []
6     if kmax > 1000:
7         kmax = 1000
8     k = 0
9     n = 2
10    while k < kmax:
11        i = 0
12        while i < k and n % p[i] <> 0:
13            i = i + 1
14        if i == k:
15            p[k] = n
16            k = k + 1
17            result.append(n)
18        n = n + 1
19    return result
```

Vamos analisar o código Pyrex, nas linhas onde ele difere do que seria escrito em Python. Na linha 2 encontramos a primeira peculiaridade: o argumento de entrada `kmax` é definido como inteiro por meio da expressão `int kmax`. Em Pyrex, devemos declarar os tipos das variáveis. Nas linhas 3 e 4 também ocorre a declaração dos tipos das variáveis que serão utilizadas na função. Note como é definida uma lista de inteiros. Se uma variável não é declarada, o Pyrex assume que ela é um objeto Python.

Quanto mais variáveis conseguirmos declarar como tipos básicos de C, mais eficiente será o código C gerado pelo Pyrex. A variável `result`(linha 5) não é declarada como uma lista de inteiros, pois não sabemos ainda qual será seu tamanho. O restante do código é equivalente ao Python. Devemos apenas notar a preferência do laço `while` ao invés de um laço do tipo `for i in range(x)`. Este ultimo seria mais lento devido a incluir a função `range` do Python.

O próximo passo é gerar a versão em C da listagem 5.6, compilar e linká-lo, transformando-o em um módulo Python.

Listagem 5.7: Gerando Compilando e linkando

```
1 $ pyrexc primes.pyx
2 $ gcc -c -fPIC -I/usr/include/python2.4/
      primes.c
3 $ gcc -shared primes.o -o primes.so
```

Agora vamos comparar a performance da nossa função com uma função em Python razoavelmente bem implementada (Listagem 5.8). Afinal temos que dar uma chance ao Python, não?

Listagem 5.8: Calculando números primos em Python

```
1 # Disponivel no pacote de programas como:
      primes2.py
2 def primes(N):
3     if N <= 3:
4         return range(2,N)
5     #testa apenas os numeros impares
6     primos = [2] + range(3,N,2)
7     index = 1
8     #convertendo em inteiro
9     #para acelerar comparacao abaixo
10    top = int(N ** 0.5)
11    while 1:
12        i = primos[index]
13        if i>top:
14            break
15        index += 1
16        primos = [x for x in primos if (x %
17                  i) or (x == i)]
18    return primos
primes(100000)
```

Comparemos agora a performance das duas funções para encontrar todos os números primos menores que 100000. Para esta

CAPÍTULO 5. INTERAGINDO COM OUTRAS LINGUAGENS

110

comparação utilizaremos o ipython que nos facilita esta tarefa através da função mágica `%timeit`.

Listagem 5.9: Comparando performances dentro do ipython

```
1 In [1]: from primes import primes
2 In [2]: from primes2 import primes as primesp
3 In [3]: %timeit primes(100000)
4 10 loops, best of 3: 19.6 ms per loop
5 In [4]: %timeit primesp(100000)
6 10 loops, best of 3: 512 ms per loop
```

Uma das desvantagens do Pyrex é a necessidade de compilar e linkar os programas antes de poder utilizá-los. Este problema se acentua se seu programa Python utiliza extensões em Pyrex e precisa ser distribuído a outros usuários. Felizmente, existe um meio de automatizar a compilação das extensões em Pyrex, durante a instalação de um módulo. O pacote setuptools, dá suporte à compilação automática de extensões em Pyrex. Basta escrever um script de instalação similar ao da listagem 5.10. Uma vez criado o script (batizado, por exemplo, de `setuppyx.py`), para compilar a nossa extensão, basta executar o seguinte comando: `python setupix.py build`.

Para compilar uma extensão Pyrex, o usuário deve naturalmente ter o Pyrex instalado. Entretanto para facilitar a distribuição destas extensões, o pacote setuptools, na ausência do Pyrex, procura a versão em C gerada pelo autor do programa, e se ela estiver incluída na distribuição do programa, o setuptools passa então para a etapa de compilação e linkagem do código C.

Listagem 5.10: Automatizando a compilação de extensões em Pyrex por meio do setuptools

```
1 # Disponível no pacote de programas como:
#       setuppyx.py
2 from setuptools import setup
```

Listagem 5.11: Utilizando Setuptools para compilar extensões em Pyrex

```
1 | import setuptools
2 | from setuptools.extension import Extension
3 |
4 |
5 | from setuptools.extension import Extension
6 |
7 | primespix = Extension(name = 'primespix',
8 |                         sources = [ 'primes.pyx'
9 |                                     ]
10 |                         )
11 | setup(name = 'primes',
12 |       ext_modules = [primespix]
13 |       )
```

5.3 Integração com C++

A integração de programas em Python com bibliotecas em C++ é normalmente feita por meio ferramentas como SWIG (www.swig.org), SIP(www.riverbankcomputing.co.uk/sip/) ou Boost.Python (<http://www.boost.org/libs/python/>). Estas ferramentas, apesar de relativamente simples, requerem um bom conhecimento de C++ por parte do usuário e, por esta razão, fogem ao escopo deste capítulo. No entanto, o leitor que deseja utilizar código já escrito em C++ pode e deve se valer das ferramentas supracitadas, cuja documentação é bastante clara.

Elegemos para esta seção sobre C++. uma ferramenta original. O ShedSkin.

Shedskin

O ShedSkin (<http://shed-skin.blogspot.com/>) se auto intitula “um compilador de Python para C++ otimizado”. O que ele faz , na verdade, é converter programas escritos em Python para C++, permitindo assim grandes ganhos de velocidade. Apesar de seu potencial, o ShedSkin ainda é uma ferramenta um pouco limitada. Uma de suas principais limitações, é que o programa em Python a ser convertido deve possuir apenas variáveis “estáticas”, ou seja as variáveis devem manter-se do mesmo tipo durante toda a execução do programa. Se uma variável é definida como um número inteiro, nunca poderá receber um número real, uma lista ou qualquer outro tipo de dado.

O ShedSkin também não suporta toda a biblioteca padrão do Python na versão testada (0.0.15). Entretanto, mesmo com estas limitações, esta ferramenta pode ser muito útil. Vejamos um exemplo: A integração numérica de uma função, pela regra do trapézio. Esta regra envolve dividir a área sob a função em um dado intervalo em multiplos trapézios e somar as suas áreas(figura 5.2).

Matemáticamente, podemos expressar a regra trapezoidal da seguinte fórmula.

$$\int_a^b f(x) dx \approx \frac{h}{2}(f(a) + f(b)) + h \sum_{i=1}^{n-1} f(a + ih), \quad h = \frac{b-a}{n} \quad (5.1)$$

Na listagem 5.12 podemos ver uma implementação simples da regra trapezoidal.

Listagem 5.12: implementação da regra trapezoidal(utilizando laço for) conforme especificada na equação 5.3

```
1 | #-*-encoding: latin-1-*-
2 | # Disponivel no pacote de programas como:
   trapintloop.py
3 | # copyright 2007 by Flávio Codeço Coelho
```

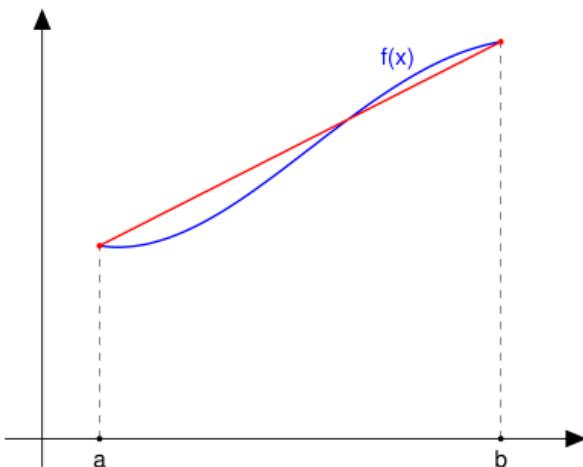


Figura 5.2: Regra trapezoidal para integração aproximada de uma função.

```

4| from time import clock
5| from math import exp, log, sin
6|
7| def tint(a,b,fun):
8|     """
9|         Integração numérica pela regra
10|             trapezoidal
11|     """
12|     n = 5000000
13|     h2 = (b-a)/(n*2.)
14|     res = h2*fun(a)+h2*fun(b)
15|     for i in xrange(n):
16|         p = a+i*2*h2
17|         res += 2*h2*fun(p)

```

```

17     return res
18
19 # Rodando as funcoes em Python
20 start = clock()
21 r = tint(1.0,5.0,lambda(x):1+x)
22 stop = clock()
23 print 'resultado: %d\nTempo: %s seg'%(r,stop
24         -start)
24 start = clock()
25 r = tint(1.0,5.0,lambda(x):exp(x**2*log(x+x*
26         sin(x))))
26 stop = clock()
27 print 'resultado: %d\nTempo: %s seg'%(r,stop
28         -start)

```

Executando o script da Listagem 5.12 (`trapintloop.py`) observamos o tempo de execução da integração das duas funções.

Listagem 5.13: Executando a listagem 5.12

```

1 $ python trapintloop.py
2 resultado: 16
3 Tempo: 11.68 seg
4 resultado: 49029
5 Tempo: 26.96 seg

```

Para converter o script 5.12 em C++ tudo o que precisamos fazer é:

Listagem 5.14: Verificando o tempo de execução em C++

```

1 $ ss trapintloop.py
2 *** SHED SKIN Python-to-C++ Compiler 0.0.15
3 *** Copyright 2005, 2006 Mark Dufour; License
      GNU GPL version 2 or later (See LICENSE)

```

```

4 ( If your program does not compile, please
   mail me at mark.dufour@gmail.com!!)

5 *WARNING* trapintloop:13: 'xrange', '
   enumerate' and 'reversed' return lists
   for now
6 [ iterative type analysis..]
7 /**
8 iterations: 2 templates: 55
9 [ generating c++ code..]
10 $ make run
11 g++ -O3 -I ...
12 ./trapintloop
13 resultado: 16
14 Tempo: 0.06 seg
15 resultado: 49029
16 Tempo: 1.57 seg
17

```

Com estes dois comandos, geramos, compilamos e executamos uma versão C++ do programa listado em 5.12. O código C++ gerado pelo Shed-Skin pode ser conferido na listagem 5.15

Como pudemos verificar, o ganho em performance é considerável. Lamentavelmente, o Shed-Skin não permite criar módulos “de extensão” que possam ser importados por outros programas em Python, só programas independentes. Mas esta limitação pode vir a ser superada no futuro.

Listagem 5.15: Código C++ gerado pelo Shed-skin a partir do script trapintloop.py

```

1 #include "trapintloop.hpp"
2
3 namespace __trapintloop__ {
4
5 str *const _0;

```

```
6  double r , start , stop ;
7
8
9  double __lambda0__(double x) {
10
11     return (1+x);
12 }
13
14 double __lambda1__(double x) {
15
16     return exp(((x*x)*log((x+(x*sin(x))))));
17 }
18
19 int __main() {
20     const_0 = new str("resultado: %d\nTempo:
21             %s seg");
22
23     start = clock();
24     r = tint(1.0 , 5.0 , __lambda0__);
25     stop = clock();
26     print("%s\n" , __mod(const_0 , __int(r) ,
27             new float_((stop-start))));
28     start = clock();
29     r = tint(1.0 , 5.0 , __lambda1__);
30     stop = clock();
31     print("%s\n" , __mod(const_0 , __int(r) ,
32             new float_((stop-start))));
33
34 /**
35      Integração numérica pela regra
36          trapezoidal
37 */
38
39 double tint(double a , double b , lambda0 fun)
```

```
36     {
37     double h2, p, res;
38     int __0, __1, i, n;
39
40     n = 5000000;
41     h2 = ((b-a)/(n*2.0));
42     res = ((h2*fun(a))+(h2*fun(b)));
43
44     FAST_FOR(i,0,n,1,0,1)
45         p = (a+((i*2)*h2));
46         res += ((2*h2)*fun(p));
47     END_FOR
48
49     return res;
50 }
51 } // module namespace
52
53 int main(int argc, char **argv) {
54     __shedskin__:__init();
55     __math__:__init();
56     __time__:__init();
57     __trapintloop__:__main();
```

5.4 Integração com a Linguagem Fortran

A linguagem **Fortran** é uma das mais antigas linguagens de programação ainda em uso. Desenvolvida nos anos 50 pela IBM, foi projetada especificamente para aplicações científicas. A sigla **Fortran** deriva de “IBM mathematical FORmula TRANslation system”. Dada a sua longa existência, existe uma grande quantidade de código científico escrito em **Fortran** disponível para uso.

CAPÍTULO 5. INTERAGINDO COM OUTRAS LINGUAGENS

118

Felizmente, a integração do **Fortran** com o Python pode ser feita de forma extremamente simples, através da ferramenta **f2py**, que demonstraremos a seguir.

f2py

Esta ferramenta está disponível como parte do Pacote **numpy** (www.scipy.org). Para ilustrar o uso do **f2py**, vamos voltar ao problema da integração pela regra trapezoidal (equação 5.3). Como vimos, a implementação deste algoritmo em Python, utilizando um laço **for**, é ineficiente. Para linguagens compiladas como C++ ou **Fortran**, laços são executados com grande eficiência. Vejamos a performance de uma implementação em **Fortran** da regra trapezoidal (listagem ??).

Listagem 5.16: implementação em **Fortran** da regra trapezoidal.
label

```
1  program trapezio
2    intrinsic exp, log, sin
3    real linear, nonl
4    external linear
5    external nonl
6    call tint(1.0,5.0,linear,res)
7    print *, "Resultado: ",res
8    call tint(1.0,5.0,nonl,res)
9    print *, "Resultado: ",res
10   end
11
12  subroutine tint(a,b,fun,res)
13    real a, b, res
14    integer*8 n
15    real h2, alfa
16    real fun
17    external fun
```

```
18      n = 5000000
19 Cf2py intent(in) a
20 Cf2py intent(in) b
21 Cf2py intent(in) fun
22 Cf2py intent(out,hide,cache) res
23     h2 = (b-a)/(2*n)
24     alfa=h2*fun(a)+h2*fun(b)
25     res=0.0
26 do i=1,n
27     p = a+i*2*h2
28     res = res+2*h2*fun(p)
29 end do
30 res = res+alfa
31 return
32 end
33
34 real function linear(x)
35 real x
36 Cf2py intent(in,out) x
37 linear = x+1.0
38 return
39 end
40
41 real function nonl(x)
42 real x
43 Cf2py intent(in,out) x
44 nonl = exp(x**2*log(x+x*sin(x)))
45 return
46 end
```

A listagem 5.17 nos mostra como compilar e executar o código da listagem ???. Este comando de compilação pressupõe que você possua o **GCC** (Gnu Compiler Collection) versão 4.0 ou superior. No caso de versões mais antigas deve-se substituir **gfortran** por

*CAPÍTULO 5. INTERAGINDO COM OUTRAS
LINGUAGENS*

120

g77 ou f77.

Listagem 5.17: Compilando e executando o programa da listagem ??

```
1 $ gfortran -o trapint trapint.f
2 $ time ./trapint
3 Resultado:    16.01428
4 Resultado:    48941.40
5
6 real      0m2.028 s
7 user      0m1.712 s
8 sys       0m0.013 s
```

Como em **Fortran** não temos a conveniência de uma função para “cronometrar” nossa função, utilizamos o comando **time** do Unix. Podemos constatar que o tempo de execução é similar ao obtido com a versão em **C++** (listagem 5.15).

Ainda que não seja estritamente necessário, é recomendável que o código **Fortran** seja preparado com comentários especiais (**Cf2py**), antes de ser processado e compilado pelo **f2py**. A listagem ?? já inclui estes comentários, para facilitar a nossa exposição. Estes comentários nos permitem informar ao **f2py** as variáveis de entrada e saída de cada função e algumas outras coisas. No exemplo ??, os principais parametros passados ao **f2py**, através das linhas de comentário **Cf2py intent()**, são **in**, **out**, **hide** e **cache**. As duas primeiras identificam as variáveis de entrada e saída da função ou procedure. O parâmetro **hide** faz com que a variável de saída **res**, obrigatoriamente declarada no cabeçalho da procedure em **Fortran** fique oculta ao ser importada no Python. O parâmetro **cache** reduz o custo da realocação de memória em variáveis que são redefinidas dentro de um laço em **Fortran**.

Antes que possamos “importar” nossas funções em **Fortran** para uso em um programa em Python, precisamos compilar nossos fontes

em Fortran com o `f2py`. A listagem 5.18 nos mostra como fazer isso.

Listagem 5.18: Compilando com `f2py`

```
1 $ f2py -m trapintf -c trapint.f
```

Uma vez compilados os fontes em Fortran com o `f2py`, podemos então escrever uma variação do script `trapintloop.py` (listagem 5.12) para verificar os ganhos de performance. A listagem 5.19 contém nosso script de teste.

Listagem 5.19: Script para comparação entre Python e Fortran via `f2py`

```
1 #-*-encoding: latin-1-*-
2 # Disponivel no pacote de programas como:
3 #   trapintloopcomp.py
4 from time import clock
5 from math import exp, log, sin
6 from trapintf import tint as ftint, linear,
7     nonl
8 def tint(a,b,f):
9     n = 5000000
10    h2 = (b-a)/(n*2.)
11    res = h2*f(a)+h2*f(b)
12    for i in xrange(n):
13        p = a+i*2*h2
14        res += 2*h2*f(p)
15    return res
16 # Rodando as funcoes em Python
17 start = clock()
18 r = tint(1,5,lambda x:1+x)
19 stop = clock()
20 print 'resultado: %d\nTempo: %s seg'%(r,stop
21      -start)
```

```
19 start = clock()
20 r = tint(1.,5.,lambda(x):exp(x**2*log(x+x*sin(
21     x)))))
21 stop = clock()
22 print 'resultado: %d\nTempo: %s seg'%(r,stop
23     -start)
#Rodando as funções em Fortran chamando
24 Python
24 print "tempo do Fortran com funcoes em
25     Python"
25 start = clock()
26 r = ftint(1.,5.,lambda(x):1+x)
27 print 'resultado: %d\nTempo: %s seg'%(r,
28     clock()-start)
28 start = clock()
29 r = ftint(1.,5.,lambda(x):exp(x**2*log(x+x*
30     sin(x))))
30 print 'resultado: %d\nTempo: %s seg'%(r,
31     clock()-start)
#Rodando as funções em Fortran puro
32 print "tempo do Fortran com funcoes em
33     Fortran"
33 start = clock()
34 r = ftint(1.,5.,linear)
35 print 'resultado: %d\nTempo: %s seg'%(r,
36     clock()-start)
36 start = clock()
37 r = ftint(1.,5.,nonl)
38 print 'resultado: %d\nTempo: %s seg'%(r,
39     clock()-start)
```

A listagem 5.19 contem uma versão da regra trapezoidal em Python puro e importa a função `tint` do nosso programa em Fortran. A função em Fortran é chamado de duas formas: uma para inte-

grar funções implementadas em Python (na forma funções `lambda`) e outra substituindo as funções `lambda` pelos seus equivalentes em `Fortran`.

Executando `trapintloopcomp.py`, podemos avaliar os ganhos em performance (listagem 5.20). Em ambas as formas de utilização da função `ftint`, existem chamadas para objetos Python dentro do laço `DO`. Vem daí a degradação da performance, em relação à execução do programa em `Fortran`, puramente.

Listagem 5.20: Executando `trapintloopcomp.py`

```
1 $ python trapintloopcomp.py
2 resultado: 16
3 Tempo: 13.29 seg
4 resultado: 49029
5 Tempo: 29.14 seg
6 tempo do Fortran com funcoes em Python
7 resultado: 16
8 Tempo: 7.31 seg
9 resultado: 48941
10 Tempo: 24.95 seg
11 tempo do Fortran com funcoes em Fortran
12 resultado: 16
13 Tempo: 4.85 seg
14 resultado: 48941
15 Tempo: 6.42 seg
```

Neste ponto, devemos parar e fazer uma reflexão. Será justo comparar a pior implementação possível em Python (utilizando laços `for`), com códigos compilados em `C++` e `Fortran`? Realmente, não é justo. Vejamos como se sai uma implementação competente da regra trapezoidal em Python (com uma ajudinha do pacote `numpy`). Consideraremos a listagem 5.21.

Listagem 5.21: Implementação vetorizada da regra trapezoidal

```
1 #-*-encoding: latin-1-*-
2 # Disponível no pacote de programas como:
3 # trapintvect.pt
4 from numpy import *
5 from time import clock
6
7 def tint(a,b,fun):
8     """
9         Integração numérica pela regra
10            trapezoidal
11    """
12    n = 5000000.
13    h = (b-a)/n
14    res = h/2*fun(a)+h/2*fun(b)+h*sum(
15        fun(arange(a,b,h)))
16    return res
17
18 if __name__=='__main__':
19     start = clock()
20     r = tint(1,5,lambda x:1+x)
21     print 'resultado: %d\nTempo: %s seg' %
22         (r,clock()-start)
23     start = clock()
24     r = tint(1,5,lambda x:exp(x**2*log(
25         x+x*sin(x))))
26     print 'resultado: %d\nTempo: %s seg' %
27         (r,clock()-start)
```

Executando a listagem 5.21, vemos que a implementação vetorializada em Python ganha (0.28 e 2.57 segundos) de nossas soluções utilizando f2py.

Da mesma forma que com o Pyrex, podemos distribuir programas escritos em Python com extensões escritas em Fortran, com a ajuda do pacote setuptools. Na listagem 5.22 vemos o exem-

exemplo de como escrever um `setup.py` para este fim. Neste exemplo, temos um `setup.py` extremamente limitado, contendo apenas os parâmetros necessários para a compilação de uma extensão denominada `flib`, a partir de uma programa em `Fortran`, localizado no arquivo `flib.f`, dentro do pacote “`meupacote`”. Observe, que ao definir módulos de extensão através da função `Extension`, podemos passar também outros argumentos, tais como outras bibliotecas das quais nosso código dependa.

Listagem 5.22: `setup.py` para distribuir programas com extensões em `Fortran`

```
1 import ez_setup
2 ez_setup.use_setuptools()
3 import setuptools
4 from numpy.distutils.core import setup,
5     Extension
6 flib = Extension(name='meupacote.flib',
7                   libraries=[],
8                   library_dirs=[],
9                   f2py_options=[],
10                  sources=['meupacote/
11                            flib.f'])
12 setup(name = 'mypackage',
13       version = '0.3.5',
14       packages = [ 'meupacote' ],
15       ext_modules = [ flib ])
```

5.5 A Pítón que sabia Javanês — Integração com Java

Peo licença ao mestre Lima Barreto, para parodiar o título do seu excelente conto, pois não pude resistir à analogia. A linguagem Python, conforme descobrimos ao longo deste livro, é extremamente versátil, e deve esta versatilidade, em parte, à sua biblioteca padrão. Entretanto existe uma outra linguagem que excede em muito o Python (ao menos atualmente), na quantidade de módulos disponíveis para os mais variados fins: o Java.

A linguagem Java tem, todavia, contra si uma série de fatores: A complexidade de sua sintaxe rivaliza com a do C++, e não é eficiente, como esperaríamos que o fosse, uma linguagem compilada, com tipagem estática. Mas todos estes aspectos negativos não são suficientes para anular as vantagens do vasto número de bibliotecas disponíveis e da sua portabilidade.

Como poderíamos capturar o que o Java tem de bom, sem levar como “brinde” seus aspectos negativos? É aqui que entra o Jython.

O Jython é uma implementação completa do Python 2.2¹ em Java. Com o Jython programadores Java podem embutir o Python em seus aplicativos e applets e nós, programadores Python, podemos utilizar, livremente, toda (ou quase toda) a biblioteca padrão do Python com classes em Java. Além destas vantagens, O Jython também é uma linguagem Open Source, ou seja de código aberto.

O interpretador Jython

Para iniciar nossa aventura com o Jython, precisaremos instalá-lo, e ter instalada uma máquina virtual Java (ou JVM) versão 1.4 ou mais recente.

¹O desenvolvimento do Jython continua, mas não se sabe ainda quando alcançará o CPython (implementação em C do Python).

Vamos tentar usá-lo como usariam o interpretador Python e ver se notamos alguma diferença.

Listagem 5.23: Usando o interpretador Jython

```
1 $ jython
2 Jython 2.1 on java1.4.2-01 (JIT: null)
3 Type "copyright", "credits" or "license" for
   more information.
4 >>> print 'hello world'
5 hello world
6 >>> import math()
7 >>> dir(math)
8 [ 'acos', 'asin', 'atan', 'atan2', 'ceil', '
   classDictInit', 'cos', 'cosh', 'e', 'exp',
   'fabs', 'floor', 'fmod', 'frexp', '
   hypot', 'ldexp', 'log', 'log10', 'modf',
   'pi', 'pow', 'sin', 'sinh', 'sqrt', '
   tan', 'tanh' ]
9 >>> math.pi
10 3.141592653589793
```

Até agora, tudo parece funcionar muito bem. Vamos tentar um exemplo um pouco mais avançado e ver de que forma o Jython pode simplificar um programa em Java.

Listagem 5.24: Um programa simples em Java usando a classe Swing.

```
1 import javax.swing.JOptionPane;
2 class testDialog {
3     public static void main ( String[] args )
4     {
5         javax.swing.JOptionPane.
6             showMessageDialog ( null, "Isto é
7             um teste." );
```

```
5 }  
6 }
```

A versão apresentada na listagem 5.24 está escrita em **Java**. Vamos ver como ficaria a versão em **Jython**.

Listagem 5.25: Versão Jython do programa da listagem 5.24.

```
1 >>> import javax.swing.JOptionPane  
2 >>> javax.swing.JOptionPane.  
      showMessageDialog(None, "Isto é um teste.  
      ")
```

Podemos observar, na listagem 5.25, que eliminamos a verborragia característica do **Java**, e que o programa em **Jython** ficou bem mais “pitônico”. Outro detalhe que vale a pena comentar, é que não precisamos compilar (mas podemos se quisermos) o código em **Jython**, como seria necessário com o **Java**. Só isto já é uma grande vantagem do **Jython**. Em suma, utilizando-se o **Jython** ao invés do **Java**, ganha-se em produtividade duas vezes: Uma, ao escrever menos linhas de código e outra, ao não ter que recompilar o programa a cada vez que se introduz uma pequena modificação.

Para não deixar os leitores curiosos acerca da finalidade do código apresentado na listagem 5.25, seu resultado encontra-se na figura 5.3.

Criando “Applets” em **Jython**

Para os convidados de **Java**, o **Jython** pode ser utilizado para criar “servlets”, “beans” e “applets” com a mesma facilidade que criamos um aplicativo em **Jython**. Vamos ver um exemplo de “applet”(listagem 5.26).

Listagem 5.26: Criando um applet em **Jython**

```
1 | import java.applet.Applet;
```

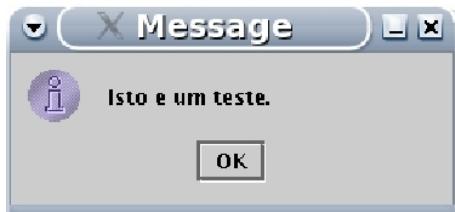


Figura 5.3: Saída da listagem 5.25.

```
2 class appletp ( java.applet.Applet ):
3     def paint ( self , g ):
4         g.drawString ( 'Eu sou um Applet
5             Jython!' , 5 , 5 )
```

Para quem não conhece Java, um applet é um mini-aplicativo feito para ser executado dentro de um Navegador (Mozilla, Opera etc.) que disponha de um “plug-in” para executar código em Java. Portanto, desta vez, precisaremos compilar o código da listagem 5.26 para que a máquina virtual Java possa executá-lo. Para isso, salvamos o código e utilizamos o compilador do Jython, o jythonc.

Listagem 5.27: Compilando appletp.py

```
1 $ jythonc --deep --core -j appletp.jar
2           appletp.py
3 processing appletp
4
5 Required packages:
6   java.applet
7
8 Creating adapters:
9 Creating .java files:
```

CAPÍTULO 5. INTERAGINDO COM OUTRAS
LINGUAGENS

130

```
10    appletp module
11        appletp extends java.applet.Applet
12
13 Compiling .java to .class...
14 Compiling with args: ['/opt/blackdown-jdk
15     -1.4.2.01/bin/javac', '-classpath', '/
16     usr/share/jython/lib/jython-2.1.jar:/usr
17     /share/libreadline-java/lib/libreadline-
18     java.jar:../jpywork:/usr/share/jython/
19     tools/jythonc:/home/fccoelho/Documents/
     LivroPython/../usr/share/jython/Lib',
     './jpywork/appletp.java']
0 Note: ./jpywork/appletp.java uses or
      overrides a deprecated API.
Note: Recompile with -deprecation for
      details.

Building archive: appletp.jar
Tracking java dependencies:
```

Uma vez compilado nosso applet, precisamos “embuti-lo” em um documento HTML (listagem 5.28). Então, basta apontar nosso navegador para este documento e veremos o applet ser executado.

Listagem 5.28: Documento HTML contendo o “applet”.

```
1 <html>
2 <head>
3     <meta content="text/html; charset=ISO
     -8859-1"
4     http-equiv="content-type">
5     <title>jython applet</title>
6 </head>
7 <body>
8 Este
```

```
9 &acute; o seu applet em Jython:<br>
10 <br>
11 <br>
12 <center>
13 <applet code="appletp" archive="appletp.jar"
14     name="Applet em Jython"
15     alt="This browser doesn't support JDK 1.1
16     applets." align="bottom"
17     height="50" width="160">
18 <PARAM NAME="codebase" VALUE=". ">
19 <h3>Algo saiu errado ao carregar
20 este applet.</h3>
21 </applet>
22 </center>
23 <br>
24 <br>
25 </body>
26 </html>
```

Na compilação, o código em Jython é convertido completamente em código Java e então compilado através do compilador Java padrão.

5.6 Exercícios

1. Compile a listagem 5.1 com o Shed-skin e veja se há ganho de performance. Antes de compilar, remova as linhas associadas ao uso do Weave.
2. Após executar a função primes (listagem 5.6), determine o tamanho da lista de números primos menor do que 1000. Em seguida modifique o código Pyrex, declarando a variável results como uma lista de inteiros, e eliminando a função

CAPÍTULO 5. INTERAGINDO COM OUTRAS
132 LINGUAGENS

`append` do laço `while`. Compare a performance desta nova versão com a da versão original.

Parte II

Aplicando o Python a Problemas Científicos Concretos

Capítulo 6

Modelagem Matemática

Introdução à modelagem matemática e sua implementação computacional. Discussão sobre a importância da Análise dimensional na modelagem e apresentação do pacote `Unum` para lidar com unidades em Python. Pré-requisitos: Conhecimentos básicos de cálculo.

A construção de modelos é uma etapa imprescindível, na cadeia de procedimentos normalmente associada ao método científico. Modelos matemáticos são uma tentativa de representar de forma lógico/quantitativa, mecanismos que observamos na natureza.

Todo modelo é por definição falso. O modelo é um instrumento de organização de idéias e hipóteses. Como tal, o modelo bom é aquele que representa bem nossas idéias. O que o fará útil para o conhecimento do mundo, dependerá, em grande parte, da qualidade de nossa concepção de mundo. Assim, bons modelos exigem uma boa compreensão do sistema a ser modelado. Alguém pode estar pensando: Se é preciso conhecer bem para modelar, então para que modelar, se o objetivo for conhecer bem? O grande segredo da modelagem é que o ato de modelar é, em si, um ato de conhecer. Isto é, para criar um modelo, o autor precisa escrutinar todo o seu conhecimento do sistema-alvo: identificar seus componentes,

discriminar o que é relevante do que não é, hipotetizar sobre as relações causais que regem os processos observados, etc.

Ao se deter nestes detalhes, começam a surgir os vazios do conhecimento, buracos a serem preenchidos com experimentos e observações ainda não pensados. Do mesmo modo, observações e intuições que antes não faziam muito sentido, começam a se encaixar melhor...

Muitas vezes, o resultado final de um modelo parece tão óbvio, que nos perguntamos para que serviu. No entanto, o óbvio só se tornou óbvio, por causa do ato de modelar.

“E aquilo que neste momento se revelará aos povos, não surpreenderá a todos por ser exótico, mas pelo fato de ter sempre estado oculto, quando terá sido, o óbvio”

Caetano Veloso

Outras vezes, os modelos abrem novas avenidas de escrutínio, ao apontar para fenômenos pouco intuitivos (como o caos determinístico).

Para matematizar um modelo, existem dois pontos de partida. Podemos construir do zero ou pegar emprestado. Pegar emprestado significa adaptar um modelo proposto por alguém. A vantagem de pegar emprestado é que o modelo já vem com alguma bagagem analítica. A desvantagem é que ele tolhe a nossa liberdade criativa. Ao olharmos para a história da modelagem, encontramos alguns modelos que ficaram de tal forma inseridos no imaginário do modelador que parecem ser os únicos existentes. Um exemplo é o modelo de Lotka-Volterra. Ele é tão usado, que as vezes se torna difícil escrever alguma coisa nova sem, inconscientemente, comparar o modelo novo com este padrão.

Modelos, embora falsos, devem ser corretos. Existem dois níveis de correção de um modelo matemático: modelos devem ter coerência lógica (isto é, não se pode somar alhos com bugalhos) e coerência com o modelo mental proposto (isto é, ele deve representar aquilo que o autor quer dizer).

Computacionalmente falando, modelos são como máquinas transformadoras, produzem informação a partir de informação. São equivalentes, portanto, a funções que, a partir de argumentos, produzem um resultado que consiste na transformação do argumento de entrada, de acordo com o mecanismo codificado dentro de si.

Vamos começar nossa jornada, em direção à representação computacional de modelos matemáticos, com uma breve introdução ao modelos mais simples e suas implementações.

6.1 Modelos

Lineares

Modelos lineares representam a relação entre grandezas que variam de forma proporcional. Um exemplo simples é o da corrida de taxi. O valor pago ao motorista numa corrida de taxi é calculado em função da quilometragem rodada, mais o valor da bandeirada inicial. Digamos que João cobre 0.50 reais por quilômetro rodado e uma bandeirada de 2,00. Se rodarmos 10km, pagaremos $0.5 \times 10 = 5$ reais (+ a bandeirada); se rodarmos o dobro (20km), pagaremos o dobro pela corrida (10 reais) + bandeirada. As duas grandezas - quilometragem rodada e valor pago pela quilometragem - variam de forma proporcional. Vamos supor que João tenha um caderno onde ele registra todas as corridas que faz, incluindo quilometragem, valor pago pela quilometragem e valor total pago (quilometragem + bandeirada).

Se fizermos um diagrama de dispersão com os dados da tabela 6.1, obteremos uma reta representando o cenário sem bandeirada e outra para o cenário com bandeirada.

O cálculo da corrida final é feito pela equação:

$$R\$ = a \times Km + b$$

Tabela 6.1: Valor de corridas de taxi

corrida	km	valor /kilometragem	valor total
1	5	2,50	4,50
2	8	4,00	6,00
3	3	1,50	3,50
4	10	5,00	7,00
5	9	4,50	6,50
6	4	2,00	4,00
7	15	7,50	9,50

onde a é o valor pago por quilômetro rodado ($a = 0,50$) e b é o valor inicial pago (isto é, a bandeirada). Em Python:

```

1 >>> def taxi(tarifa,km, band=2):
2         return tarifa*km+band
3 >>> pr = taxi(0.5,10)
4 >>> pr
5 7.0

```

Vamos ver outros exemplos:

Crescimento populacional por imigração constante. Uma população é composta inicialmente por 100 indivíduos. A partir do ano que chamaremos 0, passaram a chegar 10 novos indivíduos por ano. Esta população crescerá, isto é, terá 110 indivíduos no ano 1, 120 indivíduos no ano 2, 130 indivíduos no ano 3, etc. Se nós fizermos um gráfico da relação ano x número de indivíduos, teremos uma reta crescente. Esta reta intercepta o eixo vertical no ponto 100 (população inicial) e tem inclinação dada pelo número de indivíduos acrescidos por unidade de tempo (isto é, $a = 10$).

Decrescimento populacional por emigração constante. Agora, suponha que ao invés de entrar, estão saindo 10 indivíduos a cada ano. Se a população é composta inicialmente por 100 indivíduos, então teremos 90 indivíduos no ano 1; 80 indivíduos no ano 2; 70

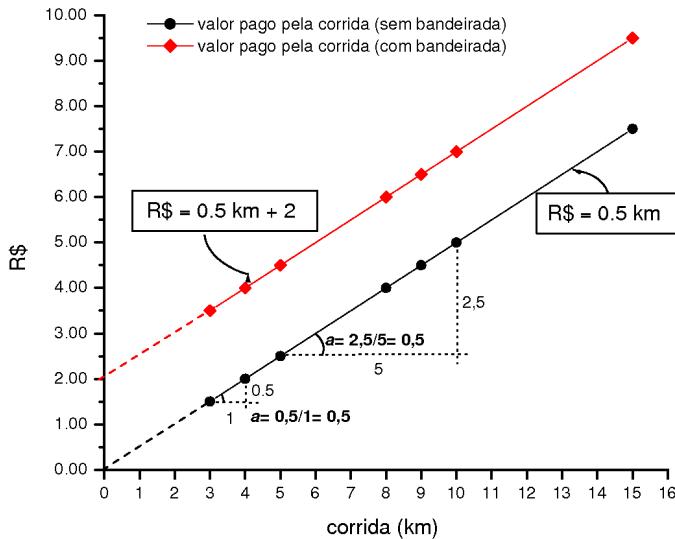


Figura 6.1: Diagrama de dispersão do exemplo do taxi. A reta superior descreve o cenário com bandeirada e a inferior, sem bandeirada.

indivíduos no ano 3, etc. Se nós fizermos um gráfico da relação entre ano e número de indivíduos, teremos uma reta decrescente. Esta reta intercepta o eixo vertical no ponto 100 (população inicial) e tem inclinação negativa, cujo valor absoluto é dado pelo número de indivíduos perdidos por unidade de tempo (isto é, $a = -10$).

Todos os exemplos dados são especificações de modelos cujo formato geral é:

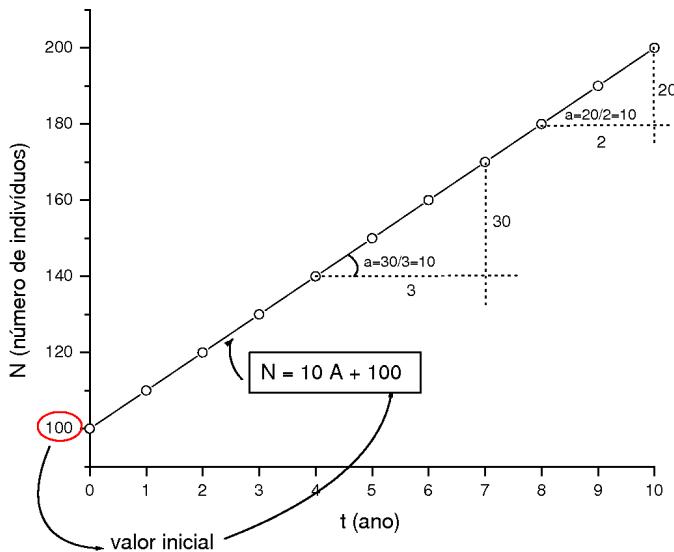


Figura 6.2: População crescendo de forma linear, devido à chegada de um número constante de novos indivíduos por ano.

$$Y = a \times X + b$$

onde a , chamado de coeficiente angular, é a taxa de variação da grandeza Y por unidade de X e b , chamado de coeficiente linear, é o valor de Y quando $X = 0$. Este modelo se expressa graficamente como uma reta com ângulo a em relação a linha horizontal e que cruza o eixo vertical no ponto b . Esta reta será crescente se a for maior do que 0; horizontal, se $a = 0$, e decrescente, se a for negativo.

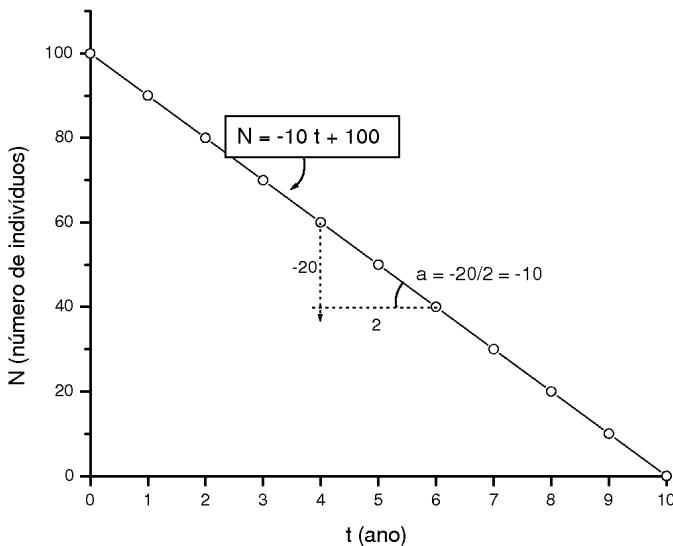


Figura 6.3: População diminui de forma linear, devido à saída de um número constante de indivíduos por ano.

Propriedades Uma propriedade importante do modelo linear é que a variação de crescimento ou decrescimento é constante e independente do valor de X. No caso da imigração, por exemplo, estavam entrando 10 pessoas todos os anos, independentemente do número de pessoas já existentes na população. No caso do taxi, o valor pago por quilômetro rodado não mudava se a pessoa já tivesse rodado 5 ou 20 km.

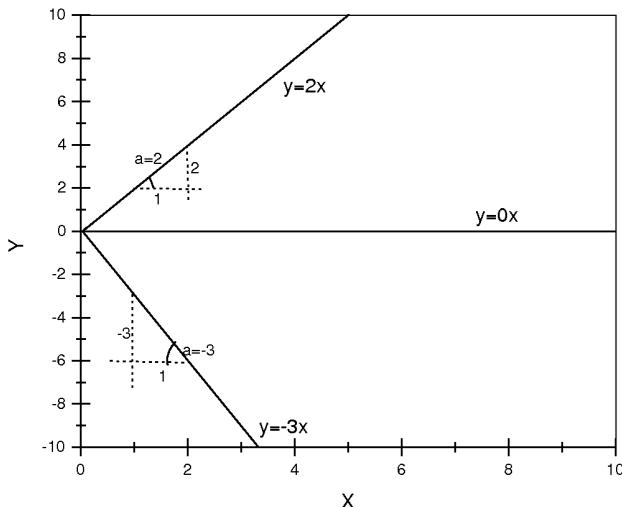


Figura 6.4: Representação gráfica de um modelo linear. Efeito do parâmetro a .

Exponenciais

O modelo linear descrito na seção anterior, pode ser um bom modelo para descrever o crescimento de uma população por imigração constante. Porém, não nos serve para descrever o crescimento por reprodução. Vejamos o clássico exemplo do crescimento bacteriano por bipartição. Digamos que temos uma população com uma bactéria no tempo $t=0$. Após 1 hora, cada bactéria se divide e forma duas bactérias-filha. Estas duas bactérias geram 4 novas bactérias

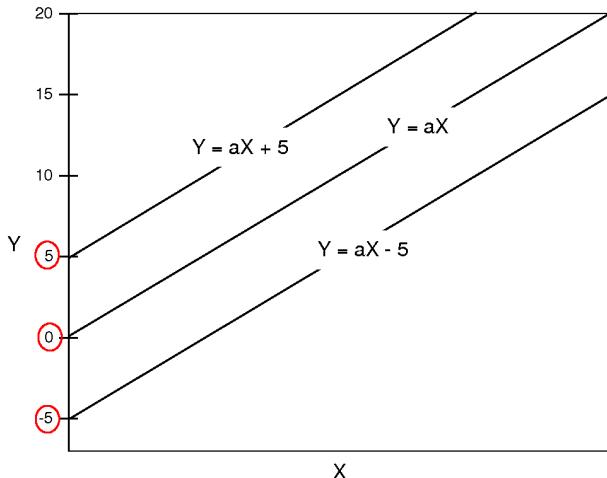


Figura 6.5: Representação gráfica de um modelo linear. Efeito do parâmetro b .

em mais uma hora e assim por diante. Se fizermos um gráfico do número de bactérias na população, teremos algo como a figura 6.6.

Veja que o número de novos indivíduos a cada geração não é constante, como no modelo linear. Agora, ele aumenta de forma geométrica. Na primeira geração foram 2 novos indivíduos, na segunda geração foram 4 e assim por diante. O crescimento é muito mais acelerado e parece tender para uma explosão demográfica! Em apenas 10 horas, teremos uma população com 1024 indivíduos e, em um dia, 16.677.216 bactérias!

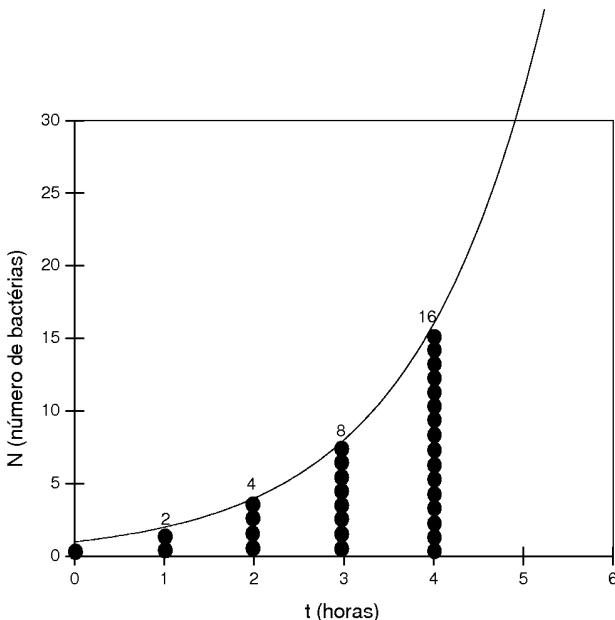


Figura 6.6: Crescimento de uma população de bactérias por bipartição.

No entanto, embora o crescimento não seja constante no tempo, ele é constante por indivíduo. O que isso quer dizer? Cada indivíduo na população contribui com o mesmo número de filhos para a geração seguinte, de forma que uma população pequena gerará um número menor de “filhotes” do que uma população grande.

Este tipo de crescimento dependente do número de indivíduos presentes é chamado de exponencial, pelo fato da variável indepen-

dente estar no expoente. Matematicamente, ele tem o formato:

$$Y = wB^X$$

onde w é o número inicial de bactérias ($w = 1$).

Ao contrário do modelo linear, onde a taxa de variação era independente do número de indivíduos presentes, no modelo exponencial, a taxa de variação é uma função, isto é, depende, do número de indivíduos presentes.

6.2 Construindo Modelos Dinâmicos

Na seção anterior vimos como representar fenômenos simples como expressões matemáticas. Agora vamos explorar representações mais sofisticadas: Vamos modelar processos dinâmicos. Processos dinâmicos modificam-se em função de uma variável independente que, nos casos mais comuns, pode ser o tempo ou o espaço.

Nos modelos estáticos da seção anterior, buscamos expressar o estado do sistema em função de outras variáveis. Uma vez determinado os valor destas variáveis, o estado do sistema permanece inalterado. Agora, não sabemos o que determina o estado do sistema, necessariamente, mas podemos descrever mecanismos que alteram um estado inicial dado.

Neste tipo de modelagem utilizamos equações de diferença (variável independente discreta) ou diferenciais (variável independente contínua), para modelar a taxa de variação do sistema, representada pelas chamadas variáveis de estado.

Modelando Iterativamente

Vamos começar a explorar os modelos dinâmicos, através de modelos com dinâmica temporal discreta. Neste caso, podemos representar o tempo discreto como a execução iterativa de uma função.

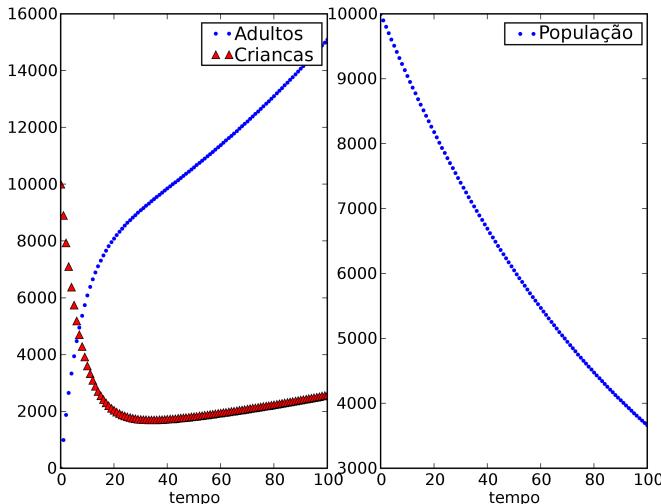


Figura 6.7: Modelos populacionais discretos. Modelo 2 à esquerda e modelo 1 à direita.

Na listagem 6.1, vemos dois modelos populacionais:

$$\text{Modelo 1 : } N_{t+1} = N_t + rN_t - mN_t + i - eN_t$$

$$\text{Modelo 2 : } \begin{cases} C_{t+1} &= C_t + r_a A_t - m_c C_t - e C_t \\ A_{t+1} &= A_t + e C_t - m_a A_t \end{cases}$$

No modelo 1, temos o tamanho da população no tempo seguinte (N_{t+1}) sendo expresso como uma alteração da população atual pelo nascimento de indivíduos a uma taxa r , pela mortalidade (m), e por fluxos migratórios (eN_t e i).

No modelo 2, temos uma demografia similar, porém agora com a população dividida em dois grupos: crianças (C_t) e adultos (A_t).

Nestes modelos, utilizamos um simples laço `for` para resolver numericamente as equações de diferença. Representações recursivas também seriam possíveis, mas para este exemplo não são necessárias.

Listagem 6.1: Dois modelos discretos de crescimento populacional

```
1 # Disponivel no pacote de programas como:  
2     modelo.py  
3 #-*-encoding: latin-1-*  
4 from pylab import *  
5  
6 N = [10000] #serie temporal de populacao  
7 n0 = 10000 #populacao inicial  
8 t = 100      #periodo de tempo da  
9      simulacao  
10 r = 0.02     #natalidade  
11 m = 0.03     #taxa de mortalidade  
12 im = 0       #taxa de imigração  
13 em = 0       #taxa de emigração  
14 #parâmetros modelo2  
15 mc = 0.01    #taxa de mortalidade  
16      infantil  
17 ma = 0.01    #taxa de mortalidade adulta  
18 ra = 0.02    #taxa de reproducao  
19 e = 0.1      #taxa de envelhecimento  
20 C = [10000]   #serie temporal de crianças  
21 A = [0]        #serie temporal de adultos  
22 a0 = 0        #populacao inicial de  
23      adultos  
24 c0 = 10000    #populacao inicial de  
25      criancas  
26  
27 def Modelo(n0):  
28     nf = n0 + r*n0 -m*n0 +im -em*n0
```

```

24     N.append( nf )
25     return nf
26
27 def Modelo2(c0 ,a0):
28     cf = c0 + ra*a0 -mc*c0 -e*c0
29     af = a0 + e*c0 - ma*a0
30     C.append( cf )
31     A.append( af )
32     return cf ,af
33
34 for i in xrange(t):
35     c0 ,a0 = Modelo2(c0 ,a0)
36     n0 = Modelo(n0)
37
38 subplot(121)
39 plot(A, ' . ', C, 'r^')
40 legend(['Adultos', 'Criancas'])
41 xlabel('tempo')
42 subplot(122)
43 plot(N, ' . ')
44 legend(['População'])
45 xlabel('tempo')
46 savefig('pop.png', dpi=400)
47 show()

```

Integração Numérica

A simulação de modelos onde a variável independente é contínua, é tecnicamente mais complexa. Por isso, neste caso, lançaremos mão do pacote `scipy`¹.

¹<http://www.scipy.org>

Modelos dinâmicos contínuos são comumente representados por sistemas de equações diferenciais ordinárias, e sua resolução envolve a integração destas numericamente.

O pacote `scipy` contém um pacote dedicado à integração numérica de equações diferenciais: o `integrate`.

1 In [1]:
`from scipy import integrate`

Nesta seção utilizaremos o pacote `integrate` para resolver um sistema de equações diferenciais ordinárias bem simples:

$$\begin{aligned}\frac{dP}{dt} &= rP - pPQ \\ \frac{dQ}{dt} &= -mQ + PQ\end{aligned}\tag{6.1}$$

O sistema de equações 6.2, representa a dinâmica de duas populações, P e Q onde Q (predador) alimenta-se de P (presa). Em nossa implementação, utilizaremos a função `odeint` do pacote `integrate`². Esta função tem como argumentos uma função que retorna as derivadas, a partir das condições iniciais, e um vetor de valores da variável independente(t) para os quais os valores das variáveis de estado serão calculados.

Listagem 6.2: Integrando um sistema de equações diferenciais ordinárias

```
1 #modelo_ode.py
2 # Disponivel no pacote de programas como:
3 #     modelo_ode.py
4 #-*encoding: utf-8-*#
4 from scipy import integrate
```

²A função `odeint` encapsula a tradicional biblioteca `lsoda` implementada em Fortran. Esta biblioteca determina dinamicamente o algoritmo de integração mais adequado a cada passo de integração, selecionando entre o método de Adams e o BDF.

```
5  from numpy import *
6  import pylab as P
7
8  class Model:
9      def __init__(self, equations, inits,
10                  trange,):
11          """
12              Equations: Lista com as equações
13                  diferenciais na forma de strings
14              inits: sequencia de condições
15                  iniciais
16              trange: extensão da simulação
17          """
18          self.eqs = equations
19
20          self.Inits = inits
21          self.Trange = arange(0, trange, 0.1)
22          self.compileEqs()
23
24
25      def compileEqs(self):
26          """
27              Compila as equações.
28          """
29          try:
30              self.ceqs = [compile(i, '<
31                          equation>', 'eval') for i in
32                          self.eqs]
33          except SyntaxError:
34              print 'Há um erro de sintaxe nas
35                  equações,\nConserte-o e
36                  tente novamente'
37
38      def Run(self):
39          """
40
```

```
32     Faz a integração numérica.  
33     """  
34     t_courseList = []  
35     t_courseList.append(integrate.odeint  
36         (self.Equations, self.Inits, self.  
37          Trange))  
38     return (t_courseList, self.Trange)  
39  
40  
41  
42  
43     #—Cria vetor de equações  
44  
45     eqs = self.ceqs  
46     Neq=len(eqs)  
47     ydot = zeros((Neq), 'd')  
48     for k in xrange(Neq):  
49         ydot[k] = eval(eqs[k])  
50     return ydot  
51  
52 if __name__=="__main__":  
53     inits = [1,1]  
54     eqs = [ '3.0*y[0]-2.0*y[0]*y[1]' , '-2.0*y  
55             [1]+y[0]*y[1]' ]  
56     ODE = Model(eqs, inits, 10)  
57     y, t = ODE.Run()  
58     #print y  
59     P.plot(t,y[0][:,0], 'v-', t,y[0][:,1], '  
60             -')  
61     P.legend(['Presa', 'Predador'])
```

```
59     P.savefig('lv.png', dpi=400)
60     P.show()
```

O modelo é implementado como uma classe. A implementação poderia ser mais simples, até mesmo feita diretamente do console do Python. Entretanto, esta implementação mais longa nos permite apresentar algumas outras técnicas interessantes.

O modelo é definido durante a inicialização do objeto, através de atributos obrigatórios. Um método `Run()` é utilizado para executar a integração numérica. É neste método que chamamos a função `odeint`, passando os argumentos requeridos, a saber: a função que implementa as equações, os valores iniciais das variáveis de estado e o vetor de tempos.

A equações são passadas pelo usuário na forma de strings. Estas strings devem conter uma expressão Python válida.

O método `Equations` precisa combinar as expressões das equações para poder calcular o valor das derivadas em cada ponto do tempo. Por isso, uma notação de lista foi utilizada para permitir referências entre as equações. Desta forma, cada variável de estado é um elemento da lista `y`.

As expressões representando as equações, são interpretadas por meio da função `eval`, a cada passo da integração numérica. Isto requer que as strings contendo as expressões, sejam compiladas para byte-code, e então interpretadas pelo Python. Para evitar que esta operação seja repetida a cada passo, o método `compileEqs` pré-compila estas expressões, o que acelera em muito a execução do programa 6.2.

Ao final da listagem 6.2, vemos a especificação do modelo e, na figura 6.8, o seu resultado.

O pacote `integrate` também nos oferece uma outra interface para a integração numérica chamada `ode`. A classe `ode` é uma alternativa orientada a objetos, à função `odeint`. A listagem 6.3, mostra como implementar o mesmo sistema de equações (6.2) utilizando a classe `ode`.

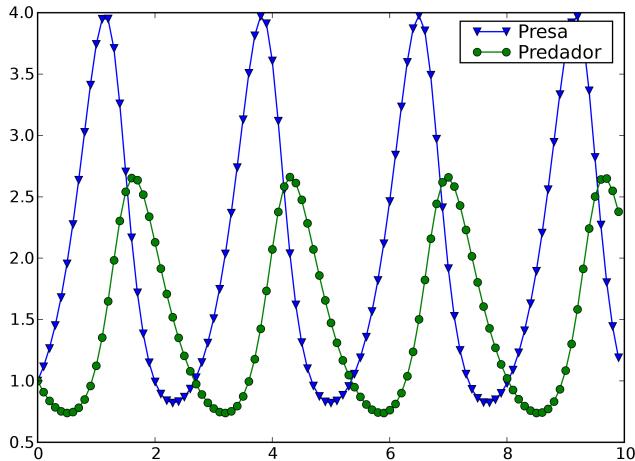


Figura 6.8: Gráfico com o resultado da integração do modelo 6.2.

Listagem 6.3: Integração numérica utilizando a classe ODE.

```

1 # Disponivel no pacote de programas como:
2     modelo_ode_minimo.py
3 #-*encoding: utf-8*-
4 from scipy import integrate
5 from numpy import *
6 import pylab as P
7
8 def Equations(t ,y):
9     """
10        Define os sistema de EDOs calculando
11            cada equacao
12        com base no valor anterior e retornando

```

```

    as derivadas .
11 #####
12 #—Cria vetor de equacoes
13 eqs = [ '3.0*y[0]-2.0*y[0]*y[1]' , '-2.0*y
14           [1]+y[0]*y[1]' ]
15 Neq=len(eqs)
16 ydot = zeros((Neq) , 'd')
17 for k in xrange(Neq):
18     ydot[k] = eval(eqs[k])
19 return ydot
20
21 if __name__=="__main__":
22     r = integrate.ode(Equations).
23         set_integrator('vode')
24     r.set_initial_value([1,1],0)
25     tf = 10 #tempo final
26     dt = 0.1
27     t = []
28     y = []
29     while r.successful() and r.t <= tf:
30         r.integrate(r.t+dt)
31         t.append(r.t)
32         y.append(r.y)
33     print r.y
34     #Plota resultados
35     P.plot(t, array(y)[:,0], 'v-',
36             array(y)[:,1], 'o-')
37     P.title('Integrador Vode')
38     P.legend(['Presa', 'Predador'])
39     P.savefig('lv2.png', dpi=400)
40     P.show()

```

Se você necessita construir, simular e analisar modelos de equa-

ções diferenciais com freqüência, talvez prefira utilizar um software mais completo, desenvolvido por mim, chamado Model-Builder³.

6.3 Grandezas, Dimensões e Unidades

Ao se construir modelos matemáticos, cedo ou tarde nos deparamos com a necessidade de aproximar o modelo conceitual do sistema real para dar valor preditivo ao modelo. Este processo envolve o mapeamento da métrica do sistema real no modelo matemático, ou seja, fazer com que os parâmetros ou constantes do nosso modelo correspondam, numericamente, a mensurações efetuadas no sistema real.

Este processo de parametrização envolve, frequentemente, a utilização de medidas das diferentes *dimensões* do sistema real (massa, comprimento, temperatura, etc). Estas mensurações raramente são efetuadas pelo modelador. Este, frequentemente, se utiliza de medidas publicadas na literatura científica, expressas nas mais variadas *unidades* de medida. A escolha das unidades de medida não altera as dimensões do objeto de estudo (medir a nossa mesa de trabalho em milímetros ao invés de metros não a torna mais longa), mas altera a ordem de *grandeza* do valor numérico que representa a dimensão medida.

Os modelos matemáticos relacionam as várias dimensões do sistema estudado, por meio das grandezas numéricas que as representam. Por isso, é fundamental que as variáveis e parâmetros do nosso modelo que representem uma mesma dimensão do objeto, sejam expressas nas mesmas unidades de medida. Por exemplo, se desenvolvemos um modelo que pretende correlacionar massa do coração com massa corporal, não podemos expressar a massa do coração em gramas(g) e a massa corporal em quilogramas(kg)⁴.

³<http://model-builder.sourceforge.net>

⁴A menos que adicionemos contantes ao modelo que re-equilibrem a equação do ponto de vista dimensional

Tabela 6.2: Dimensões básicas.

Grandezas físicas	Dimensão	Unidade (SI)	Símbolo
Comprimento	L	metro	m
Massa	M	quilograma	kg
Tempo	T	segundo	s
Temperatura	θ	kelvin	K
Quantidade de uma substância	N	mol	mol
Corrente elétrica	I	ampére	A
Intensidade luminosa	J	candela	cd

Grandeza, dimensão e unidade não são sinônimos e devido à ubiquidade da sua utilização em modelagem, são conceitos que devem estar muito bem definidos na mente de todo modelador.

Unidades derivadas de nomes próprios (e.g. Lord Kelvin, tabela 6.2), têm seu símbolo maiúsculo, mas o nome da unidade é grafado em minúsculas.

As dimensões de variáveis e parâmetros utilizados em um modelo podem representar dimensões físicas diretamente mensuráveis do sistema, como massa, comprimento, etc. ou podem ser construídos pelo investigador, que visam representar propriedades do sistema que interessam ao modelo (preço, periculosidade, fecundidade etc.). Devido a isso, antes de começar um trabalho de modelagem, devemos definir quais são as nossas *dimensões básicas*. Estas dimensões são escolhidas de forma a que todas as outras dimensões utilizadas no modelo, possam ser expressas em termos destas dimensões básicas, ou seja, através de operações algébricas entre as dimensões básicas. A tabela 6.2 lista algumas dimensões básicas que podem ser úteis no contexto da modelagem de sistemas biológicos.

Várias dimensões *derivadas* podem ser construídas a partir do limitado conjunto de dimensões básicas apresentado na tabela 6.2.

Tabela 6.3: Dimensões derivadas.

Grandeza física	Dimensão (derivação)	Unidade (SI)	Símbolo
Área	L^2	metro ²	m^2
Volume	L^3	metro ³	m^3
Velocidade	LT^{-1}	metro/segundo	m/s
Aceleração	LT^{-2}	metro/segundo ²	m/s^2
Força	MLT^{-2}	newton	$N (kg\ m\ s^{-2})$
Energia	ML^2T^{-2}	joule	$J (kg\ m^2\ s^{-2})$
Potência	ML^2T^{-3}	watt	$W (kg\ m^2\ s^{-3})$
Pressão	$ML^{-1}T^{-2}$	pascal	$Pa (Nm^{-2})$
Viscosidade	$ML^{-1}T^{-1} \times 10^{-1}$	poise	P
Voltagem	$ML^2T^{-3}I^{-1}$	volt	V
Resistência elét.	$ML^2T^{-3}I^{-2}$	ohm	$\Omega (V/A)$
Ângulo	—	graus, radianos	graus, radianos
Veloc. angular	T^{-1}	radianos/seg.	radianos/s
Frequênciā	T^{-1}	hertz	$Hz (s^{-1})$

A tabela 6.3 lista dimensões familiares ao leitor e suas derivações.

É interessante notar que algumas grandezas físicas são adimensionais (e.g. ângulo, tabela 6.3). Este fato não as impede, entretanto, de possuir unidades.

Um ângulo é calculado como a razão entre um dado arco e o raio da circunferência que o define. Portanto, sua dimensão é $L/L = 1$.

O fato de um ângulo ser adimensional, faz com que seu valor numérico não se altere, ao se utilizar diferentes unidades de medida para o raio e o arco da circunferência que o contém. Funções trigonométricas são adimensionais pois o argumento destas é um ângulo, que é adimensional. O expoente de uma função exponencial nada mais é do que um logaritmo, e se os logaritmos são adimensionais,

os expoentes também o serão.

Outros adimensionais:

Funções trigonométricas,

Expoentes,

Logaritmos,

Contagens,

π , e , etc.

Por exemplo, em uma equação de crescimento exponencial $y = e^{kt}$, onde t é o tempo, o parâmetro k deve ter dimensão T^{-1} para que o expoente seja adimensional.

Análise Dimensional

Um modo eficiente de evitar erros dimensionais na construção das equações que compõem o nosso modelo, consiste em construir uma versão da equação em questão, substituindo-se as variáveis e parâmetros por suas unidades ou dimensões. Em seguida, expande-se as expressões em ambos os lados da igualdade em termos das dimensões básicas. Então simplificamos ao máximo os dois lados da equação e verificamos se são de fato iguais. A este processo dá-se o nome de análise dimensional.

Todas as equações científicas devem estar dimensionalmente corretas, assim sendo, a análise dimensional é uma excelente ferramenta para analisar nossas formulações. Vamos listar algumas regras básicas da análise dimensional:

1. Dimensões e unidades podem ser combinadas e manipuladas, utilizando-se regras algébricas.
2. Os dois lados de uma equação devem ser dimensionalmente idênticos.
3. Deve-se tomar cuidado para não cancelar unidades idênticas de objetos independentes. Por exemplo, $\frac{ml\ CO_2}{ml\ sangue}$ não é uma

expressão adimensional, apesar de ser uma razão entre dois volumes.

4. Grandezas de dimensões diferentes não podem ser somadas nem subtraídas.

Unidades também podem ser utilizadas, diretamente, em análise dimensional. Unidades derivadas podem ser construídas a partir de unidades básicas. Porém, frequentemente, as unidades derivadas recebem símbolos especiais para simplificar sua notação (e.g. N , W , J etc. na tabela 6.3).

O Pacote Unum

⁵ Quando escrevemos nossos modelos em Python, podemos fazer uso de unidades para nossas variáveis de forma a não cometer erros quanto às dimensões do nosso problema. Vamos explorar algumas possibilidades interativamente dentro do Ipython.

```
1 In [1]: from unum.units import *
2 In [2]: M
3 Out[2]: 1.0 [m]
```

Uma vez importadas as unidades do módulo units, podemos representar números com unidades. No exemplo acima temos a representação de 1 metro. As classes que representam as unidades são derivadas das classes que representam os números puros em Python; portanto, todas as operações aritméticas válidas para números, são possíveis para números com unidades. O Unum vem com muitas unidades pré definidas, mas estas podem ser redefinidas, durante a sessão, assim como novas unidades podem ser definidas. Todas as unidades prédefinidas são definidas em maiúsculas para facilitar a sua diferenciação de nomes associados a variáveis.

⁵O pacote Unum pode ser baixado de <http://home.scarlet.be/be052320/Unum.html>.

Qualquer grandeza pode ser definida multiplicando-se um número por uma unidade.

```

1 In [3]: a=3*M
2 In [4]: t=1*S
3 In [5]: a/t
4 Out[5]: 3.0 [m/s]
5 In [6]: 1/(3*M/S)
6 Out[6]: 0.33333333333333 [s/m]
7 In [7]: 25*M**2
8 Out[7]: 25.0 [m2]
9 In [8]: (3*M)*(4*M)
10 Out[8]: 12.0 [m2]
11 In [9]: 13*KG*M/S**2
12 Out[9]: 13.0 [kg.m/s2]
```

Grandeza adimensionais também podem ser representadas.

```

1 In [10]: (2 * M/S) * (3 * S/M)
2 Out[10]: 6.0 []
```

Agora que já aprendemos a operar com unidades, vamos implementar um simples modelo com unidades.

```

1 In [11]: massa=1.5*KG
2 In [12]: velocidade=2*M/S
3 In [13]: energia_cinetica=(massa*velocidade
           **2)/2
4 In [14]: energia_cinetica
5 Out[14]: 3.0 [kg.m2/s2]
```

Evitando Erros

Utilizar unidades não é apenas uma questão estética, também pode ajudar a evitar enganos:

```

1 In [15]: energia_cinetica+3*KG
2 DimensionError: [kg.m2/s2] incompatible with
   [kg]
3 In [16]:M**KG
4 DimensionError: unit [kg] unexpected

```

No exemplo acima, somos lembados de que não podemos somar grandezas com unidades diferentes, nem elevar metros a quilogramas! Expoentes devem sempre ser adimensionais.

Também podemos fazer conversões de unidades. Vamos converter nossa energia cinética em joules:

```

1 In [17]: energia_cinetica . as (J)
2 Out[17]:3.0 [J]

```

Integração com Funções Matemáticas

Já vimos que logaritmos e ângulos são adimensionais; portanto funções logarítmicas e trigonométricas não aceitam números com unidades. No caso dos ângulos podemos usar pseudo-unidades como RAD(radianos) e ARCDEG(graus).

```

1 In [18]:from math import pi ,log10 , sin ,cos
2 In [19]:log10 (M/ANGSTROM)
3 Out[19]:10.0
4 In [20]:cos (180*ARCDEG)
5 Out[20]:-1.0
6 In [24]:cos (pi*RAD)
7 Out[24]:-1.0
8 In [25]:f = 440*HZ
9 In [26]:sin (f)
10 DimensionError: unit [Hz] unexpected
11 In [27]:dt = 0.1 * S
12 In [28]:sin (f*dt*2*pi)
13 Out[28]:-3.9198245344040927e-14

```

As unidade do `Unum` também funcionam bem com outras estruturas de dados, como matrizes do Numpy, por exemplo.

```

1 In [1]: from unum.units import *
2 In [2]: from numpy import *
3 In [3]: a= arange(10)*M
4 In [4]: a
5 Out[4]:
6 array([0.0 [m], 1.0 [m], 2.0 [m], 3.0 [m],
      4.0 [m], 5.0 [m], 6.0 [m],
7      7.0 [m], 8.0 [m], 9.0 [m]], dtype=
      object)
8 In [5]: a**2
9 Out[5]:
10 array([0.0 [m2], 1.0 [m2], 4.0 [m2], 9.0 [m2],
      16.0 [m2], 25.0 [m2],
11      36.0 [m2], 49.0 [m2], 64.0 [m2], 81.0
      [m2]], dtype=object)
```

Por fim, o `Unum` também simplifica automaticamente expressões de unidades. No exemplo abaixo, o `Unum` sabe que uma pressão multiplicada por uma área é uma força. Portanto, converte a unidade para Newtons (N).

```

1 In [6]: forca = PA * M**2
2 In [7]: forca
3 Out[7]: 1.0 [N]
```

Conforme já mencionamos, também podemos criar novas unidades. Vejamos abaixo como fazer isso:

Listagem 6.4: Criando novas unidades.

```

1 In [1]: from unum import Unum
2 In [2]: unit = Unum.unit
3 In [3]: LEGUA = unit('legua')
4 In [4]: LEGUA
```

```

5 Out[4]: 1.0 [legua]
6 In [5]: KLEGUA = unit('kilolegua', 1000*LEGUA)
7 In [6]: 20*KLEGUA+1*LEGUA
8 Out[6]: 20.001 [kilolegua]

```

Uma Aplicação Concreta

Muitas vezes, ao modelar um sistema biológico, a relação entre grandezas biológicas e físicas não é óbvia. Nestes casos, a análise dimensional pode ajudar a validar nossas formulações. Suponha que um pesquisador deseje determinar o trabalho realizado pelo coração. Ele se lembra da física que:

$$Trabalho(J) = Forca(N) \times distancia(m)$$

Esta equação não é diretamente aplicável ao coração, pois este não se desloca e a força que exerce não é facilmente mensurável. Contudo, a força exercida pelo coração se reflete na pressão sanguínea, e o volume de sangue bombeado parece ser análogo à distância da fórmula original. Então nosso pesquisador-modelador escreve, timidamente, a seguinte fórmula:

$$Trabalho = Pressão \times Volume$$

Mas como saber se esta relação é correta? Fazendo a análise dimensional.

$$Trabalho(J) = Pressão \times Volume = \frac{N}{m^2} \times m^3 = N \cdot m = J$$

Esta definição de Trabalho ($N \cdot m$), não é a mesma da tabela 6.3 mas um pouco de álgebra pode demonstrar a sua validade⁶. Com o Unum, isso seria uma tarefa trivial:

⁶Lembre-se de que Trabalho=Energia.

```

1 In [8]: J == PA * M**3
2 Out[8]: True

```

Para que uma análise dimensional funcione com unidades, devemos nos manter em um mesmo sistema de unidades. O sistema utilizado em ciência é o chamado Sistema Internacional (SI). As unidades apresentadas nas tabelas 6.2 e 6.3 estão de acordo com o SI.

Um detalhe em relação a unidades que é frequentemente mal entendido, é a aplicação de fatores de escala normalmente expressos como prefixos às unidades (ver tabela 6.4). Devido a este fato o SI é frequentemente denominado sistema MKS (metro, kilograma, segundo). Um outro sistema que frequentemente é confundido com o SI é o CGS (centímetro, grama, segundo). Apesar deste último se diferenciar do SI apenas nos prefixos, é um sistema completamente diferente.

MKS vs. CGS

Força(MKS): Newton ($m \cdot kg/s^2$)	Força(CGS): Dyne ($cm \cdot g/s^2$)
--	--

Ao adicionar ou modificar prefixos de unidades estamos abandonando nosso sistema de unidades original e nossas equações não estarão mais equilibradas dimensionalmente, a menos que realizemos as correções necessárias nos demais termos da equação.

Para re-equilibrar dimensionalmente equações cujas unidades de uma ou mais de suas variáveis tenham sido modificadas, precisamos incluir fatores de correção.

Consideremos a relação entre a massa corporal e a massa do Coração:

$$Y = 0.006M$$

Tabela 6.4: Prefixos de Unidades

Prefixo	Símbolo	Fator	Prefixo	Símbolo	Fator
yocto	y	10^{-24}	deca	da	10
zepto	z	10^{-21}	hecto	h	10^2
atto	a	10^{-18}	kilo	k	10^3
femto	f	10^{-15}	mega	M	10^6
pico	p	10^{-12}	giga	G	10^9
nano	n	10^{-9}	tera	T	10^{12}
micro	μ	10^{-6}	peta	P	10^{15}
mili	m	10^{-3}	exa	E	10^{18}
centi	c	10^{-2}	zetta	Z	10^{21}
deci	d	10^{-1}	yotta	Y	10^{24}

colocando esta equação em termos dimensionais:

$$M = 1 \cdot M$$

se quisermos expressar a massa do coração em gramas (CGS), mantendo a massa corporal em kg (SI/MKS), basta apenas multiplicarmos o lado direito da equação por 1000 para reequilibrar a equação. Assim, teríamos:

$$Y = 6M$$

Contudo, nem sempre a determinação dos fatores de correção é tão óvia. Vejamos a seguinte equação, que relaciona a massa do fígado com a massa corporal (ambas em kg):

$$M_f = 0.082 \cdot M_c^{0.87} \quad (6.2)$$

Esta relação se mantém, para mamíferos, ao longo de várias ordens de magnitude de tamanho. Se estivéssemos interessados apenas em pequenos roedores, seria mais conveniente trabalhar com gramas ao invés de kilogramas. O investigador desavisado, determina a massa de um hamster (200g), o aplica na equação 6.2: e obtém o seguinte

resultado:

$$M_f = 8.24$$

Este valor não seria correto nem em gramas e muito menos em kg! O que poderia estar errado? Por desencargo de consciência, ele decide realizar o cálculo novamente aplicando a massa em kg (0.2kg):

$$M_f = 0.020kg$$

este resultado (20g) parece mais realístico. Analisando dimensionalmente a equação, podemos entender o que deu errado em nossa conversão de unidades:

$$M_f = 0.082 \cdot (1000 \cdot M_c)^{0.87}$$

Se esta equação possui apenas duas variáveis, ambas de massa, porque a simples substituição de unidades não funciona? A resposta está no fato de que a massa corporal está elevada a 0.87, portanto sua unidade não é mais Kg, da mesma forma que 1 metro quadrado não é igual a 1 metro. Para a equação ser válida dimensionalmente, a constante 0.082 não pode ser adimensional. Para descobrir a sua unidade basta rearranjarmos a equação,

$$\frac{M_f}{M_c^{0.87}} = 0.082$$

e resolvê-la com o Unum. Como vemos a unidade de nossa constante é $kg^{0.13}$.

```

1 In [13]: KG/KG**0.87
2 Out[13]: 1.0 [kg0.13]

```

se quisermos multiplicar a unidade da massa do coração por 1000 ($Kg \rightarrow g$), temos que multiplicar a unidade da constante por 1000 também e isso representa multiplicá-la por $1000^{0.13}$. Podemos verificar que esta operação retorna um número 1000 vezes o valor em Kg, ou seja, o valor em gramas.

```
1 In [20]:=1000**0.13*KG**0.13 *1000**0.87*KG  
      **0.87  
2 Out[20]:=1000.0 [kg]  
3 In [21]:=0.082*(1000**0.13)*(200**0.87)  
4 Out[21]:=20.216685199791563
```

6.4 Exercícios

1. Determine as dimensões das seguintes grandezas (manualmente e depois utilizando o Unum):
 - a) Volume
 - b) Aceleração (velocidade/tempo)
 - c) Densidade (massa/volume)
 - d) Força (massa \times aceleração)
 - e) Carga elétrica (corrente \times tempo)
2. Utilizando as repostas da questão anterior determine as dimensões das seguintes grandezas:
 - a) Pressão (força/área)
 - b) $(\text{volume})^2$
 - c) Campo elétrico (força/carga elétrica)
 - d) Trabalho (força \times distância)
 - e) Energia Potencial Gravitacional ($= mgh = \text{massa} \times \text{aceleração gravitacional} \times \text{altura}$)

Capítulo 7

Teoria de Grafos

Breve introdução a teoria de grafos e sua representação computacional. Introdução ao Pacote NetworkX, voltado para a manipulação de grafos. Pré-requisitos: Programação orientada a objetos.

7.1 Introdução

A teoria de grafos é uma disciplina da matemática cujo objeto de estudo se presta, muito bem, a uma representação computacional como um objeto. Matematicamente, um grafo é definido por um conjunto finito de vértices (V) e por um segundo conjunto (A) de relações entre estes vértices, denominadas arestas. Grafos tem aplicações muito variadas, por exemplo: uma árvore genealógica é um grafo onde as pessoas são os vértices e suas relações de parentesco são as arestas do grafo.

Um grafo pode ser definido de forma não ambígua, por sua lista de arestas (A), que implica no conjunto de vértices que compõem o grafo. Grafos podem ser descritos ou mensurados através de um conjunto de propriedades:

- Grafos podem ser *direcionados* ou não;

- A *ordem* de um grafo corresponde ao seu número de vértices;
- O *tamanho* de um grafo corresponde ao seu número de arestas;
- Vértices, conectados por uma aresta, são ditos *vizinhos* ou *adjacentes*;
- A *ordem* de um vértice corresponde ao seu número de vizinhos;
- Um *caminho* é uma lista de arestas que conectam dois vértices;
- Um *ciclo* é um caminho que começa e termina no mesmo vértice;
- Um grafo sem ciclos é denominado *acíclico*.

A lista acima não exaure as propriedades dos grafos, mas é suficiente para esta introdução.

Podemos representar um grafo como um objeto Python de várias maneiras, dependendo de como desejamos utilizá-lo. A forma mais trivial de representação de um grafo em Python seria feita utilizando-se um dicionário. A Listagem 7.1 mostra um dicionário representando o grafo da figura 7.1. Neste dicionário, utilizamos como chaves os vértices do grafo associados a suas respectivas listas de vizinhos. Como tudo em Python é um objeto, poderíamos já nos aproveitar dos métodos de dicionário para analisar nosso grafo (Listagem 7.2).

Listagem 7.1: Definindo um grafo como um dicionário.

```
1 >>> g = { 'a' :[ 'c', 'd', 'e'], 'b' :[ 'd', 'e'], 'c' :[ 'a', 'd'], 'd' :[ 'b', 'c', 'a'], 'e' :[ 'a', 'b'] }
```

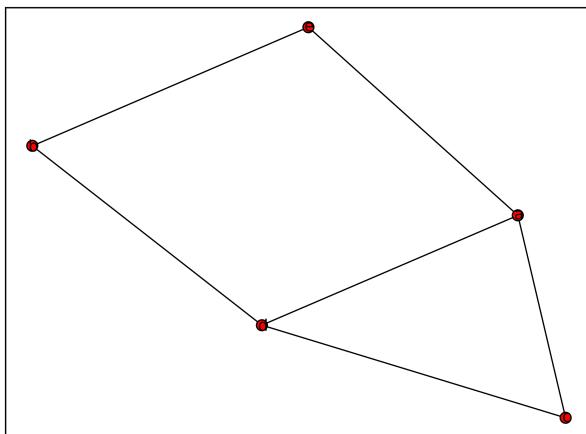


Figura 7.1: Um grafo simples.

Podemos utilizar o método `keys` para obter uma lista dos vértices de nosso grafo.

Listagem 7.2: Obtendo a lista de vértices.

```
1 >>> g.keys()  
2 [ 'a', 'c', 'b', 'e', 'd' ]
```

Uma extensão do conceito de grafos é o conceito de redes. Redes são grafos nos quais valores numéricos são associados às suas arestas. Redes herdam as propriedade dos grafos e possuem algumas propriedades específicas.

A representação de redes, a partir de objetos pitônicos simples, como um dicionário, também é possível. Porém, para dar mais alcance aos nossos exemplos sobre teoria de grafos, vamos nos uti-

lizar do pacote `NetworkX`¹ que já implementa uma representação bastante completa de grafos e redes em Python.

7.2 NetworkX

O pacote `NetworkX` se presta à criação, manipulação e estudo da estrutura, dinâmica e funções de redes complexas.

A criação de um objeto grafo a partir de seu conjunto de arestas, A , é muito simples. Seja um grafo G com vértices $V = \{W, X, Y, Z\}$:

$$G : A = \{(W, Z), (Z, Y), (Y, X), (X, Z)\}$$

Listagem 7.3: Definindo um grafo através de seus vértices

```

1 # Disponível no pacote de programas como:
2   graph1.py
3 import networkx as NX
4
5 G = NX.Graph()
6 G.add_edges_from([( 'W' , 'Z' ),( 'Z' , 'Y' ),( 'Y' , 'X' ),( 'X' , 'Z' )])
7 print G.nodes() , G.edges()
```

Executando o código acima, obtemos:

`['Y', 'X', 'Z', 'W'][('Y', 'X'), ('Y', 'Z'), ('X', 'Z'), ('Z', 'W')]`

Ao lidar com grafos, é conveniente representá-los graficamente. Vejamos como obter o diagrama do grafo da listagem 7.3:

Listagem 7.4: Diagrama de um grafo

¹<https://networkx.lanl.gov/>

```
1 # Disponivel no pacote de programas como:  
2     graph2.py  
3 import networkx as NX  
4 G = NX.Graph()  
5 G.add_edges_from([( 'W' , 'Z') ,( 'Z' , 'Y') ,( 'Y' ,  
6     'X') ,( 'X' , 'Z')])  
7 import pylab as P  
8 NX.draw(G)  
9 P.show()
```

A funcionalidade do pacote NetworkX é bastante ampla. A seguir exploraremos um pouco desta funcionalidade.

Construindo Grafos

O NetworkX oferece diferentes classes de grafos, dependendo do tipo de aplicação que desejada. Abaixo, temos uma lista dos comandos para criar cada tipo de grafo.

G=Graph() Cria um grafo simples e vazio G .

G=DiGraph() Cria grafo direcionado e vazio G .

G=XGraph() Cria uma rede vazia, ou seja, com arestas que podem receber dados.

G=XDiGraph() Cria uma rede direcionada.

G=empty_graph(n) Cria um grafo vazio com n vértices.

G=empty_graph(n,create_using=DiGraph()) Cria um grafo direcionado vazio com n vértices.

G=create_empty_copy(H) Cria um novo grafo vazio do mesmo tipo que H .

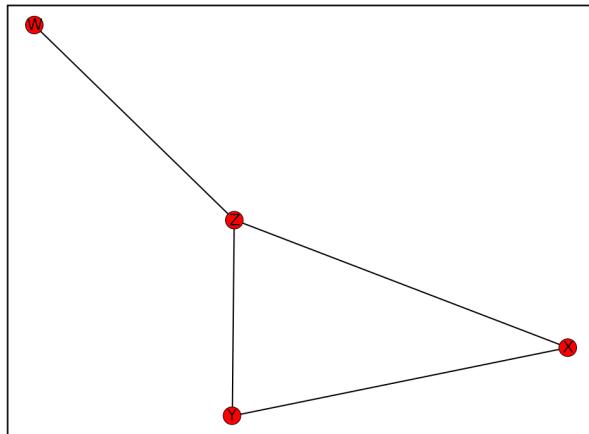


Figura 7.2: Diagrama de um grafo

Manipulando Grafos

Uma vez de posse de um objeto grafo instanciado a partir de uma das classes listadas anteriormente, é de interesse poder manipulá-lo de várias formas. O próprio objeto dispõe de métodos para este fim:

G.add_node(n) Adiciona um único vértice a G.

G.add_nodes_from(lista) Adiciona uma lista de vértices a G.

G.delete_node(n) Remove o vértice n de G.

G.delete_nodes_from(lista) Remove uma lista de vértices de G.

G.add_edge(u,v) Adiciona a aresta (u, v) a G . Se G for um grafo direcionado, adiciona uma aresta direcionada $u \rightarrow v$. Equivalente a `G.add_edge((u,v))`.

G.add_edges_from(lista) Adiciona uma lista de arestas a G .

G.delete_edge(u,v) Remove a aresta (u, v) .

G.delete_edges_from(lista) Remove uma lista de arestas de G .

G.add_path(listadevertices) Adiciona vértices e arestas de forma a compor um caminho ordenado.

G.add_cycle(listadevertices) O mesmo que `add_path`, exceto que o primeiro e o último vértice são conectados, formando um ciclo.

G.clear() Remove todos os vértices e arestas de G .

G.copy() Retorna uma cópia “rasa” do grafo G .²

G.subgraph(listadevertices) Retorna subgrafo correspondente à lista de vértices.

Criando Grafos a Partir de Outros Grafos

subgraph(G, listadevertices) Retorna subgrafo de G correspondente à lista de vértices.

union(G1,G2) União de grafos.

disjoint_union(G1,G2) União disjunta, ou seja, assumindo que todos os vértices são diferentes.

²Uma cópia rasa significa que se cria um novo objeto grafo referenciando o mesmo conteúdo. Ou seja, se algum vértice ou aresta for alterado no grafo original, a mudança se reflete no novo grafo.

cartesian_product(G1,G2) Produto cartesiano de dois grafos (Figura 7.3).

compose(G1,G2) Combina grafos, identificando vértices com mesmo nome.

complement(G) Retorna o complemento do grafo(Figura 7.3).

create_empty_copy(G) Cópia vazia de G.

convert_to_undirected(G) Retorna uma cópia não direcionada de \overline{G} .

convert_to_directed(G) Retorna uma cópia não direcionada de \overline{G} .

convert_node_labels_to_integers(G) Retorna uma cópia com os vértices renomeados como números inteiros.

Gerando um Grafo Dinamicamente

Muitas vezes, a topologia da associação entre componentes de um sistema complexo não está dada a priori. Frequentemente, esta estrutura é dada pela própria dinâmica do sistema.

No exemplo que se segue, simulamos um processo de contágio entre os elementos de um conjunto de vértices, observando ao final, a estrutura produzida pelo contágio.

Listagem 7.5: Construindo um grafo dinamicamente

```

1 # Disponível no pacote de programas como:
2   grafodin.py
3 import networkx as NX
4 import threading,random, pylab as P
5 class Contagio:
```

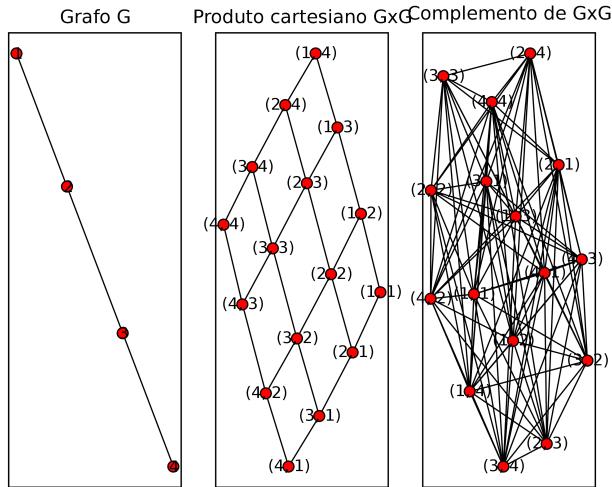


Figura 7.3: Grafo, produto cartesiano e complemento.

```

6  def __init__(self, nome):
7      self.nome = nome
8      self.doente = 0
9      self.transmite()
10
11     def transmite(self):
12         if G doentes == G.order():
13             return
14         for alvo in random.sample(G.nodes(),
15             3):
16             if not alvo doente:
17                 G.add_edge((self, alvo))
18                 print "%s infectou %s" % (self
19

```

```

17                                     . nome , alvo . nome )
t = threading . Thread ( target =
alvo . contraiu ( ) )
t . start ( )
18
19 def contraiu ( self ) :
20     self . doente +=1
21     G . doentes +=1
22     self . transmite ( )
23
24 G = NX . XDiGraph ( )
25 G . doentes = 0
26 nos = [ Contagio ( n ) for n in xrange ( 80 ) ]
27 G . add _ nodes _ from ( nos )
28 caso _ indice = G . nodes ( ) [ 0 ]
29 caso _ indice . contraiu ( )
30 print "usamos %s das arestas possiveis "%(NX .
density ( G ) )
31 print NX . degree _ histogram ( G ) , G . doentes
32 nomes = dict ( [ ( no , no . nome ) for no in G . nodes
( ) ] )
33 NX . draw ( G , labels = nomes , alpha = 0.7 , width = 2 ,
style = 'dotted' , node _ size = 450 , font _ size
= 14 )
34 P . savefig ( 'contagio . png ' , dpi = 400 )
35 P . show ( )

```

Módulo `threading`: Permite executar mais de uma parte do programa em paralelo, em um “fio” de execução independente. Este fios, compartilham todas as variáveis globais e qualquer alteração nestas é imediatamente visível a todos os outros fios.

O objeto grafo do `NetworkX` aceita qualquer objeto como um vértice. Na listagem 7.5, nos valemos deste fato para colocar ins-

tâncias da classe `Contagio` como vértices do grafo G . O grafo G é construído somente por vértices (desconectado). Então infectamos um vértice do grafo, chamando o seu método `contraiu()`. O vértice, após declarar-se doente e incrementar o contador de doentes a nível do grafo, chama o método `transmite()`.

O método `transmite` assume que durante seu período infecioso, cada vértice tem contatos efetivos com apenas dez outros vértices. Então cada vértice irá transmitir para cada um destes, desde que não estejam já doentes.

Cada vértice infectado inicia o método `contraiu` em um “thread” separado. Isto significa que cada vértice sai infectando os restantes, em paralelo. Na verdade, como o interpretador Python só executa uma instrução por vez, cada um destes objetos recebe do interpretador uma fatia de tempo por vez, para executar suas tarefas. Pode ser que o tempo de uma destas fatias seja suficiente para infectar a todos no seu grupo, ou não. Depois que o processo se desenrola, temos a estrutura do grafo como resultado (Figura 7.4)

Construindo um Grafo a Partir de Dados

O conceito de grafos e redes é extremamente útil na representação e análise de sistemas complexos, com muitos componentes que se relacionam entre si. Um bom exemplo é uma rede social, ou seja, uma estrutura de interação entre pessoas. Esta interação pode ser medida de diversas formas. No exemplo que se segue, vamos tentar inferir a rede social de um indivíduo, por meio de sua caixa de mensagens.

Listagem 7.6: Construindo uma rede social a partir de e-mails

```
1 # Disponível no pacote de programas como:  
    mnet.py  
2 import email, sys  
3 import email.Errors
```

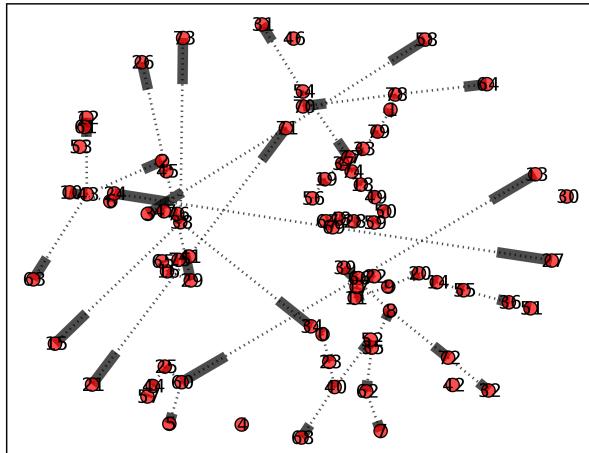


Figura 7.4: Resultado da simulação do exemplo 7.5.

```
4| import mailbox
5| import networkx as NX
6| from pylab import show
7|
8| def msgfactory(fp):
9|     try:
10|         return email.message_from_file(fp)
11|     except email.Errors.MessageParseError:
12|         return ''
13|
14| def starG(mess,G):
15|     try:
```

```
16     centro = mess.get('from').split()
17         [-1].strip('<>')
18     pontas = [j.split()[-1].strip('<>')
19             for j in mess.get_all('to')[0].
20             split(',')]
21     arestas = [(centro,k) for k in
22                 pontas]
23     G.add_edges_from(arestas)
24     return G
25 except:
26     print'falhou'
27
28 dir='/home/flavio/Mail/inbox'
29 # Veja documentacao do modulo mailbox
30 # para outros tipo de mailboxes.
31 mbox = mailbox.Maildir(dir, msgfactory) #
32     mailbox do kmail
33 G = NX.DiGraph()
34 for n in xrange(50):
35     i = mbox.next()
36     starG(i,G)
37 print G.nodes()
38 G = NX.convert_node_labels_to_integers(G)
39 NX.draw(G, width=2, style='dotted', alpha
40         =0.5)
41 show()
```

Na Listagem 7.6, usamos dois módulos interessantes da biblioteca padrão do Python: O módulo `email` e o módulo `mailbox`.

Módulo email: Módulo para decodificar, manusear, e compor emails.

Módulo mailbox: Conjunto de classes para lidar com caixas de correio no formato Unix, MMDF e MH.

Neste exemplo, utilizei a minha mailbox associada com o programa Kmail; portanto, se você usa este mesmo programa, basta substituir o diretório de sua mailbox e o programa irá funcionar para você. Caso use outro tipo de programa de email, consulte a documentação do Python para buscar a forma correta de ler o seu mailbox.

A classe `Maildir` retorna um iterador, que por sua vez, retornará mensagens decodificadas pela função `msgfactory`, definida por nós. Esta função se utiliza do módulo `email` para decodificar a mensagem.

Cada mensagem recebida é processada para gerar um grafo do tipo “estrela”, com o remetente no centro e todos os destinatários da mensagem nas pontas. Este grafo é então adicionado ao grafo original, na forma de uma lista de arestas. Depois de todas as mensagens terem sido assim processadas, geramos a visualização do grafo (Figura 7.5).

7.3 Exercícios

1. Determine o conjunto de arestas A que maximiza o tamanho do grafo cujos vértices são dados por $V = \{a, b, c, d, e\}$.
2. No exemplo do contágio, verifique se existe alguma relação entre o tamanho da amostra de cada vértice e a densidade final do grafo.
3. Ainda no exemplo do contágio, refaça o experimento com um grafo de topologia dada a priori no qual os vértices só podem infectar seus vizinhos.

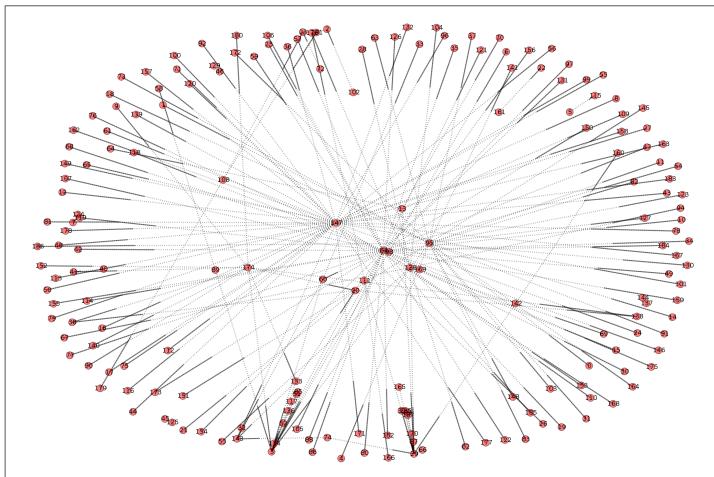


Figura 7.5: Rede social a partir de mensagens de email.

4. Insira um print no laço for do exemplo 7.6 para ver o formato de saída do iterador mbox.
5. Modifique o programa 7.6 para associar apenas mensagens que contêm uma palavra em comum.

Capítulo 8

Interação com Bancos de Dados

Apresentação dos módulos de armazenamento de dados Pickle e Sqlite3 que fazem parte da distribuição padrão do Python. Apresentação do pacote SQLAlchemy para comunicação com os principais sistemas de bancos de dados existentes. Pré-requisitos: Conhecimentos básicos de bancos de dados e SQL.

O gerenciamento de dados não se constitui numa disciplina científica *per se*. Entretanto, cada vez mais, permeia as atividades básicas de trabalho científico. O volume crescente de dados e o aumento de sua complexidade há muito ultrapassou a capacidade de gerenciamento através de simples planilhas.

Atualmente, é muito comum a necessidade de se armazenar dados quantitativos, qualitativos e mídias dos mais diferentes formatos(imagens, vídeos, sons) em uma plataforma integrada de onde possam ser facilmente acessados para fins de análise, visualização ou simplesmente consulta.

A linguagem Python dispõe de soluções simples para resolver esta necessidade em seus mais distintos níveis de sofisticação. Seguindo a filosofia de “baterias incluídas” do Python, a sua biblioteca

padrão nos apresenta o módulo Pickle e cPickle e, a partir da versão 2.5, o banco de dados relacional sqlite3.

8.1 O Módulo Pickle

O módulo `pickle` e seu primo mais veloz `cPickle`, implementam algoritmos que permitem armazenar, em um arquivo, objetos implementados em Python.

Listagem 8.1: Exemplo de uso do módulo `pickle`

```

1 In [1]: import pickle
2 In [2]: class oi:
3     .2.:     def digaoi(self):
4         .2.:             print "oi"
5 In [3]: a= oi()
6 In [4]: f = open( 'picteste' , 'w')
7 In [5]: pickle.dump(a,f)
8 In [6]: f.close()
9 In [7]: f = open( 'picteste' , 'r')
10 In [8]: b=pickle.load(f)
11 In [9]: b.digaoi()
12 oi

```

Como vemos na listagem 8.1, com o módulo `pickle` podemos armazenar objetos em um arquivo, e recuperá-lo sem problemas para uso posterior. Contudo, uma característica importante deste módulo não fica evidente no exemplo 8.1. Quando um objeto é armazenado por meio do módulo `pickle`, nem o código da classe, nem seus dados, são incluídos, apenas os dados da instância.

```

1 In [10]: class oi:
2     .10.:     def digaoi(self , nome='flavio'):
3         .10.:             print 'oi %s!' % nome
4
5 In [11]: f = open( 'picteste' , 'r')

```

```
6 In [12]: b=pickle.load(f)
7 In [13]: b.digaoi()
8 oi flavio!
```

Desta forma, podemos modificar a classe, e a instância armazenada reconhecerá o novo código ao ser restaurada a partir do arquivo, como podemos ver acima. Esta característica significa que os `pickles` não se tornam obsoletos quando o código em que foram baseados é atualizado (naturalmente isto vale apenas para modificações que não removam atributos já incluídos nos `pickles`).

O módulo `pickle` não foi construído para armazenamento de dados, pura e simplesmente, mas de objetos computacionais complexos, que podem conter em si, dados. Apesar desta versatilidade, peca por consistir em uma estrutura de armazenamento legível apenas pelo próprio módulo `pickle` em um programa Python.

8.2 O módulo Sqlite3

Este módulo passa a integrar a biblioteca padrão do Python a partir da versão 2.5. Portanto, passa a ser uma excelente alternativa para usuários que requerem a funcionalidade de um banco de dados relacional compatível com SQL¹.

O Sqlite nasceu de uma biblioteca em C que disponibilizava um banco de dados extremamente leve e que dispensa o conceito “servidor-cliente”. No `sqlite`, o banco de dados é um arquivo manipulado através da biblioteca `sqlite`.

Para utilizar o `sqlite` em um programa Python, precisamos importar o módulo `sqlite3`.

```
1 In [1]: import sqlite3
```

¹SQL significa “Structured Query Language”. o SQL é um padrão internacional na interação com bancos de dados relacionais. Para saber mais, consulte <http://pt.wikipedia.org/wiki/SQL>

O próximo passo é a criação de um objeto “conexão”, através do qual podemos executar comandos SQL.

```
1 In [2]: c = sqlite3.connect( '/tmp/exemplo' )
```

Agora dispomos de um banco de dados vazio, consistindo no arquivo `exemplo`, localizado no diretório `/tmp`. O `sqlite` também permite a criação de bancos de dados em RAM; para isso basta substituir o nome do arquivo pela string “`:memory:`”. Para podermos inserir dados neste banco, precisamos primeiro criar uma tabela.

```
1 In [3]: c.execute( '''create table especimes( nome text , altura real , peso real)''' )
2 Out[3]: <sqlite3.Cursor object at 0x83fed10>
```

Note que os comandos SQL são enviados como strings através do objeto `Connection`, método `execute`. O comando `create table` cria uma tabela; ele deve ser necessariamente seguido do nome da tabela e de uma lista de variáveis tipadas(entre parênteses), correspondendo às variáveis contidas nesta tabela. Este comando cria apenas a estrutura da tabela. Cada variável especificada corresponderá a uma coluna da tabela. Cada registro, inserido subsequentemente, formará uma linha da tabela.

```
1 In [4]: c.execute( '''insert into especimes values('tom',12.5,2.3)'')
```

O comando `insert` é mais um comando SQL útil para inserir registros em uma tabela.

Apesar dos comandos SQL serem enviados como strings através da conexão, não se recomenda, por questão de segurança, utilizar os métodos de formatação de strings (`' ... values(%s,%s)'%(1,2)`) do Python. Ao invés, deve-se fazer o seguinte:

```
1 In [5]: t = ('tom',)
2 In [6]: c.execute('select * from especimes where nome=?', t)
```

```
3 In [7]: c.fetchall()
4 [( 'tom' , 12.5 , 2.299999999999998 )]
```

No exemplo acima utilizamos o método `fetchall` para recuperar o resultado da operação. Caso desejássemos obter um único registro, usariámos `fetchone`.

Abaixo, vemos como inserir mais de um registro a partir de estruturas de dados existentes. Neste caso, trata-se de repetir a operação descrita no exemplo anterior, com uma sequência de tuplas representando a sequência de registros que se deseja inserir.

```
1 In [8]: t = (( 'jerry' , 5.1 , 0.2 ) , ( 'butch'
     , 42.4 , 10.3 ))
2 In [9]: for i in t:
3     .9.:     c.execute( 'insert into especimes
     values ( ?, ?, ? ) ' , i )
```

O objeto `cursor` também pode ser utilizado como um iterador para obter o resultado de uma consulta.

```
1 In [10]: c.execute( 'select * from especimes
     by peso' )
2 In [11]: for reg in c:
3             print reg
4 ( 'jerry' , 5.1 , 0.2 )
5 ( 'tom' , 12.5 , 2.299999999999998 )
6 ( 'butch' , 42.4 , 10.3 )
```

O módulo `sqlite` é realmente versátil e útil, porém, requer que o usuário conheça, pelo menos, os rudimentos da linguagem SQL. A solução apresentada a seguir procura resolver este problema de uma forma mais “pitônica”.

8.3 O Pacote SQLObject

O pacote SQLObject² estende as soluções apresentadas até agora de duas maneiras: oferece uma interface orientada a objetos para bancos de dados relacionais e, também, nos permite interagir com diversos bancos de dados sem ter que alterar nosso código.

Para exemplificar o `sqlobject`, continuaremos utilizando o `sqlite` devido à sua praticidade.

Construindo um aranha digital

Neste exemplo, teremos a oportunidade de construir uma aranha digital que recolherá informações da web (Wikipedia³) e as armazenará em um banco sqlite via `sqlobject`.

Para este exemplo, precisaremos de algumas ferramentas que vão além do banco de dados. Vamos explorar a capacidade da biblioteca padrão do Python para interagir com a internet, e vamos nos utilizar de um pacote externo para decodificar as páginas obtidas.

Listagem 8.2: Módulos necessários

```

1 # disponivel no pacote de programas como:
2     aranha.py
3 import networkx as NX
4 from BeautifulSoup import SoupStrainer,
5         BeautifulSoup as BS
6 from BeautifulSoup import BeautifulSoup
7         as XS
8 import sys, os, urllib2, urllib, re
9 from sqlobject import *

```

²<http://www.sqlobject.org/>

³<http://pt.wikipedia.org>

O pacote `BeautifulSoup`⁴ é um “destrinchador” de páginas da web. Um dos problemas mais comuns ao se lidar com páginas html, é que muitas delas possuem pequenos defeitos em sua construção que nossos navegadores ignoram, mas que podem atrapalhar uma análise mais minuciosa. Daí o valor do `BeautifulSoup`; ele é capaz de lidar com páginas defeituosas, retornando uma estrutura de dados com métodos que permitem uma rápida e simples extração da informação que se deseja. Além disso, se a página foi criada com outra codificação, o `BeautifulSoup`, retorna todo o conteúdo em Unicode, automaticamente, sem necessidade de intervenção do usuário.

Da biblioteca padrão, vamos nos servir dos módulos `sys`, `os`, `urllib`, `urllib2` e `re`. A utilidade de cada um ficará clara à medida que avançarmos no exemplo.

O primeiro passo é especificar o banco de dados. O `sqlobject` nos permite escolher entre MySQL, PostgreSQL, `sqlite`, Firebird, MAXDB, Sybase, MSSQL, ou ADODBAPI. Entretanto, conforme já explicamos, nos restringiremos ao uso do banco `sqlite`.

Listagem 8.3: Especificando o banco de dados.

```

8 laracnadir = os.path.expanduser('~/laracna')
9     )
10 if not os.path.exists(laracnadir):
11     os.mkdir(laracnadir)
11 sqlhub.processConnection = connectionForURI(
    'sqlite://'+laracnadir+'/knowdb')

```

Na listagem 8.3, criamos o diretório(`os.mkdir`) onde o banco de dados residirá (se necessário) e definimos a conexão com o banco. Utilizamos `os.path.exists` para verificar se o diretório existe. Como desejamos que o diretório fique na pasta do usuário, e não temos como saber, de antemão, qual é este diretório, utiliza-

⁴<http://www.crummy.com/software/BeautifulSoup/>

mos `os.path.expanduser` para substituir o `~` por `/home/usuario` como aconteceria no console unix normalmente.

Na linha 11 da listagem 8.3, vemos o comando que cria a conexão a ser utilizada por todos os objetos criados neste módulo.

Em seguida, passamos a especificar a tabela do nosso banco de dados como se fora uma classe, na qual seus atributos são as colunas da tabela.

Listagem 8.4: Especificando a tabela `ideia` do banco de dados.

```

16 class Ideia(SQLObject):
17     nome = UnicodeCol()
18     nlinks = IntCol()
19     links = PickleCol()
20     ender = StringCol()

```

A classe que representa nossa tabela é herdeira da classe `SQLObject`. Nesta classe, a cada atributo (coluna da tabela) deve ser atribuído um objeto que define o tipo de dados a ser armazenado. Neste exemplo, vemos quatro tipos distintos, mas existem vários outros. `UnicodeCol` representa textos codificados como Unicode, ou seja, podendo conter caracteres de qualquer língua. `IntCol` corresponde a números inteiros. `PickleCol` é um tipo muito interessante pois permite armazenar qualquer tipo de objeto Python. O mais interessante deste tipo de coluna, é que não requer que o usuário invoque o módulo pickle para armazenar ou ler este tipo de variável. As variáveis são convertidas/reconvertidas automaticamente, de acordo com a operação. Por fim, temos `StringCol` que é uma versão mais simples de `UnicodeCol`, aceitando apenas strings de caracteres `ascii`. Em SQL é comum termos que especificar diferentes tipos, de acordo com o comprimento do texto que se deseja armazenar em uma variável. No `sqlobject`, não há limite para o tamanho do texto que se pode armazenar tanto em `StringCol` quanto em `UnicodeCol`.

A funcionalidade da nossa aranha foi dividida em duas classes: `Crawler`, que é o rasteador propriamente dito, e a classe `UrlFac` que constrói as urls a partir da palavra que se deseja buscar na Wikipedia.

Cada página é puxada pelo módulo `urllib2`. A função `urlencode` do módulo `urllib`, facilita a adição de dados ao nosso pedido, de forma a não deixar transparecer que este provém de uma aranha digital. Sem este disfarce, a Wikipedia recusa a conexão.

A páginas são então analisadas pelo método `verResp`, no qual o `BeautifulSoup` tem a chance de fazer o seu trabalho. Usando a função `SoupStrainer`, podemos filtrar o resto do documento, que não nos interessa, analizando apenas os links (tags 'a') cujo destino são urls começadas pela string `/wiki/`. Todos os artigos da wikipedia, começam desta forma. Assim, evitamos perseguir links externos. A partir da “sopa” produzida, extraímos apenas as urls, ou seja, o que vem depois de `href=`. Podemos ver na listagem 8.5 que fazemos toda esta filtragem sofisticada em duas linhas de código(55 e 56), graças ao `BeautifulSoup`.

Listagem 8.5: Restante do código da aranha.

```

15 urlatual = ''
16 class Ideia(SQLObject):
17     nome = UnicodeCol()
18     nlinks = IntCol()
19     links = PickleCol()
20     ender = StringCol()
21
22 class Crawler:
23     def __init__(self, starturl, depth):
24         try:
25             Ideia.createTable()
26         except:
27             pass
28         self.SU = starturl

```

194 CAPÍTULO 8. INTERAÇÃO COM BANCOS DE DADOS

```
29         self.depth = depth
30         self.fila = []
31         self.depth = depth
32         self.curdepth = 0
33         self.started = 0
34         self.nlinks = 0
35         self.history = []
36         self.G = NX.Graph()
37     def parsePag(self, urlend):
38         urlatual = urlend
39         user_agent = 'Mozilla/4.0 ('
40             compatible; MSIE 5.5; Windows NT
41             )'
42         values = { 'name' : 'John Smith',
43                 'location' : 'Northampton',
44                 'language' : 'Python' }
45         headers = { 'User-Agent' :
46                     user_agent }
47         data = urllib.urlencode(values)
48         print "Abrindo ", urlend
49         req = urllib2.Request(urlend, data,
50                               headers)
51         fd = urllib2.urlopen(req)
52         html = fd.read()
53         return html
54
55     def verResp(self, html):
56         '''
57         Verifica se resposta é um hit ou não
58         '''
59         lnkart = SoupStrainer('a', href=re.compile('^/wiki/*'))
60         artlist = [tag['href'] for tag in
61                    BS(html, parseOnlyThese=lnkart)]
```

```
57     if artlist[0].endswith('Disambig.svg'):
58         self.fila.append('http://'+langatual+'.wikipedia.org'+artlist[3])
59         self.curlinks = artlist
60     else:
61         self.curlinks = artlist
62         Ideia(nome=nomeatual, nlinks =
63             len(artlist), links =
64             artlist, ender = urlatual)
65         self.G.add_edges_from([(nomeatual, i) for i in self.
66             curlinks]))
67         if self.curdepth > self.depth:
68             return
69         self.fila.extend(['http://'+langatual+'.wikipedia.org' +
70             i for i in artlist])
71         self.curdepth +=1
72
73     def move(self):
74         if not self.fila:
75             if not self.started:
76                 self.fila.append(self.SU)
77             while self.fila:
78                 self.started = 1
79                 urlatual = self.fila.pop(0)
80                 nomeatual = urlatual.split('/')[-1]
81                 if ":" in nomeatual: continue
82                 if nomeatual in ['Main_page']+self.history: continue
83                 print "buscando ", nomeatual,
```

196 CAPÍTULO 8. INTERAÇÃO COM BANCOS DE DADOS

```
80         print "Faltam ", len(self.fila)
81     try:
82         html = self.parsePag(
83             urlatual)
84     except:
85         continue
86     self.verResp(html)
87     self.nlinks +=1
88     self.history.append(nomeatual)
89
90 class UrlFac:
91     def __init__(self, lang='en'):
92         global langatual
93         self.lang = lang
94         langatual = lang
95     def urlifica(self, palavra):
96         nomeatual = palavra
97         u = "http://"+self.lang+".wikipedia.
98             org/wiki/"+palavra
99         urlatual = u
100        return u
101 if __name__=="__main__":
102     UF = UrlFac('pt')
103     u = UF.urlifica(sys.argv[1])
104     Cr = Crawler(u,1)
105     Cr.move()
```

A listagem 8.5 mostra o restante do código da aranha e o leitor poderá explorar outras solução implementadas para otimizar o trabalho da aranha. Note que não estamos guardando o html completo das páginas para minimizar o espaço de armazenamento, mas este programa pode ser modificado facilmente de forma a reter

todo o conteúdo dos artigos.

8.4 Exercícios

1. Modifique a aranha apresentada neste capítulo, para guardar os documentos varridos.
2. Crie uma classe capaz de conter os vários aspectos (links, figuras, etc) de um artigo da wikipedia, e utilize a aranha para criar instâncias desta classe para cada artigo encontrado, a partir de uma única palavra chave. Dica: para simplificar a persistência, armazene o objeto artigo como um Pickle, no banco de dados.

Capítulo 9

Simulações Estocásticas

Seleção de problemas relacionados com a simulação e análise de processos estocásticos. Pré-requisitos: Conhecimentos avançados de estatística.

Neste capítulo, ilustraremos métodos computacionais voltados para a geração e análise de processos estocásticos.

Em probabilidade, um processo estocástico é uma função que produz resultados aleatórios. Se o domínio desta função for o tempo, o processo estocástico se manifestará como uma série temporal. Processos estocásticos espaciais, geram os chamados campos aleatórios

Exemplos de processo estocásticos são séries temporais de preços de ações, flutuações de populações, distribuição espacial de poluição particulada.

Muitos processos estocásticos naturais possuem estrutura em meio à aleatoriedade. Muitas das técnicas desenvolvidas para lidar com estes processos visam encontrar esta estrutura.

9.1 Números Aleatórios

Ao se estudar processos estocásticos por meio computacional, o primeiro problema que se precisa resolver é o da geração de núme-

ros aleatórios de forma confiável. Felizmente, hoje em dia, pacotes como o `numpy` ou o `scipy`, fornecem geradores de números aleatórios de qualidade para muitas famílias de distribuições, de forma que na maioria das vezes não precisamos nos preocupar com este problema. Vez por outra, problemas de amostragem mais complexos podem exigir geradores mais sofisticados. Neste capítulo exploraremos algumas destas situações.

Hipercubo Latino - LHS

A técnica de amostragem do hipercubo latino, mais conhecida pelo seu nome original “Latin Hypercube Sampling” ou simplesmente LHS, é uma técnica de geração de amostras aleatórias a partir de uma distribuição conhecida. O diferencial deste algoritmo é que ele garante a amostragem de áreas de baixa probabilidade, mesmo em amostras de tamanho pequeno a moderado (Figura 9.1).

A vantagem de se utilizar o LHS está na redução do custo computacional necessário para cobrir o espaço amostral, sem deixar brechas significativas.

O algoritmo do LHS funciona particionando o suporte da variável em um número n de intervalos equiprováveis, onde n é o número de amostras que se deseja obter (Figura 9.2). Então retiramos uma amostra de cada intervalo (quantis) definido no eixo y da figura 9.2¹. Isto é feito por amostragem uniforme naqueles intervalos (Linha 8 da listagem 9.1). Invertendo-se a função de densidade acumulada, obtemos as amostras de x correspondentes. Esta inversão é calculada por meio da função `ppf` (“percentile point function”) que é o inverso da função de densidade acumulada (Linha 9 da listagem 9.1). As amostras assim geradas equivalem a amostras retiradas da distribuição de x , mas com uma cobertura homogênea do seu suporte. Na listagem 9.1, a amostragem por Hipercubo Latino é implementada como uma função de apenas

¹Veja no programa `lhsexp.py` como o gráfico da figura 9.2 foi gerado

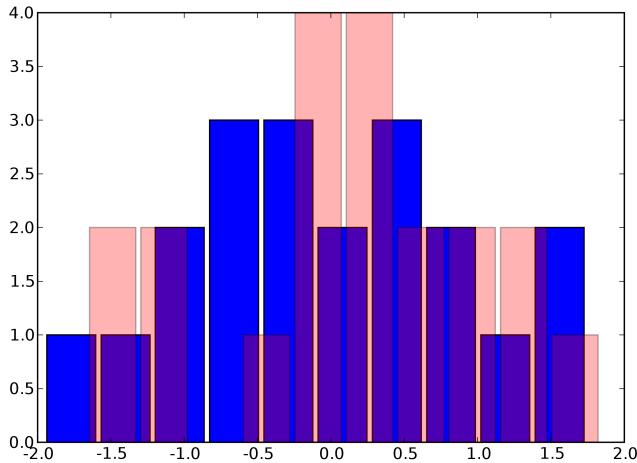


Figura 9.1: Amostragem ($n=20$) de uma distribuição normal com média 0 por LHS (histograma mais escuro) e amostragem padrão do `scipy` (histograma mais claro).

três linhas, graças à utilização dos geradores de números aleatórios disponíveis no módulo `stats` do `scipy`. Neste módulo os geradores são classes portadoras de métodos muito convenientes como o `rvs` que gera amostras aleatórias, e o `ppf`(ou percentile point function) que representa a inversa da função de densidade acumulada. Além dos métodos utilizados neste exemplo, existem muitos outros. O leitor é encorajado a explorar estas classes e seus métodos.

Listagem 9.1: Amostragem por Hipercubo latino (LHS)

```
1 #!/usr/bin/python
2 # Disponível no pacote de programas como:
```

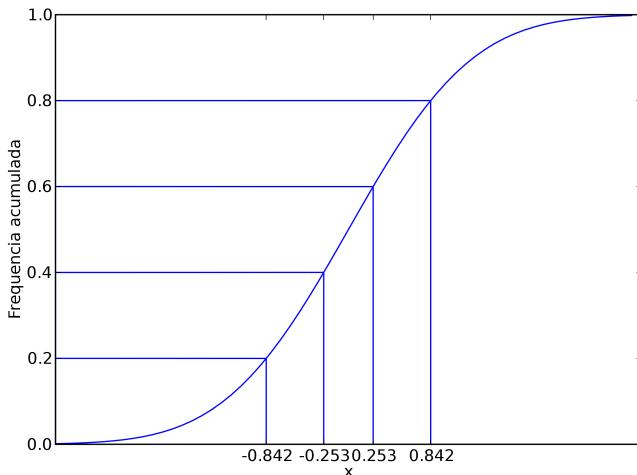


Figura 9.2: Distribuição acumulada da variável $x \sim N(0, 1)$, mostrando o particionamento do espaço amostral em segmentos equi-prováveis.

```

3   ||  lhs.py
4   ||  from pylab import plot, figure, hist, show,
5   ||  savefig
6   ||  import scipy.stats as stats
7   ||  import numpy
8
9   ||  def lhs(dist, parms, n=100):
10  ||    perc = numpy.arange(0, 1., 1./n)
11  ||    smp = [stats.uniform(i, 1./n).rvs() for i
12  ||          in perc]
13  ||    v = dist(parms[0], 1./parms[1]).ppf(smp)

```

```

11    return v
12
13 if __name__=='__main__':
14     c=lhs(stats.norm, [0,1],20)
15     hist(c)
16     n = stats.norm.rvs(size=20)
17     hist(n.ravel(), facecolor='r', alpha =0.3)
18     savefig('lhs.png', dpi=400)
19     show()

```

9.2 Inferência Bayesiana

Atualmente, a resolução de problemas de inferência Bayesiana de considerável complexidade, tem se tornado cada vez mais acessível, graças à utilização de métodos computacionais. Isto não significa que a computação bayesiana seja simples; pelo contrário. Entretanto, as vantagens da inferência Bayesiana sobre a abordagem frequentista, justificam um certo esforço por parte do pesquisador no sentido de dominar estas técnicas.

Antes de chegarmos às técnicas computacionais utilizadas para resolver os problemas da inferência Bayesiana, vamos nos deter um pouco sobre o porquê da intratabilidade analítica da inferência Bayesiana, aplicada a modelos complexos.

Sejam dois eventos E e F . A probabilidade condicional de E dado F é dada por:

$$P(E | F) = \frac{P(E \cap F)}{P(F)}$$

A partir desta fórmula, podemos obter a versão mais simples do teorema de Bayes.

$$P(E | F) = \frac{P(F | E)P(E)}{P(F)}$$

Se particionarmos o eventos E em n eventos, mutuamente excludentes, pelo teorema da probabilidade total, chegaremos à versão mais comumente utilizada da fórmula de Bayes:

$$P(E_i | F) = \frac{P(F | E_i)P(E_i)}{\sum_{i=1}^n P(F | E_i)P(E_i)}$$

O teorema de Bayes nos permite calcular as probabilidades de um conjunto de hipóteses explanatórias (E_i), dado que observamos certos fatos (F): $P(E_i | F)$. Mas o teorema de Bayes também nos diz que estas probabilidades dependem das probabilidades *a priori* das hipóteses $P(E_i)$ (antes da observação dos fatos). Dadas estas, o teorema de Bayes nos fornece uma excelente ferramenta para atualização de nossos conhecimentos prévios, $P(E_i)$ à luz de novos fatos $P(E_i | F)$.

A definição acima reflete uma situação em que temos um conjunto discreto de hipóteses explanatórias (E_i), sobre as quais desejamos realizar nossa inferência. Quando nossas hipóteses são representadas, não por um conjunto enumerável de opções, mas pelo valor de uma ou mais variáveis contínuas, Θ , e os fatos ou dados denotados por X , nossos conhecimentos prévios passam a ser representados por distribuições de probabilidade $p(\theta)$ e os dados, condicionados às hipóteses, $f(x | \theta)$, são representados pela função de verossimilhança $L(\theta; x)$.

Com a ajuda de alguma álgebra e do teorema da probabilidade total, em sua versão contínua, chegamos à forma contínua do teorema de Bayes:

$$\pi(\theta | x) = \frac{p(\theta)L(\theta; x)}{\int_{\Theta} L(\theta; x)p(\theta)d\theta}$$

Ao resolvemos a integral no denominador, este deixa de ser uma função de θ e torna-se apenas uma constante de proporcionalidade que garante que a integral de $\pi(\theta | x)$ seja igual a 1. Isto nos permite re-escrever a equação acima como:

$$\pi(\theta | x) \propto p(\theta)L(\theta; x)$$

Logo, a densidade posterior de θ é proporcional à sua densidade *a priori* multiplicada pela verossimilhança.

Conforme vimos acima, a inferência Bayesiana pode ser bastante intuitiva, conceitualmente. Na prática, porém, quando queremos calcular a distribuição posterior dos parâmetros, nos deparamos com vários problemas numéricos de considerável dificuldade.

Para encontrar a constante de proporcionalidade mencionada na seção anterior, precisamos integrar o produto da distribuição *a priori* dos parâmetros pela verossimilhança, sobre o suporte de Θ . Se este suporte é infinito e/ou multidimensional, temos um problema de integração numérica complicadíssimo.

Uma vez obtida a constante de proporcionalidade, se o espaço de parâmetros é multidimensional, encontrar as distribuições marginais para cada um deles, também é um problema de integração numérica difícil.

9.3 Simulações Estocásticas

Se não podemos determinar de forma exata, analítica ou numericamente, as distribuições posteriores geradas pelo teoremas de Bayes, só nos resta tentar simular amostras destas distribuições e estudá-las. Esta abordagem é denominada método de Monte Carlo, em homenagem ao famoso cassino de Mônaco.

Integração Monte Carlo

Suponhamos que precisemos resolver a seguinte integral²:

²Para humanizar um pouco esta equação, imagine que $f(y | x)$ representa a probabilidade condicional de y dado x , e que $g(x)$ a função de densidade de probabilidade de x .

$$J(y) = \int f(y | x)g(x)dx = E[f(y | x)] < +\infty$$

Se $g(x)$ é uma densidade da qual podemos amostrar, podemos aproximar nossa integral da seguinte forma:

$$\hat{J}(y) = \frac{1}{n} \sum_{i=1}^n f(y | x_i)$$

onde $x_1, \dots, x_n \sim g(x)$. Pela lei dos grandes números, esta estimativa se aproximará assintoticamente de $J(y)$.

Vejamos um exemplo: gostaríamos de encontrar a integral da função $y = x$ no intervalo $[0, 1]$. Pela fórmula da área do triângulo, $(base \times altura)/2$ o resultado exato é 0.5. Na listagem 9.2, nosso $g(x)$ passa a ser uma distribuição uniforme entre 0 e 1, visto que este é o intervalo sobre qual desejamos integrar nossa função. Vemos que quanto maior o número de amostras de x , mais nossa estimativa se aproximará de 0.5

Listagem 9.2: Integração Monte Carlo

```

1 from numpy import *
2 from numpy.random import *
3 # desejamos área embaixo de uma reta
4 # dada pela função y=x
5 # ou seja: 0.5
6 x = uniform(0,1,5000)
7 # Seja f(y|x) = x
8 J = 1./5000*sum(x)
9 print J

```

9.4 Amostragem por Rejeição

Suponhamos que se deseje obter amostras de uma função de densidade $p(x)$, com suporte no intervalo $[a, b]$, tal que $p(x) \leq m$, $\forall x \in [a, b]$. Então definimos:

$$X \sim U[a, b)$$

e

$$Y \sim U[0, m)$$

Geramos x e y a partir destas distribuições e aceitamos x como um valor de $p(x)$ se $y < f(x)$, caso contrário, rejeitamos a amostra e tentamos novamente. Este procedimento significa que definimos uma função constante $f(x) = m$, que funciona como um “envelope” para o nosso processo de amostragem.

Em suma, geramos um valor x a partir do suporte de X e aceitamos este valor com uma probabilidade $f(x)/m$, caso contrário rejeitamos e tentamos novamente.

A eficiência deste método depende da proporção de pontos amostrados que são aceitos. A probabilidade de aceitação para este método é:

$$P(\text{aceitar}) = P((X, Y) \in A)$$

$$\begin{aligned} &= \int_a^b P((X, Y) \in A \mid X = x) \times \frac{1}{b-a} dx \\ &= \int_a^b \frac{f(x)}{m} \times \frac{1}{b-a} dx \\ &= \frac{1}{m(b-a)} \int_a^b f(x) dx \\ &= \frac{1}{m(b-a)} \end{aligned}$$

Se a probabilidade de aceitação for muito baixa, devemos buscar um outro método mais eficiente. Especialmente para distribuições com suporte infinito, a escolha de uma “função envelope” mais adequada, pode ajudar a aumentar a eficiência da amostragem.

Método do Envelope

Suponhamos que desejemos simular X com uma densidade de probabilidade (PDF) $f(\cdot)$ e que podemos obter amostras de Y (que possui o mesmo suporte de X), cuja PDF é $g(\cdot)$. Suponhamos ainda que exista uma constante a tal que:

$$f(x) \leq ag(x), \quad \forall x$$

ou seja, a é um limite superior para $f(x)/g(x)$.

Então, amostramos $Y = y$ de $g(\cdot)$, e $U = u \sim U[0, ag(y)]$. Aceitamos y como uma amostra de X se $u < f(y)$; caso contrário, rejeitamos e tentamos novamente. Este método funciona pois distribui pontos uniformemente sobre a região que envolve $f(x)$, e aceita apenas os pontos que cumprem o requisito de pertencer à região delimitada por $f(x)$.

Qual a probabilidade de aceitação?

$$\begin{aligned} P(U < f(Y)) &= \int_{-\infty}^{\infty} P(U < f(Y) \mid Y = y) g(y) dy \\ &= \int_{-\infty}^{\infty} \frac{f(y)}{ag(y)} g(y) dy \\ &= \int_{-\infty}^{\infty} \frac{f(y)}{a} dy \\ &= \frac{1}{a} \end{aligned}$$

Naturalmente, devemos escolher $g(x)$ de forma que a seja o menor possível.

Vamos ver como podemos representar estas idéias em Python.

Listagem 9.3: Amostragem por rejeição utilizando o método do envelope

```
1 # Disponivel no pacote de programas como:  
2     envelope.py  
3 from numpy import *  
4 from numpy.random import *  
5 from pylab import *  
6  
6 def amostra(n):  
7     """  
8         Esta funcao amostra de x e retorna  
9             um vetor mais curto que n.  
10        """  
11    x=uniform(0,1,n)  
12    y=uniform(0,1,n)*1.5  
13    fx=6*x*(1-x) # x tem distribuicao beta  
14    #retorna so os valores que satisfazem a  
15    #seguinte condicao:  
16    s=compress(y<fx,x)  
17    return s  
18  
18 def eficiencia(vector,n):  
19     """  
20         Testa a eficiencia da amostragem.  
21         retorna a probabilidade de aceitacao.  
22     """  
23     l = len(vector)  
24     prob = l/float(n)  
25     diff = n-l
```

```

26     #n necessario para obter as amostras que
27     #faltam
28     n2 = int(diff/prob)
29     vec2 = amostra(n2)
30     s = concatenate((vector,vec2))
31     #gera histograma
32     nb, bins, patches = hist(s, bins=50,
33                               normed=0)
34     xlabel('Amostra')
35     ylabel('Probabilidade')
36     title('Histograma de s: n=%s % n')
37     savefig('envelope.png', dpi=400)
38     show()
39     return s
40
41
42
43
44
45
46
47
48
def testRS():
    """
    Esta funcao testa o modulo
    """
    n=100000
    sample=amostra(n)
    eficiencia(sample,n)

if __name__ == '__main__':
    testRS()

```

Aplicando ao Teorema de Bayes

Podemos aplicar o conceito de amostragem por rejeição a problemas de inferência Bayesiana. Frequentemente, em inferência Bayesiana, conhecemos bem a distribuição *a priori*, a ponto de poder amostrá-la e, também, possuímos dados com os quais podemos construir a função de verossimilhança.

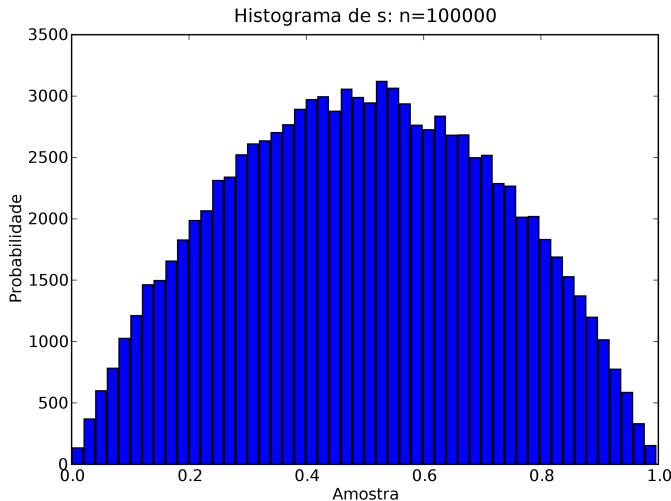


Figura 9.3: Resultado da amostragem por envelope de uma distribuição beta(1,1)

De posse destes elementos podemos invocar a técnica de amostragem por rejeição para nos ajudar a amostrar da distribuição posterior. Dado que:

$$\pi(\theta) = p(\theta)L(\theta; x)$$

temos que, sendo L_{Max} o valor máximo da nossa função de verossimilhança,

$$\pi(\theta) \leq p(\theta)L_{Max}$$

,

$$\frac{\pi(\theta)}{L_{Max}p(\theta)} = \frac{p(\theta)L(\theta; x)}{L_{Max}p(\theta)} = \frac{L(\theta; x)}{L_{Max}}$$

Assim sendo, podemos utilizar o método do envelope, amostrando da distribuição *a priori* e aceitando as amostras com probabilidade igual a $\frac{L(\theta;x)}{L_{Max}}$.

Listagem 9.4: Amostragem da Posterior Bayesiana por rejeição.

```

1 # Disponivel no pacote de programas como:
2     postrej.py
3 from numpy import *
4 from numpy.random import *
5 from pylab import *
6
7 def Likeli(data, limits, nl):
8     n = len(data) # Numero de amostras
9     data = array(data)
10    (ll, ul) = limits #limites do espaço de
11        params.
12    step = (ul-l1)/float(nl)
13    res = [] #lista de resultados
14    sd = std(data) #DP dos dados
15    for mu in arange(ll, ul, step):
16        res.append(exp(-0.5*sum(((data-mu) /
17            sd)**2)))
18    lik = array(res)/max(array(res)) # Verossimilhança
19    return lik
20
21 def amostra(n, data, plotted=0):
22     x=uniform(0,1,n) #suporte
23     limits = 0,1
24     L=Likeli(data, limits, n)
25     fx=6*x*(1-x) # priori ,beta(2,2)
26     s=compress(L[:len(x)]<fx,x) #Rejeicao
27     if not plotted:
28

```

```
25     p1 = scatter(x,fx)
26     p2 = plot(sort(x),L)
27     legend([p1,p2],[ 'Priori' , ,
28               'Verossimilhança'])
29
30   def eficiencia(vector,n, data):
31     l = len(vector)
32     prob = l/float(n)
33     diff = n-l
34     n2 = int(diff/prob)
35     vec2 = amostra(n2,data, plotted=1)
36     s = concatenate((vector,vec2))
37   return s,prob
38
39 def main():
40   n=90000
41   data = uniform(0,1,3)
42   sample=amostra(n, data)
43   s,prob = eficiencia(sample,n, data)
44   figure(2) #gera histograma
45   hist(s, bins=50, normed=1)
46   xlabel('x')
47   text(0.8,1.2, 'Eficiencia:%s %round(prob
48           ,2))')
49   ylabel('frequencia')
50   title('Posterior: n=%s %% n')
51   savefig('rejeicao.png',dpi=400)
52   show()
53   return s
54
55 if __name__ == '__main__':
56   main()
```

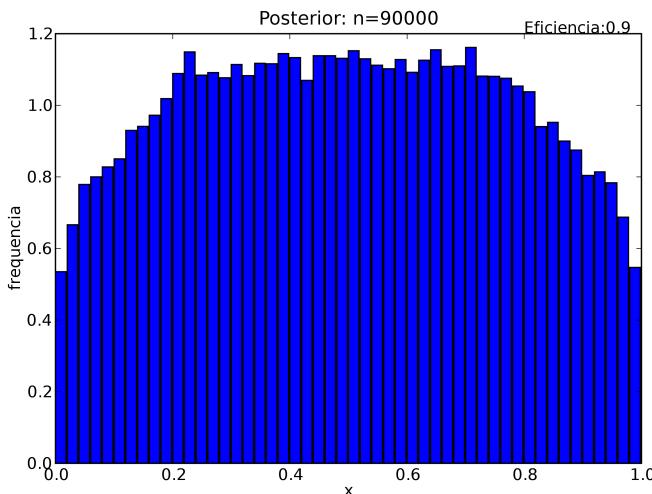


Figura 9.4: Amostragem por rejeição da posterior Bayesiana.

Na linha 22 da listagem 9.4, vemos a rejeição de acordo com a verossimilhança. Note que a verossimilhança é calculada a partir de um modelo normal enquanto que a distribuição *a priori* apresenta uma distribuição beta(2,2).

A figura 9.4 mostra o resultado da amostragem da posterior, junto com a eficiência do processo(canto superior direito).

9.5 Cadeias de Markov

Cadeias de Markov são um tipo particular de processo estocástico no qual o estado atual do sistema depende apenas do estado imediatamente anterior. Um exemplo muito conhecido de um processo

de markov é o modelo auto-regressivo de primeira ordem:

$$Z_t = \alpha Z_{t-1} + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma^2)$$

O processo apresentado acima, à medida em que o número de iterações cresce, converge para uma distribuição $\pi(z)$. Esta distribuição é denominada distribuição estacionária da cadeia. Entretanto, a cadeia necessita de um determinado número de iterações para alcançar $\pi(z)$. Se desejamos fazer inferências sobre $\pi(z)$, precisamos descartar os resultados iniciais da cadeia. Este período necessário para a convergência, é denominado de “burn-in”.

Listagem 9.5: Cadeia de Markov com kernel de transição exponencial

```

1  #-*-coding: latin-1-*-
2  # Disponivel no pacote de programas como:
3  # markov.py
4  #gerando uma cadeia de markov
5  from numpy.random import *
6  import random as rn
7  from pylab import *
8
9  n=10000
10 alpha=0.99
11 x=zeros(10000, Float)
12 for i in xrange(1,n):
13     x[ i]=alpha*x[ i-1] + exponential(1,1) [0]
14 # plotting
15 subplot(211)
16 title('Processo de Markov')
17 plot(x)
18 xlabel(u'Iterações')
19 ylabel('x')
20 subplot(212)

```

```

20 hist(x, bins=50)
21 xlabel('x')
22 ylabel(u'frequênciā')
23 savefig('markov.png', dpi=400)
24 show()

```

Executando o código apresentado na listagem 9.5 obtém-se a figura 9.5. Note o período até a convergência. Como exercício, refaça o histograma eliminando os valores do período “burn-in”, e compare com o apresentado na figura 9.5.

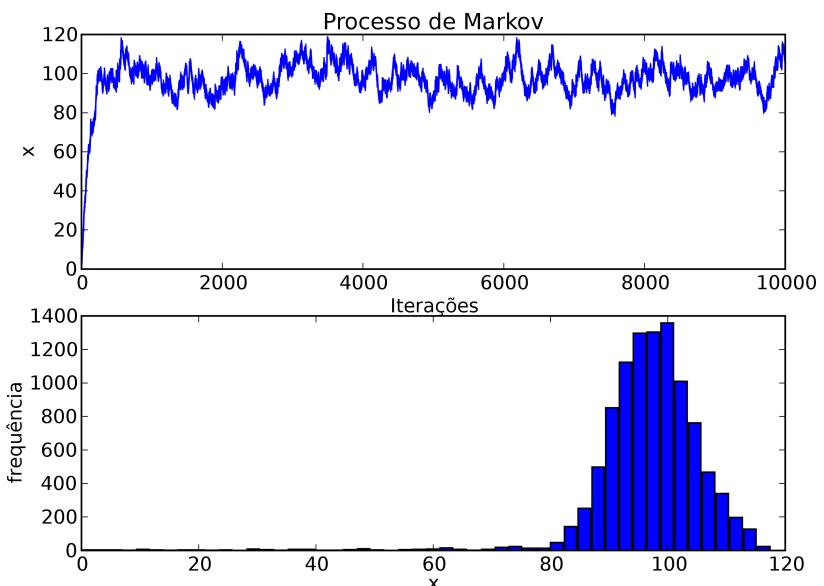


Figura 9.5: Cadeia de Markov com kernel de transição exponencial.

9.6 MCMC (Markov Chain Monte Carlo)

Nós já discutimos algumas ferramentas básicas da simulação estocástica, como o método de Monte Carlo e as cadeias de Markov. Agora vamos ver como estes dois métodos podem ser combinados para dar origem ao famoso método MCMC.

As técnicas aqui apresentadas, são de grande utilidade na estatística, particularmente na inferência Bayesiana.

O MCMC é uma classe de algoritmos que tem como objetivo construir cadeias de Markov cuja distribuição estacionária seja idêntica a de uma distribuição alvo. Após alcançarem esta distribuição estacionária, as cadeias podem ser usadas para se obter amostras da distribuição alvo.

As cadeias de Markov utilizadas na maioria dos algoritmos de MCMC, percorrem o espaço amostral utilizando-se de alguma forma de passeio aleatório. Portanto, podem levar muito tempo para cobrir o espaço desejado. Isto se deve ao fato de que a cadeia pode voltar atrás percorrendo partes do espaço já percorridas. A seguir descreveremos e implementaremos dois dos mais utilizados métodos de MCMC: os amostradores de Metropolis-Hastings e de Gibbs.

O Amostrador de Metropolis-Hastings

Este amostrador consiste em gerar amostras sequenciais de forma a aproximar uma distribuição da qual não temos meios de amostrar diretamente. Este algoritmo pode amostrar de qualquer distribuição de probabilidade $p(x)$, desde que a densidade em x possa ser calculada.

Como qualquer cadeia de markov, o amostrador começa em um dado valor arbitrário para a variável x e prossegue em um passeio aleatório, onde cada deslocamento é amostrado de uma distribuição de “Propostas”. Cada proposta(x') é avaliada de acordo com a seguinte regra: u é amostrado de uma distribuição uniforme, $U(0, 1)$.

Se

$$u < \frac{p(x')}{p(x_t)}$$

x' se torna x_{t+1} , caso contrário $x_{t+1} = x_t$.

Listagem 9.6: Amostrador de Metropolis-Hastings

```

1 #--*-encoding: latin-1-*-
2 # Disponível no pacote de programas como: MH
3 .py
4 from numpy import *
5 from scipy.stats import *
6
7 class Amostrador:
8     def __init__(self, alvo, proposta):
9         """
10             Inicializa o amostrador
11             alvo: dados definindo densidade alvo
12             proposta: objeto gerador de
13                 propostas.
14         """
15         self.alvo = kde.gaussian_kde(alvo)
16         self.prop = proposta
17
18     def Run(self, n, burnin=100):
19         """
20             Roda o amostrador por n passos
21         """
22         amostra = zeros(n, float)
23         alvo = self.alvo
24         i = 0
25         while i < n-1:
26             x = amostra[i]
27             inova = self.prop.rvs()[0]
28             can = x + inova

```

```

27         aprob = min([1 , alvo . evaluate (can
28                         ) / alvo . evaluate (x) ])
29         u = uniform . rvs () [0]
30         if u < aprob :
31             amostra [ i+1] = x = can
32             i += 1
33         return amostra [ burnin : ]
34
35 if __name__ == "__main__":
36     import pylab as P
37     dados = concatenate (( norm . rvs ( size =500) ,
38                           norm . rvs (4 ,1 , size =500)))
39     a = Amostrador (dados , norm (0 ,1))
40     res = a . Run (10000)
41     P . plot ( arange (-5 ,10 ,.01) , a . alvo . evaluate
42                 ( arange (-5 ,10 ,.01)) , 'r' , lw =2)
43     P . hist (res , normed =1 , alpha =0.5)
44     P . legend ([ 'Alvo' , 'Amostra' ])
45     P . savefig ('MH. png' , dpi =400)
46     P . show ()

```

Normalmente, utiliza-se um amostrador como este para amostrar de distribuições das quais toda a informação que se tem provém de uma amostra. Portanto, a Listagem 9.6, aceita um conjunto de dados como alvo. A partir desta amostra, uma função de densidade de probabilidade (PDF) é estimada por meio da função `gaussian_kde` do Scipy. Esta função será utilizada como alvo para o amostrador.

Note que a distribuição de propostas passada para o objeto `Amostrador` é uma das classes geradoras de números aleatórios oferecidos por `scipy.stats`³. A Figura 9.6, mostra o resultado

³Estas classes podem ser chamadas diretamente com seus parâmetros como mostrado na linha 36 da listagem 9.6, retornando um objeto “frozen” com os

do amostrador após a remoção de um número de amostras do início da cadeia determinado pelo parâmetro `burnin`. Esta remoção é importante, pois a cadeia leva algum tempo para convergir para a área sob a curva alvo.

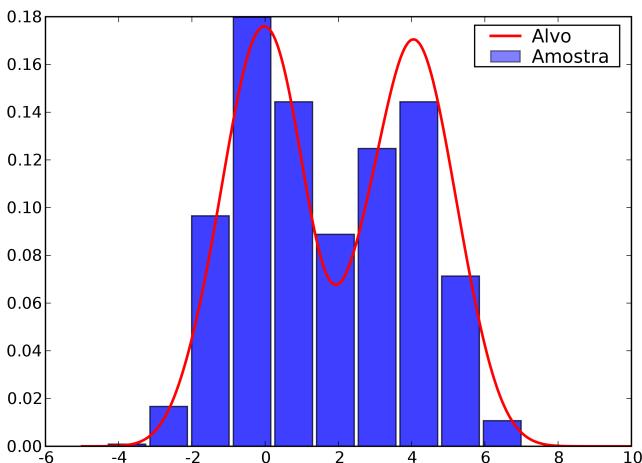


Figura 9.6: Amostrador de Metropolis-Hastings($n=10000$).

O Amostrador de Gibbs

O amostrador de Gibbs foi desenvolvido para permitir a geração de amostras a partir de distribuições conjuntas de duas ou mais variáveis. O amostrador de Gibbs é um caso especial do amostrador de Metropolis-Hastings. Na listagem 9.7, vemos uma implementa-

mesmos métodos mas com os parâmetros fixos

ção do amostrador de Gibbs, que envolve a definição de uma classe amostradora.

Listagem 9.7: Amostrador de Gibbs aplicado à uma distribuição tri-variaada

```
1 # Disponivel no pacote de programas como:  
2     gibbs.py  
3 #-*-encoding:latin-1-*  
4 from numpy import zeros, sqrt, array  
5 from numpy.random import *  
6  
6 class Amostra:  
7     def __init__(self, n, varlist, loc, scale,  
8                  cormat, burnin=1000):  
9         '''  
10            n é o tamanho da amostra  
11            varlist é a lista de funções  
12                geradoras do numpy  
13            loc e scale sao listas para cada  
14                gerador em varlist  
15            geradores testados:  
16                normal, beta, gamma, gumbel  
17                logistic, lognormal, wald, laplace  
18                '''  
19         self.n = n  
20         self.vars = varlist  
21         self.loc = loc  
22         self.scale = scale  
23         self.ro = cormat  
24         self.burnin = burnin  
25     def Run(self):  
26         results = zeros((len(self.vars), self  
27                         .n), float)  
28         dp = sqrt(1 - self.ro**2)
```

```

25     ms = self.loc
26     dps = self.scale
27     ro = self.ro
28     for i in xrange(1, self.n):
29         for n, j in enumerate(self.vars):
30             results[n, i] = j(ms[n]+ro[n,
31                               n-1]*(results[n-1, i-1]-
32                               ms[n])/dps[n-1], dps[n]*
33                               dp[n, n-1])
34     return results[:, self.burnin:]
35
36 if __name__=="__main__":
37     import pylab as P
38     n = 10000
39     varlist = (beta, gamma, normal)
40     loc = (2., 9., 5.)
41     scale = (2., .5, 1.)
42     cormat = zeros((3,3), float)
43     #cormat = array
44     ([[1, 0.6, 0.7], [0.6, 1, 0.8], [0.7, 0.8, 1]])
45
46     s = Amostra(n, varlist, loc, scale, cormat)
47     res = s.Run()
48     s1=P.plot(res[0], res[2], ' .')#marker='p',
49                 c='y', mec='k', lw=0)
50     s2=P.plot(res[1], res[2], '+')#marker='v',
51                 c='y', mec='k', lw=0)
52     P.legend(['Beta x Normal', 'Gamma x
53               Normal'])
54     P.savefig('gibbs.png', dpi=400)
55     P.show()

```

O amostrador de Gibbs pode ser aplicado quando não se conhece a distribuição conjunta, mas a distribuição condicional de

cada variável é conhecida. No amostrador de Gibbs convencional, cada nova amostra x_{t+1}^i é calculada de forma condicional ao valor de todas outras variáveis no tempo t . No exemplo mostrado na listagem 9.7, uma variante deste método é implementado, na qual as variáveis são calculadas a partir de uma estrutura de dependência circular (Figura 9.7).

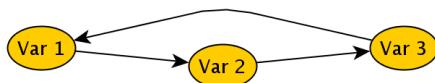


Figura 9.7: Estrutura de dependência circular utilizada na listagem 9.7.

9.7 Métodos Monte Carlo Sequenciais

Estes métodos, também chamados de filtros de partículas, visam determinar a distribuição de um processo markoviano pela análise seqüencial de observações. Seja x_k uma sequência de parâmetros desconhecidos (por exemplo o valor esperado de um sistema dinâmico) onde $k = 0, 1, 2, 3, \dots$ e $y_k \in [y_0, y_1, \dots, y_k]$ o conjunto de observações sobre X (Figura 9.9).

Os métodos sequenciais procuram estimar x_k a partir da distribuição posterior

$$p(x_k | y_0, y_1, \dots, y_k)$$

ao passo que os métodos MCMC, modelam a posterior completa:

$$p(x + 0, x_1, \dots, x_k | y_0, y_1, \dots, y_k)$$

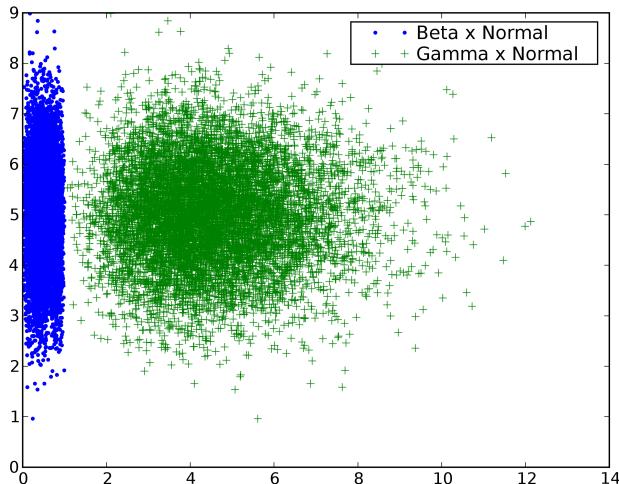


Figura 9.8: Amostrador de Gibbs aplicado a uma distribuição tri-variada.

Sampling Importance Resampling – SIR

O algoritmo de amostragem e re-amostragem por importância é um dos métodos Monte Carlo sequenciais mais utilizados. Ele procura aproximar $p(x_k | y_0, y_1, \dots, y_k)$ por meio de valores de x ponderados por uma função de importância, W que é uma aproximação da distribuição posterior de X

A descrição de um passo do SIR, é a seguinte:

1. Retiramos P amostras (\hat{x}) da distribuição *a priori* de transição de X , $p(x_k | x_{k-1})$;

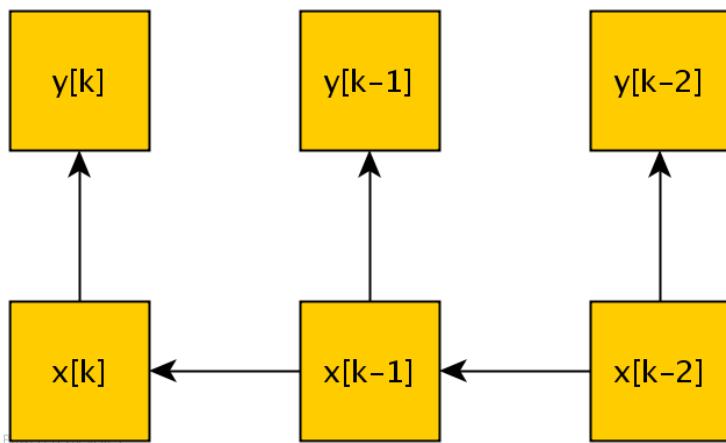


Figura 9.9: Processo markoviano e suas observações.

2. Calculamos a $p(y_k | x_k)$ (verossimilhança, no caso da distribuição alvo ser a posterior Bayesiana);
3. Calculamos $W = p(x_k | x_{k-1})p(y_k | x_k)$;
4. Normalizamos W;
5. Re-amostramos P valores de \hat{x} com probabilidades dadas por W;

A re-amostra (distribuição posterior de x_k) passa a ser a *a priori* de transição para o próximo passo.

Listagem 9.8: Filtro de partículas usando algoritmo SIR

```

1 #-*encoding: latin-1-*-
2 # Disponível no pacote de programas como:
   sir.py

```

```
3  from numpy import *
4  import like
5  import pylab as P
6  from scipy.stats import *
7
8  class SMC:
9      """
10         Monte Carlo Sequencial usando SIR
11         """
12     def __init__(self, priortype, pars, range
13                 ,resolution=512):
14         """
15             Inicializa Filtro.
16             priortype deve ser um RNG de scipy.
17             stats
18             pars são os parâmetros da priori.
19             """
20         self.priori = priortype
21         self.pars = pars
22         self.range = range
23         self.res = (range[1]-range[0])*1./
24             resolution
25         self.likelihood = None
26         self.posterior=array([])
27
28     def __call__(self,datum):
29         sc = self.pars[1]
30         m = self.range[0]
31         M = self.range[1]
32         step = self.res
33         sup = arange(m,M,step)
34         lik = exp(array([like.Normal(datum,i
35                         ,sc) for i in sup]))
36         self.likelihood = lik/sum(lik)
```

```
33     post = self . calcPosterior(1000)
34     return post
35
36 def calcPosterior(self , n):
37     """
38         Calcula a distribuição posterior
39     """
40     if self . posterior . any():
41         s = self . priori . rvs(size=n, loc=
42             self . pars[0] , scale=self .
43             pars[1])
44         #pdens= kde.gaussian_kde(self .
45         #posterior)
46         #s= pdens.resample(n)
47     else :
48         s = self . priori . rvs(size=n,
49             loc=self . pars[0] , scale=
50             self . pars[1])
51     m = array ([self . range[0]])
52     M = array ([self . range[1]])
53     step = self . res
54     supp = arange(m,M,step)
55     s = compress(less(s . ravel() ,M) &
56                 greater(s . ravel() ,m) ,s)
57     d = uniform . rvs(loc=0,scale=1,size=s
58                         . size)
59     if self . posterior . any():
60         w= self . priori . pdf(supp)*self .
61             likelihood
62     else :
63         w = self . priori . pdf(supp,loc=
64             self . pars[0] , scale=self . pars
65             [1])*self . likelihood
66     w = w/sum(w)
```

```

57     sx = searchsorted(supp, s)
58     w = w[sx-1]
59     w = w.ravel()
60     post = compress(d < w, s)
61     post = post.ravel()
62     self._posterior = post
63     return post
64
65 if __name__ == "__main__":
66     pf = SMC(norm, (3, 3), range=(-10, 15))
67     data = 2 * sin(arange(0, 7, .2)) + 2 * ones(10)
68     sup = arange(-10, 15, 25 / 512.)
69     priori = norm.pdf(sup, loc=3, scale=1)
70     P. ion()
71     lin, = P. plot(sup, priori)
72     pt, = P. plot([data[0]], [0.05], 'o')
73     P. grid()
74     for i in data:
75         est = pf([i])
76         lin.set_ydata(kde.gaussian_kde(est).evaluate(sup))
77         pt.set_xdata([i])
78     P. draw()
79     print(i, compress(pf.likelihood == max(
80                     pf.likelihood), sup), median(est))

```

Na Listagem 9.8, vemos uma implementação de um filtro de partículas utilizando o algoritmo SIR. O filtro é definido como uma classe. Nesta classe definimos o método `__call__`, característico de funções. Desta forma, uma vez criado o objeto (instância da classe `SMC`), ele poderá ser chamado diretamente pelo nome. Se o objeto não possuísse tal método, teríamos que nos utilizar de um de seus métodos para acionar sua “maquinaria”.

Parametrizamos o objeto `SMC` com uma distribuição *a priori*

dada pela classe `norm` do módulo `scipy.stats` com média $\mu = 3$ e desvio padrão $\sigma = 3$. Também precisamos especificar os limites de busca do algoritmo, o que é feito através do argumento `range` passado durante a inicialização. Neste exemplo as funções de cálculo das verossimilhanças foram implementadas de um módulo separado chamado `like.py`. Este módulo está no pacote de programas disponível no website deste livro.

A convergência dos métodos sequenciais, depende fortemente da qualidade do algoritmo de amostragem da *priori*, uma vez que estas amostras formam a “matéria-prima” do algoritmo SIR.

9.8 Funções de Verossimilhança

Ser capaz de calcular funções de verossimilhança é essencial quando se deseja utilizar a inferência Bayesiana. A verossimilhança é o termo da fórmula de Bayes que representa a evidência contida em um conjunto de dados, e que é utilizada para atualizar nosso conhecimento *a priori* do problema, de forma a gerar uma nova descrição para suas variáveis, denominada de distribuição posterior conjunta.

Definição da função de verossimilhança

Variável Discreta

Seja Θ um espaço de parâmetros. Seja $\{p_\theta; \theta \in \Theta\}$ uma família parametrizada de modelos probabilísticos para um vetor de variáveis discretas $X = (X_1, \dots, X_n)$. A função de verossimilhança para (x_1, \dots, x_n) é dada por:

$$L(\theta | x_1, \dots, x_n) \triangleq p_\theta(x_1, \dots, x_n)$$

e é uma função de θ .

A função de verossimilhança nos dá a probabilidade de que $(X_1, \dots, X_n) = (x_1, \dots, x_n)$, ou seja um determinado conjunto de

dados será amostrado em uma realização do processo gerador, para cada valor possível de θ . Matematicamente, a função de verossimilhança é apenas uma notação diferente para a função de probabilidade. Contudo, conceitualmente existe uma distinção importante: Quando se estuda a função de probabilidade, fixa-se o parâmetro e obtém-se a variação da probabilidade em função do valor da variável, x . Quando estudamos a verossimilhança, consideramos o valor de x fixo e variamos o parâmetro, obtendo a verossimilhança de cada valor. Para uma única observação de uma variável discreta, a verossimilhança é igual à probabilidade de se observar aquele valor. Por exemplo: Seja P_p uma distribuição Bernoulli com parâmetro p , $0 < p < 1$. Uma variável aleatória com uma distribuição Bernoulli, só pode assumir dois valores, 0 e 1.

Para $x = 0$,

$$L(p | 0) = 1 - p$$

Para $x = 1$,

$$L(p | 1) = p$$

Um pouco de álgebra, nos permite combinar ambas as fórmulas acima escrevendo:

$$L(p | x) = p^x(1 - p)^{1-x}, \quad x \in \{0, 1\}$$

Podemos visualizar a forma da verossimilhança com quatro linhas de código:

Listagem 9.9: Função de verossimilhança para x igual a 0.

```
1 >>> from pylab import *
2 >>> L=[1-p for p in arange(0,1,.01)]
3 >>> plot(arange(0,1,.01),L)
4 >>> show()
```

Para múltiplas observações independentes, a função de verossimilhança ficaria assim:

$$\begin{aligned} L(p | x_1, \dots, x_n) &= [p^{x_1}(1 - p)^{1-x_1}][p^{x_2}(1 - p)^{1-x_2}] \dots [p^{x_n}(1 - p)^{1-x_n}] \\ &= p^{\sum_i^n x_i}(1 - p)^{n - \sum_i^n x_i} \end{aligned} \quad (9.1)$$

Isto dito, vamos a um exemplo científico mais concreto: um pesquisador deseja comparar duas sequências de DNA, buscando quantificar a homologia entre elas, ou seja, em quantas posições possuem nucleotídeos idênticos. Sejam as duas sequências:

*ATTAGCCCTTGGAACATCCC
ATGAGCTCTGGTTAAGACCC*

Assumindo independência entre os *loci*, podemos representar esta comparação como uma variável Bernoulli (X_i) que assume valor 1 para *loci* com letras iguais e 0 para *loci* com letras distintas. Vamos resolver este problema aplicando a equação 9.1.

Listagem 9.10: Calculando a verossimilhança da comparação de duas sequências de DNA

```

1 #*-coding: latin-1-*-
2 # Disponivel no pacote de programas como:
   lik1.py
3 from pylab import *
4 a ="ATTAGCCCTTGGAACATCCC"
5 b ="ATGAGCTCTGGTTAAGACCC"
6 n = len(a)
7 # Comparando as sequências:
8 comp = [ int(a[i]==b[i]) for i in range(len(a)) ]
9 # Contando número de correspondências
10 soma = sum(array(comp))
11 # Calculando L
12 L = [p**soma*(1-p)**(n-soma) for p in arange(0,1,.01)]
13 plot(arange(0,1,.01), L)
14 savefig('lik1.png', dpi=400)
15 show()

```

A figura 9.10 mostra a função de verossimilhança gerada pela listagem 9.10. Mais adiante usaremos este resultado em outro exemplo.

Analisando o código da listagem 9.10, utilizamos a técnica de *list comprehension* para comparar as duas sequências *locus* a *locus* retornando 1 quando fossem iguais e 0, caso contrário. Como a comparação retorna verdadeiro (`True`) ou falso (`False`), aplicamos a função `int` para transformar estas saídas em 1 e 0, respectivamente. Na linha 9, convertemos a lista de resultados da comparação em vetor para poder somar os seus elementos com `sum`. Na linha 11, usamos técnica similar à da linha 7 para calcular a função de verossimilhança.

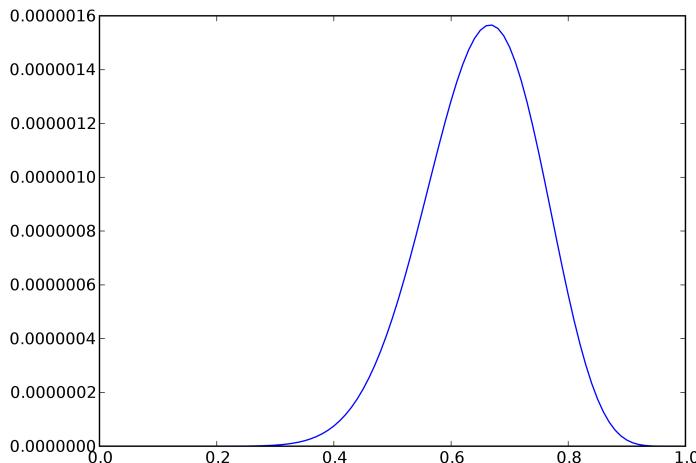


Figura 9.10: Função de verossimilhança para a comparação de sequências de DNA.

Variável Contínua

A função de verossimilhança para variáveis aleatórias contínuas é similar ao caso discreto apresentado acima. Para uma variável contínua, substituímos a função de probabilidade da definição discreta pela função de densidade de probabilidade. Se $g_\theta(x_1, \dots, x_n), \theta \in \Theta$, representa uma família de funções de densidade de probabilidade conjuntas para (X_1, \dots, X_n) , então

$$L(\theta | x_1, \dots, x_n) = \prod_{i=1}^n f_\theta(x_i)$$

se X_1, \dots, X_n forem independentes e identicamente distribuídas.

A título de exemplo, vamos escrever a função de verossimilhança para uma variável aleatória normal X com média μ e variância 1. Neste caso, μ é o nosso parâmetro θ . Dado que a densidade de X_i é $(2\pi)^{-1/2}e^{-(x-\mu)^2/2}$, para $n=3$ temos

$$L(\mu | x_1, x_2, x_3) = (2\pi)^{3/2} e^{1/2 \sum_1^3 (x_i - \mu)^2}. \quad (9.2)$$

Como podemos ver pelas equações 9.2 e 9.1, a verossimilhança é uma família de funções que difere por um fator que não depende de θ . Portanto, apenas seu valor relativo (forma) nos interessa.

Log-verossimilhança

Na função de verossimilhança, o termo que não depende de θ , depende de n (o tamanho amostral), com isso o valor da verossimilhança facilmente torna-se extremo⁴. Devido a esta característica da função de verossimilhança, frequentemente utiliza-se o seu logaritmo. Vamos plotar a log-verossimilhança calculada na listagem⁵ 9.10.

⁴Se isto não é óbvio para você, utilize seus conhecimentos de Python para investigar numericamente este fato.

⁵Modifique a listagem anterior incluindo estas linhas

Listagem 9.11: Log-verossimilhança

```

1 L[0]=L[1] #removendo o zero do início da
    lista
2 plot(arange(0,1,.01), log(array(L)))
3 show()

```

Compare agora os dois gráficos (?? e 9.11). Preste atenção aos valores do eixo y. A função mudou de forma, porém preservou uma coisa muito importante: o seu máximo continua com a mesma abscissa. Isto nos leva a uma aplicação muito importante da função verossimilhança: a estimativa de parâmetros por máxima verossimilhança.

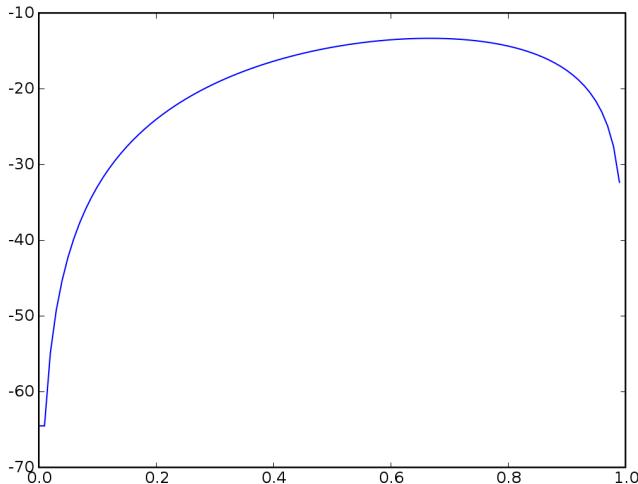


Figura 9.11: Log-verossimilhança da função descrita anteriormente.

Vimos na listagem 9.10 que, quando calculamos a função de verossimilhança sobre dados obtidos experimentalmente, ela representa a “probabilidade” de cada valor de parâmetro no eixo x ter gerado aquele conjunto de dados. Logo, se desejamos estimar este parâmetro a partir dos dados, devemos escolher o valor que maximiza a função de verossimilhança. Felizmente, este valor é o mesmo para a log-verossimilhança.

Voltando ao problema da comparação de duas sequências de DNA, vamos maximizar a função de verossimilhança para estimar p (ver listagem 9.12).

Listagem 9.12: Maximizando a verossimilhança

```

1 MLE = compress( equal(L, max(L)) , arange
                  (0, 1, .01) )
2 print MLE
3 [ 0.67]

```

Na linha 1 da listagem 9.12, utiliza-se uma função proveniente do módulo `numpy`⁶. Esta função (`compress`) nos retorna os elementos do segundo argumento `arange(0, 1, .01)`, nas posições em que o vetor de condições, `equal(L, max(L))`, for verdadeiro. Trocando em miúdos: nos retorna o valor do parâmetro p para o qual o valor da verossimilhança é máximo. As funções `max` e `equal` também são provenientes do módulo `numpy`.

9.9 Inferência Bayesiana com Objetos.

Na visão Bayesiana de mundo, nosso conhecimento sobre o mundo é representado por nossas crenças que, por sua vez, são representadas por variáveis aleatórias com suas distribuições de probabilidade (PDFs). Como nosso conhecimento do mundo não é perfeito, estas distribuições não têm precisão infinita, ou seja, não podem ser

⁶Importado, neste exemplo, pelo módulo `pylab`.

representadas por um simples número. Por exemplo, minha altura não é $1,73m$ mas uma distribuição de valores em torno deste, visto que depende de vários fatores tais como postura, quantidade de cabelo, etc.

Naturalmente, nossas crenças podem ser modificadas. A cada vez que nós nos expomos a novas evidências(dados) a respeito de um fato, podemos reforçar nossa crença original (aumentando sua precisão). Ou podemos ficar mais confusos, se os dados não concordam com nossa crença original, e esta última perderá precisão.

O objetivo desta seção, é construir um objeto que emule as características de variável aleatória Bayesiana, facilitando-nos a realização das operações mencionadas acima.

Em computação se desejarmos representar uma distribuição $p(x)$, teremos que fazê-lo por meio de uma função (se esta função for conhecida e computável) que recebe x como argumento e na sua probabilidade, ou por alguma outra estrutura de dados que combine todos valores de x e suas respectivas probabilidades, como por exemplo um dicionário. Naturalmente esta última representação se presta apenas a uma gama bastante limitada de variáveis.

```
1 >>> # Distribuição de probabilidades de um
          dado cúbico
2 >>> dado =
          {1:1/6., 2:1/6., 3:1/6., 4:1/6., 5:1/6., 6:1/6.}
```

Listagem 9.13: Classe Variável Aleatória Bayesiana – Requisitos.

```
3 # copyright 2007 Flavio Codeco Coelho
4 # Licensed under GPL v3
5 from numpy import *
6 import like, sys
```

Como toda variável Bayesiana, nossa classe deve possuir uma distribuição *a priori*, neste exemplo definida de forma paramétrica.

Deve também conter dados referentes a ela, e uma distribuição posterior. Naturalmente deve possuir os métodos para atualizar a sua distribuição *a priori*, transformando-a em posterior.

Listagem 9.14: Classe Variável Aleatória Bayesiana – Inicialização.

```

8 from scipy.stats import *
9
10 class BayesVar:
11     """
12         Bayesian random variate.
13     """
14     def __init__(self, priortype, pars, range
15                 , resolution=512):
16         """
17             Inicializa variável aleatória.
18             Adquire métodos da classe priortype
19
20             priortype deve ser um RNG de scipy.
21             stats
22             pars são os parâmetros da priori.
23         """
24         self.priorn = priortype.name
25         self._flavorize(priortype(*pars),
26                          priortype)
27         self.pars = pars
28         self.range = range
29         self.res = (range[1]-range[0])*1./
30                     resolution
31         self.likefun = self._Likelihood(self
32                                         .priorn)
33         self.likelihood = None

```

Esta classe deve solicitar estas informações na sua inicialização(Listagem 9.14). O pacote `scipy.stats` já possui classes cor-

respondendo a várias famílias paramétricas. Portanto nossa classe requer que o argumento `priortype` seja uma destas classes. Requer também os parâmetros da *a priori*, a extensão do suporte para a variável,e a resolução da representação das distribuições(número de pontos em se subdividirá o suporte).

Durante a inicialização da classe, importamos os métodos da distribuição *a priori* para se juntar aos métodos de nossa classe. Esta absorção de funcionalidade não poderia ser conseguida através de herança comum, pois não se sabe de antemão de qual classe desejamos herdá-los. Felizmente, a natureza dinâmica do Python nos ajudou neste ponto, permitindo uma herança dinâmica. Esta “herança” é feita no método `_flavorize` (Listagem 9.15).

Listagem 9.15: Classe Variável Aleatória Bayesiana – Herança dinâmica de métodos.

```

29         self . posterior = array ( [ ] )
30
31     def _ flavorize ( self , pt , ptbase ) :
32         ,
33         add methods from distribution type
34         ,
35         self . cdf = pt . cdf
36         self . isf = pt . isf
37         if isinstance ( ptbase , rv_continuous ) :
38             self . pdf = pt . pdf
39         elif isinstance ( ptbase , rv_discrete ) :
40             self . pmf = pt . pmf
41         else : sys . exit ( ' Invalid distribution
42                         object ' )

```

De acordo com o tipo de distribuição *a priori*, o método `Like-likelihood` escolhe e retorna a função de verossimilhança adequada(Listagem 9.16).

Listagem 9.16: Classe Variável Aleatória Bayesiana – Herança dinâmica de métodos.

```

100     return []
101
102     def __init__(self, typ):
103         """
104             Define família paramétrica da
105             verossimilhança.
106             Retorna função de verossimilhança.
107             typ deve ser uma string.
108         """
109         if typ == 'norm':
110             self._likelihood = lambda x: stats.norm(x[0], x[1], 1./x[2])
111         elif typ == 'expon':
112             self._likelihood = lambda x: (1./x[2])**x[0].size * exp(-(1./x[2])*sum(x[0]))

```

Durante a inicialização da variável, uma lista vazia de dados é criada. Esta lista conterá todos os conjuntos de dados independentes que forem atribuídos à variável. Isto é importante pois a teoria Bayesiana opera por atualizações sucessivas da distribuição posterior a cada novo conjunto de evidências. Uma vez gerada, a distribuição posterior fica guardada no atributo `posterior`.

Nossa classe possui ainda um método chamado `addData`, cuja função é a de armazenar os dados, e recalcular a função de verossimilhança chamando o método `_update` (Listagem 9.17).

Listagem 9.17: Classe Variável Aleatória Bayesiana – Adição de dados e cálculo da Verossimilhança.

```

42         self._likelihood = pt.ppf
43         self._rvs = pt.rvs
44     def _update(self):

```

```

45      """
46      Calculate likelihood function
47      """
48      if self.data:
49          d = self.data[-1]
50          sc = self.pars[1]
51          m = self.range[0]
52          M = self.range[1]
53          step = self.res
54          #self.likefun returns log-
55          #likelihood
56          lik = exp(array([self.likefun((d
57              , i, sc)) for i in arange(m,M,
58              step)]))
59          self.likelihood = lik/sum(lik)
60
61      def addData(self, data = []):
62          """
63              Adds dataset to variable's data
64              store
65          """

```

Os métodos `getPriorSample` e `getPriorDist` retornam amostras e a PDF da distribuição *a priori*, respectivamente(Listagem 9.18).

Listagem 9.18: Classe Variável Aleatória Bayesiana – Amostras e PDF da *a priori*.

```

63          self._update()
64
65      def getPriorSample(self, n):
66          """
67              Returns a sample from the prior
68              distribution

```

```

68      , ,
69      return self.rvs(size=n)
70
71  def getPriorDist(self):
72      """
73          Returns the prior PDF.

```

Por fim temos o método mais complexo: `getPosteriorSample` (Listagem 9.19). Este método é responsável por atualizar a distribuição posterior da variável sempre que se solicitar uma amostra. Utiliza um método de amostragem por importânci no qual se reamostra a distribuição *a priori* com um probabilidade proporcional ao produto da verossimilhança e da distribuição *a priori*. Vale notar, que se existir alguma posterior já calculada, esta é utilizada como *a priori*. Como deve ser.

Listagem 9.19: Classe Variável Aleatória Bayesiana – Gerando a posterior.

```

74      """
75      return self.pdf(arange(self.range
76                          [0], self.range[1], self.res))
76  def getPosteriorSample(self, n):
77      """
78          Return a sample of the posterior
79          distribution.
80      """
80      if self.posterior.any():# Use last
81          posterior as prior
81          k= kde.gaussian_kde(self.
82                               posterior)
82          s= k.resample(n)
83      else:
84          s = self.getPriorSample(n)
85      if self.data:

```

```

86     m = self . range [ 0 ]
87     M = self . range [ 1 ]
88     step = self . res
89     supp = arange (m,M,step )#support
90     s = compress ( less ( s . ravel () ,M) &
91                 greater ( s . ravel () ,m) ,s )#
92                 removing out-of-range
93                 samples
94     d = uniform . rvs ( loc=0,scale=1,
95                         size=len (s ))#Uniform 0-1
96                         samples
97     w = self . pdf (supp)*self .
98                 likelihood
99     w = w/sum (w) #normalizing
100                weights
101    sx = searchsorted (supp ,s )
102    w = w[ sx-1 ]#search sorted
103                returns 1-based binlist
104    post = compress (d<w, s )
105    self . posterior = post
106    return post

```

O código completo deste programa(Listagem 9.21) inclui, ainda, algumas linhas para testar a classe (Listagem 9.20). Este teste consiste em inicializar a variável com uma distribuição a priori Normal com média $\mu = 3$ e desvio padrão $\sigma = 1$. Um pequeno conjunto de observações, escolhido de forma a não concordar com a distribuição *a priori*, é utilizado para demonstrar a atualização da variável(Figura 9.12).

Listagem 9.20: Classe Variável Aleatória Bayesiana – Código de teste.

```

113 |         return lambda (x) : like . Beta (x
114 |             [ 0 ] ,x [ 1 ] ,x [ 2 ])

```

```
114  
115 if __name__ == "__main__":  
116     bv = BayesVar(norm,(3,1),range=(0,5))  
117     data = ones(20)  
118     bv.addData(data)  
119     p = bv.getPosteriorSample(200000)  
120     P.plot(arange(bv.range[0],bv.range[1],  
121                 bv.res),bv.likelihood,'ro',lw=2)  
122     P.plot(arange(bv.range[0],bv.range[1],  
123                 bv.res),bv.getPriorDist(),'g+',lw=2)  
124     P.hist(p,normed=1)  
125     P.legend(['Likelihood','Prior'])
```

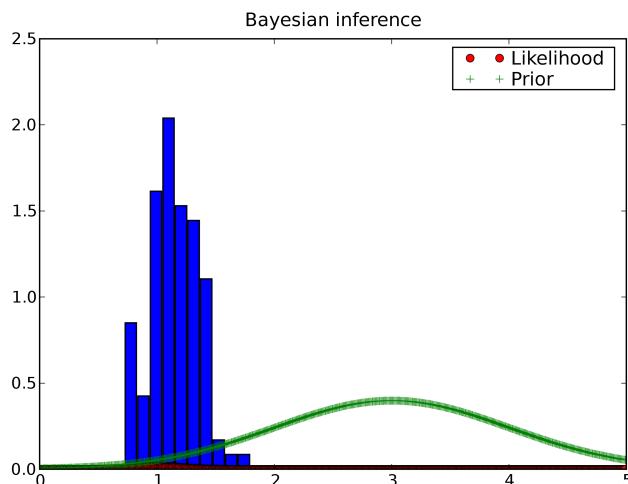


Figura 9.12: Resultado da listagem 9.21.

Listagem 9.21: Classe Variável Aleatória Bayesiana – Listagem completa

```
1  #-*encoding:latin-1*-
2  # Disponivel no pacote de programas como:
3  #      Bayes.py
4  # copyright 2007 Flavio Codeco Coelho
5  # Licensed under GPL v3
6  from numpy import *
7  import like, sys
8  import pylab as P
9  from scipy.stats import *
10
11 class BayesVar:
12     """
13         Bayesian random variate.
14     """
15     def __init__(self, priortype, pars, range
16                 , resolution=512):
17         """
18             Inicializa variável aleatória.
19             Adquire métodos da classe priortype
20
21             priortype deve ser um RNG de scipy.
22             stats
23             pars são os parâmetros da priori.
24
25             self.priorn = priortype.name
26             self._flavorize(priortype(*pars),
27                             priortype)
28             self.pars = pars
29             self.range = range
30             self.res = (range[1]-range[0])*1./
31                     resolution
```

```
26         self.likefun = self._Likelihood(self
27             .priorn)
28         self.likelihood = None
29         self.data = []
30         self.posterior=array([])
31
32     def __flavorize(self, pt, ptbase):
33         """ add methods from distribution type
34         """
35         self.cdf = pt.cdf
36         self.isf = pt.isf
37         if isinstance(ptbase, rv_continuous):
38             self.pdf = pt.pdf
39         elif isinstance(ptbase, rv_discrete):
40             self.pdf = pt.pmf
41         else: sys.exit('Invalid distribution
42             object')
43         self.ppf = pt.ppf
44         self.rvs = pt.rvs
45     def __update(self):
46         """
47             Calculate likelihood function
48         """
49         if self.data:
50             d = self.data[-1]
51             sc = self.pars[1]
52             m = self.range[0]
53             M = self.range[1]
54             step = self.res
55             #self.likefun returns log-
56             #likelihood
57             lik = exp(array([self.likefun((d
58                 , i, sc)) for i in arange(m,M,
```

```

56                         step)))
57         self.likelihood = lik/sum(lik)
58
59     def addData(self, data = []):
60         """
61             Adds dataset to variable's data
62             store
63         """
64         self.data.append(array(data))
65         self._update()
66
67     def getPriorSample(self, n):
68         """
69             Returns a sample from the prior
70                 distribution
71         """
72         return self.rvs(size=n)
73
74     def getPriorDist(self):
75         """
76             Returns the prior PDF.
77         """
78         return self.pdf(arange(self.range
79                             [0], self.range[1], self.res))
80
81     def getPosteriorSample(self, n):
82         """
83             Return a sample of the posterior
84                 distribution.
85         """
86         if self.posterior.any():# Use last
87             posterior as prior
88             k= kde.gaussian_kde(self.
89                                 posterior)
90             s= k.resample(n)
91
92

```

```

83     else:
84         s = self.getPriorSample(n)
85     if self.data:
86         m = self.range[0]
87         M = self.range[1]
88         step = self.res
89         supp = arange(m,M,step)#support
90         s = compress(less(s.ravel(),M) &
91                     greater(s.ravel(),m),s)#
92                     removing out-of-range
93                     samples
94         d = uniform.rvs(loc=0,scale=1,
95                         size=len(s))#Uniform 0-1
96                         samples
97         w = self.pdf(supp)*self.
98             likelihood
99         w = w/sum(w) #normalizing
100            weights
101         sx = searchsorted(supp,s)
102         w = w[sx-1]#search sorted
103             returns 1-based binlist
104         post = compress(d<w,s)
105         self.posterior = post
106         return post
107     else:
108         return []
109
110
111
112     def Likelihood(self,typ):
113         ;;,
114         Define familia paramétrica da
115             verossimilhança.
116         Retorna função de verossimilhança.
117         typ deve ser uma string.
118         ,,,
```

```

108     if typ == 'norm':
109         return lambda(x): like.Normal(x
110                                     [0],x[1],1./x[2])
111     elif typ == 'expon':
112         return lambda(x):(1./x[2])**x
113                                     [0].size*exp(-(1./x[2])*sum(
114                                     x[0]))
115     elif typ == 'beta':
116         return lambda(x): like.Beta(x
117                                     [0],x[1],x[2])
118
119 if __name__=="__main__":
120     bv = BayesVar(norm,(3,1),range=(0,5))
121     data = ones(20)
122     bv.addData(data)
123     p = bv.getPosteriorSample(200000)
124     P.plot(arange(bv.range[0],bv.range[1],
125                   bv.res),bv.likelihood,'ro',lw=2)
126     P.plot(arange(bv.range[0],bv.range[1],
127                   bv.res),bv.getPriorDist(),'g+',lw=2)
128     P.hist(p, normed=1)
129     P.legend(['Likelihood','Prior'])
130     P.title('Bayesian inference')
131     P.savefig('bayesvar.png',dpi=400)
132     P.show()

```

9.10 Exercícios

1. Compare a acurácia do amostrador de Metropolis-Hastings com o amostrador oferecido pelo método `resample` do objeto retornado por `gaussian_kde`. Teste com diferentes formas de distribuição. O método `resample` é uma mera amostragem com reposição.

2. Verifique a importância do tamanho amostral na atualização da distribuição posterior de uma variável, utilizando o programa da listagem 9.21
3. Experimente diferentes valores para a variância da distribuição *a priori* do exemplo 9.8, e verifique o efeito na convergência do algoritmo.
4. Utilize o método do hipercubo latino ilustrado na listagem 9.1 para gerar amostras das distribuições *a priori* nos exemplos 9.8 e 9.21 e avalie o seu impacto na qualidade das estimativas.

Apêndices

Introdução ao Console Gnu/Linux

Guia de sobrevivência no console do Gnu/Linux

O console Gnu/Linux é um poderoso ambiente de trabalho, em contraste com a interface limitada, oferecida pelo sistema operacional DOS, ao qual é comumente comparado. O console Gnu/Linux tem uma longa história desde sua origem no “Bourne shell”, distribuído com o Sistema operacional(SO) UNIX versão 7. Em sua evolução, deu origem a algumas variantes. A variante mais amplamente utilizada e que será objeto de utilização e análise neste capítulo é o Bash ou “Bourne Again Shell”. Ao longo deste capítulo o termo console e shell serão utilizados com o mesmo sentido, ainda que, tecnicamente não sejam sinônimos. Isto se deve à falta de uma tradução mais adequada para a palavra inglesa “shell”.

Qual a relevância de um tutorial sobre shell em um livro sobre computação científica? Qualquer cientista com alguma experiência em computação está plenamente consciente do fato de que a maior parte do seu trabalho, se dá através da combinação da funcionalidade de diversos aplicativos científicos para a realização de tarefas científicas de maior complexidade. Neste caso, o ambiente de trabalho é chave para agilizar esta articulação entre aplicativos. Este capítulo se propõe a demonstrar, através de exemplos, que o GNU/Linux é um ambiente muito superior para este tipo de atividade, se comparado com Sistemas Operacionais voltados

principalmente para usuários leigos.

Além do Console e sua linguagem (bash), neste capítulo vamos conhecer diversos aplicativos disponíveis no sistema operacional Gnu/Linux, desenvolvidos para serem utilizados no console.

A linguagem BASH

A primeira coisa que se deve entender antes de começar a estudar o shell do Linux, é que este é uma linguagem de programação bastante poderosa em si mesmo. O termo Shell, cápsula, traduzido literalmente, se refere à sua função como uma interface entre o usuário e o sistema operacional. A shell nos oferece uma interface textual para invocarmos aplicativos e lidarmos com suas entradas e saídas. A segunda coisa que se deve entender é que a shell não é o sistema operacional, mas um aplicativo que nos facilita a interação com o SO.

O Shell não depende de interfaces gráficas sofisticadas, mas comumente é utilizado através de uma janela, do conforto de uma interface gráfica. Na figura 1, vemos um exemplo de uma sessão do bash rodando em uma janela.

Alguns Comando Úteis

ls Lista arquivos.

pwd Imprime o nome do diretório corrente (caminho completo).

cp Copia arquivos.

mkdir Cria um diretório.

mv Renomeia ou move arquivos.

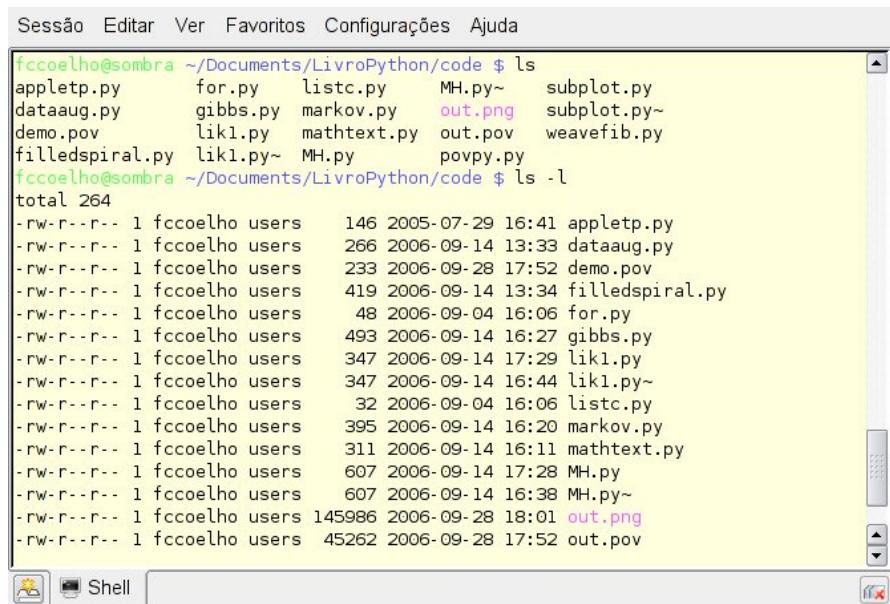
rmdir Remove um diretório.

rm Apaga arquivos (de verdade!).

Para remover recursivamente toda uma árvore de diretórios use rm -rf(cuidado!).

ln Cria links para arquivos.

A linguagem BASH



The screenshot shows a terminal window titled 'Sessão' in the KDE desktop environment. The window contains a command-line session:

```
fccoelho@sombra ~/Documents/LivroPython/code $ ls
appletp.py      for.py    listc.py    MH.py~    subplot.py
dataaug.py       gibbs.py   markov.py   out.png   subplot.py~
demo.pov        likl.py    mathtext.py out.pov   weavefib.py
filledspiral.py likl.py~  MH.py     povpy.py
fccoelho@sombra ~/Documents/LivroPython/code $ ls -l
total 264
-rw-r--r-- 1 fccoelho users 146 2005-07-29 16:41 appletp.py
-rw-r--r-- 1 fccoelho users 266 2006-09-14 13:33 dataaug.py
-rw-r--r-- 1 fccoelho users 233 2006-09-28 17:52 demo.pov
-rw-r--r-- 1 fccoelho users 419 2006-09-14 13:34 filledspiral.py
-rw-r--r-- 1 fccoelho users 48 2006-09-04 16:06 for.py
-rw-r--r-- 1 fccoelho users 493 2006-09-14 16:27 gibbs.py
-rw-r--r-- 1 fccoelho users 347 2006-09-14 17:29 likl.py
-rw-r--r-- 1 fccoelho users 347 2006-09-14 16:44 likl.py~
-rw-r--r-- 1 fccoelho users 32 2006-09-04 16:06 listc.py
-rw-r--r-- 1 fccoelho users 395 2006-09-14 16:20 markov.py
-rw-r--r-- 1 fccoelho users 311 2006-09-14 16:11 mathtext.py
-rw-r--r-- 1 fccoelho users 607 2006-09-14 17:28 MH.py
-rw-r--r-- 1 fccoelho users 607 2006-09-14 16:38 MH.py~
-rw-r--r-- 1 fccoelho users 145986 2006-09-28 18:01 out.png
-rw-r--r-- 1 fccoelho users 45262 2006-09-28 17:52 out.pov
```

The window has standard KDE window controls at the bottom: minimize, maximize, and close.

Figura 1: Konsole: janela contendo uma sessão bash (ou outra) no KDE.

cat Joga o arquivo inteiro na tela.

tail Visualiza o final do arquivo.

less Visualiza o arquivo com possibilidade de movimentação e busca dentro do mesmo.

nl Visualiza com numeração das linhas.

od Visualiza arquivo binário em base octal.

head Visualiza o início do arquivo.

xxd Visualiza arquivo binário em base hexadecinal.

CONSOLE GNU/LINUX

gv Visualiza arquivos Postscript/PDF.	locate Localiza arquivo por meio de índice criado com updatedb.
xdvi Visualiza arquivos DVI gerados pelo \TeX .	which Localiza comandos.
stat Mostra atributos dos arquivos.	whereis Localiza o binário (executável), fontes, e página man de um comando.
wc Conta bytes/palavras/líneas.	grep Busca em texto retornando linhas.
du Uso de espaço em disco.	cut Extrai colunas de um arquivo.
file Identifica tipo do arquivo.	paste Anexa colunas de um arquivo texto.
touch Atualiza registro de última atualização do arquivo. Caso o arquivo não exista, é criado.	sort Ordena linhas.
chown Altera o dono do arquivo.	uniq Localiza linhas idênticas.
chgrp Altera o grupo do arquivo.	gzip Compacta arquivos no formato GNU Zip.
chmod Altera as permissões de um arquivo.	compress Compacta arquivos.
chattr Altera atributos avançados de um arquivo.	bzip2 Compacta arquivos(maior compactação do que o gzip, porém mais lento).
lsattr Lista atributos avançados do arquivo.	zip Compacta arquivos no formato zip(Windows).
find Localiza arquivos.	diff Compara arquivos linha a linha.

Entradas e Saídas, redirecionamento e "Pipes".

comm Compara arquivos ordenados.

cmp Compara arquivos byte por byte.

md5sum Calcula checksums.

df Espaço livre em todos os discos(pendrives e etc.) montados.

mount Torna um disco acessível.

fsck Verifica um disco procurando por erros.

sync Esvazia caches de disco.

ps Lista todos os processos.

w Lista os processos do usuário.

uptime Retorna tempo desde o último boot, e carga do sistema.

top Monitora processos em execução.

free Mostra memória livre.

kill Mata processos.

nice Ajusta a prioridade de um processo.

renice Altera a prioridade de um processo.

watch Executa programas a intervalos regulares.

crontab Agenda tarefas periódicas.

Entradas e Saídas, redirecionamento e "Pipes".

O esquema padrão de entradas e saídas dos SOs derivados do UNIX, está baseado em duas idéias muito simples: toda comunicação é formada por uma sequência arbitrária de caracteres (Bytes), e qualquer elemento do SO que produza ou aceite dados é tratado como um arquivo, desde dispositivos de hardware até programas.

Por convenção um programa UNIX apresenta três canais de comunicação com o mundo externo: entrada padrão ou STDIN, saída padrão ou STDOUT e saída de erros padrão ou STDERR.

O Bash(assim como praticamente todas as outras shells) torna muito simples a utilização destes canais padrão. Normalmente, um

usuário utiliza estes canais com a finalidade de redirecionar dados através de uma sequência de passos de processamento. Como este processo se assemelha ao modo como canalizamos água para levá-la de um ponto ao outro, Estas construções receberam o apelido de “pipelines” ou tubulações onde cada segmento é chamado de “pipe”.

Devido a essa facilidade, muitos dos utilitários disponíveis na shell do Gnu/Linux foram desenvolvidos para fazer uma única coisa bem, uma vez que funções mais complexas poderiam ser obtidas combinando programas através de “pipelines”.

Redirecionamento

Para redirecionar algum dado para o STDIN de um programa, utilizamos o caracter <. Por exemplo, suponha que temos um arquivo chamado **nomes** contendo uma lista de nomes, um por linha. O comando sort < nomes irá lançar na tela os nomes ordenados alfabeticamente. De maneira similar, podemos utilizar o caracter > para redirecionar a saída de um programa para um arquivo, por exemplo.

Listagem 22: Redirecionando STDIN e STDOUT

```
1 $ sort < nomes > nomes_ordenados
```

O comando do exemplo 22, cria um novo arquivo com o conteúdo do arquivo **nomes**, ordenado.

“Pipelines”

Podemos também redirecionar saídas de comandos para outros comandos, ao invés de arquivos, como vimos anteriormente. O caractere que usamos para isto é o | conhecido como “pipe”. Qualquer linha de comando conectando dois ou mais comandos através de “pipes” é denominada de “pipeline”.

Listagem 23: Lista ordenada dos usuários do sistema.

```
1 $ cut -d: -f1 < /etc/passwd | sort
2 ajaxterm
3 avahi
4 avahi-autoipd
5 backup
6 beagleindex
7 bin
8 boinc
9 ...
```

O simples exemplo apresentado dá uma idéia do poder dos “pipelines”, além da sua conveniência para realizar tarefas complexas, sem a necessidade de armazenar dados intermediários em arquivos, antes de redirecioná-los a outros programas.

Pérolas Científicas do Console Gnu/Linux

O console Gnu/Linux extrai a maior parte da sua extrema versatilidade de uma extensa coleção de aplicativos leves desenvolvidos⁷ para serem utilizados diretamente do console. Nesta seção, vamos ver alguns exemplos, uma vez que seria impossível explorar todos eles, neste simples apêndice.

Gnu plotutils

O “GNu Ploting Utilities” é uma suite de aplicativos gráficos e matemáticos desenvolvidos para o console Gnu/Linux. São eles:

graph Lê um ou mais conjuntos de dados a partir de arquivos ou de STDIN e prepara um gráfico;

⁷Desenvolvidos principalmente pelo projeto Gnu

CONSOLE GNU/LINUX

plot Converte Gnu metafile para qualquer dos formatos listados acima;

pic2plot Converte diagramas criados na linguagem **pic** para qualquer dos formatos acima;

tek2plot Converte do formato Tektronix para qualquer dos formatos acima.

Estes aplicativos gráficos podem criar e exportar gráficos bidimensionais em treze formatos diferentes: **SVG**, **PNG**, **PNM**, **pseudo-GIF**, **WebCGM**, **Illustrator**, **Postscript**, **PCL 5**, **HP-GL/2**, **Fig** (editável com o editor de desenhos **xfig**), **ReGIS**, **Tektronix** ou **GNU Metafile**.

Aplicativos Matemáticos:

ode Integra numericamente sistemas de equações diferenciais ordinárias (EDO);

spline Interpola curvas utilizando “splines” cúbicas ou exponenciais. Pode ser utilizado como filtro em tempo real.

graph

A cada vez que chamamos o programa **graph**, ele lê um ou mais conjuntos de dados a partir de arquivos especificados na linha de comando, ou diretamente da **STDIN**, e produz um gráfico. O gráfico pode ser mostrado em uma janela, ou salvo em um arquivo em qualquer dos formatos suportados.

Listagem 24: Plotando dados em um arquivo.

```
1 $ graph -T png < arquivo_de_dados_ascii >
   plot.png
```

Se o **arquivo_de_dados_ascii** contiver duas colunas de números, o programa as atribuirá a **x** e **y**, respectivamente. Os pares

ordenados que darão origem aos pontos do gráfico não precisam estar em linhas diferentes. por exemplo:

Listagem 25: Desenhando um quadrado.

```
1 $ echo .1 .1 .1 .9 .9 .9 .1 .1 | graph  
-T X -C -m 1 -q 0.3
```

A listagem 25 plotará um quadrado com vértices em $(0.1, 0.1)$, $(0.1, 0.9)$, $(0.9, 0.9)$ e $(0.9, 0.1)$. A repetição do primeiro vértice garante que o polígono será fechado. A opção **-m** indica o tipo da linha utilizada: 1-linha sólida, 2-pontilhada, 3-ponto e traço, 4-traços curtos e 5-traços longos. A opção **-q** indica que o quadrado será preenchido (densidade 30%) com a mesma cor da linha: vermelho (**-C** indica gráfico colorido).

O programa **graph** aceita ainda muitas outras opções. Leia o manual(**man graph**) para descobrí-las.

spline

O programa funciona de forma similar ao **graph** no que diz respeito à entradas e saídas. Como todos os aplicativos de console, beneficia-se muito da interação com outros programas via “pipes”.

Listagem 26: Uso do **spline**

```
1 $ echo 0 0 1 1 2 0 | spline | graph -T X
```

Spline não serve apenas para interpolar funções, também pode ser usado para interpolar curvas em um espaço d-dimensional utilizando-se a opção **-d**.

Listagem 27: Interpolando uma curva em um plano.

```
1 echo 0 0 1 0 1 1 0 1 | spline -d 2 -a -s |  
graph -T X
```

O comando da listagem 27 traçará uma curva passando pelos pontos $(0,0)$, $(1,0)$, $(1,1)$ e $(0,1)$. A opção **-d 2** indica que

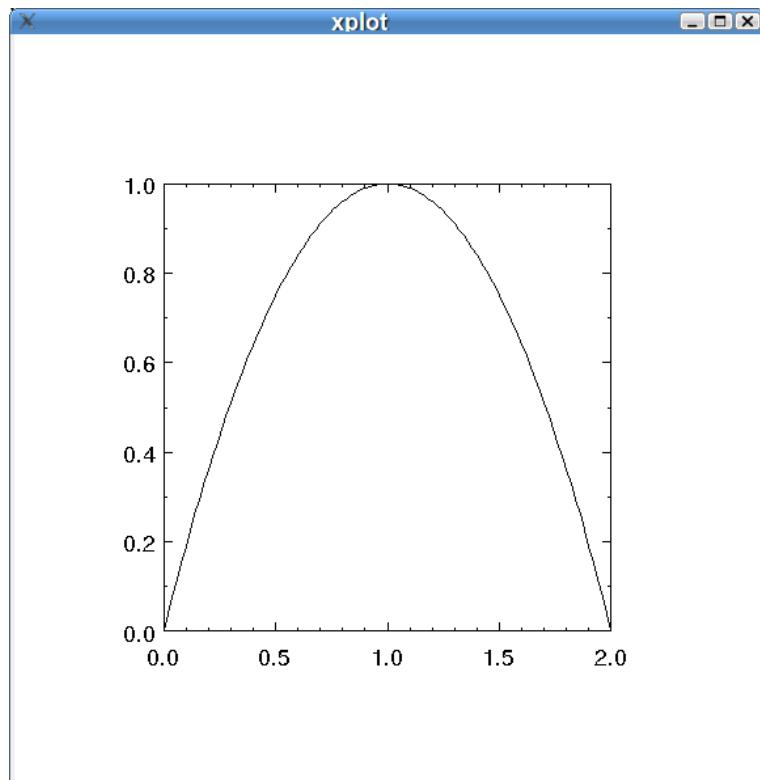


Figura 2: Usando `spline`.

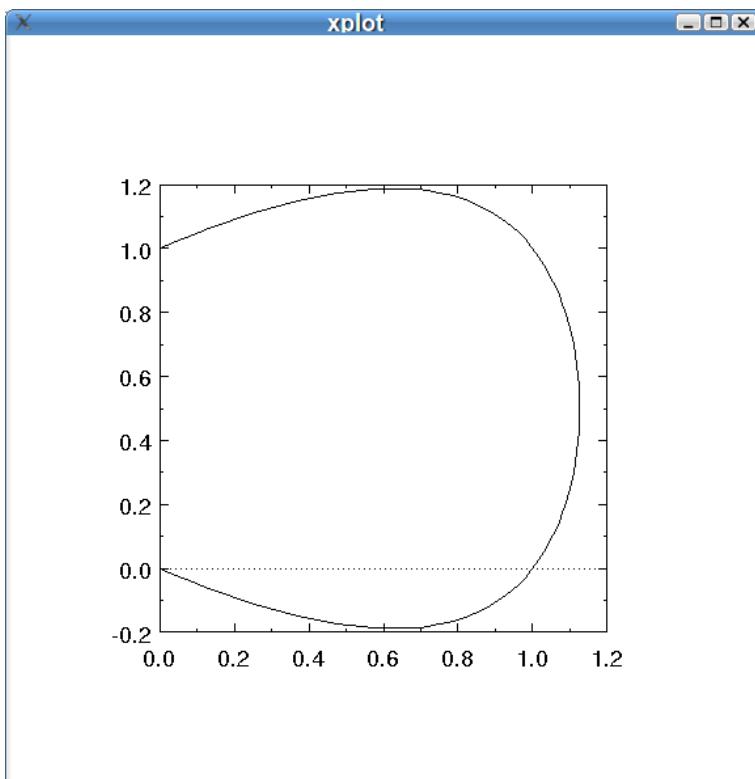


Figura 3: Interpolando uma curva em um plano.

a variável dependente é bi-dimensional. A opção **-a** indica que a variável independente deve ser gerada automaticamente e depois removida da saída (opção **-s**).

ode

O utilitário **ode** é capaz de produzir uma solução numérica de sistemas de equações diferenciais ordinárias. A saída de **ode** pode ser redirecionada para o utilitário **graph**, que já discutimos anteriormente, de forma que as soluções sejam plotadas diretamente, à medida em que são calculadas.

Vejamos um exemplo simples:

$$\frac{dy}{dt} = y(t) \quad (3)$$

A solução desta equação é:

$$y(t) = e^t \quad (4)$$

Se nós resolvemos esta equação numericamente, a partir do valor inicial $y(0) = 1$, até $t = 1$ esperaríamos obter o valor de e como último valor da nossa curva ($e^1 = 2.718282$, com 7 algarismos significativos). Para isso digitamos no console:

Listagem 28: Resolvendo uma equação diferencial simples no console do Linux.

```
1 $ ode
2 y'=y
3 y=1
4 print t ,y
5 step 0 ,1
```

Após digitar a ultima linha do exemplo 28, duas colunas de números aparecerão: a primeira correspondendo ao valor de t e a segunda ao valor de y ; a ultima linha será 1 2.718282. Como esperávamos.

Para facilitar a re-utilização dos modelos, podemos colocar os comandos do exemplo 28 em um arquivo texto. Abra o seu editor favorito, e digite o seguinte modelo:

Listagem 29: Sistema de três equações diferenciais acopladas

```
1 # O modelo de Lorenz , Um sistema de três
2 # EDOs acopladas ,
3 x' = -3*(x-y)
4 y' = -x*z+r*x-y
5 z' = x*y-z
6
7 r = 26
8 x = 0; y = 1; z = 0
9 print x, y
10 step 0, 200
```

Salve o arquivo com o nome `lorenz`. Agora digite no console a seguinte linha de comandos:

```
1 $ ode < lorenz | graph -T X -C -x -10 10 -y
   -20 20
```

E eis que surgirá a bela curva da figura 4.

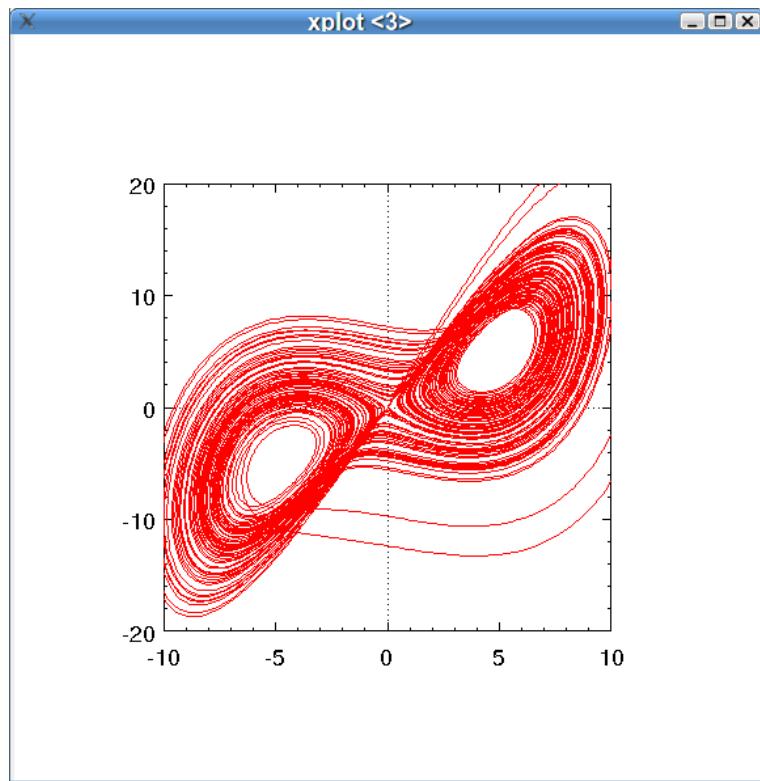


Figura 4: Atrator de Lorenz.

Índice Remissivo

- arange, 46
- aranha, 190
- array, 46
 - .shape, 46
- bancos de dados, 185
- break, 32
- cadeias de Markov, 214
- classe, 53
 - atributos, 54
 - métodos, 55
- conjuntos, 27
- Console Python, 4
- Controle de Versões, 86
 - Mercurial, 88
 - Ctypes, 105
- decoradores, 38
- dicionários, 24
 - métodos, 25
- Editores, 80
- editores, 80
- EDO, 258
- elif, 28
- else, 28
- Emacs, 81
- enumerate, 30
- equações de diferença, 145
- equações diferenciais, 148
- espaço de nomes, 10
- exceções, 32
- except, 32
- finally, 32
- for, 30
- FORTRAN, 117
- funções, 34
 - lista de argumentos variável, 36
 - passando argumentos, 36
- funções
 - argumentos opcionais, 35
- geradores, 38
- Gibbs, 219
- Gnu Nano, 81

ÍNDICE REMISSIVO

- Grafos, 169
graph, 258
- Herança, 56
- IDEs, 83
if, 28
import, 42
Inferência Bayesiana, 203
integração numérica, 148
Ipython, 73
 Comandos mágicos, 75
iteração, 29
- Java, 125
Jedit, 81
Jython, 126
- lambda, 37
LHS, 200
listas, 13
 métodos, 16
- módulo
 numpy.linalg, 46
 scipy, 46
- Módulos
 BeautifulSoup, 190
 email, 181
 mailbox, 182
 threading, 178
- módulos, 41
 numpy, 45
- Mathematica, xix
- Matlab, xix
- MCMC, 216
Mercurial, 88
Metropolis-Hastings, 217
modelagem matemática, 135
modelos dinâmicos, 145
modelos exponenciais, 142
modelos lineares, 137
monte carlo, 205
- Números complexos, 10
NetworkX, 172
numpy, 45
- objetos, 52
operadores aritméticos, 8
- pacotes, 44
Palavras reservadas, 4
pickle, 186
print, 46
pydoc, 47
Pyrex, 106
- R, xix
return, 37
- scipy, 46
Scite, 81
Shedskin, 111
sqlite, 187
SQLObject, 190
strings, 22
 formatando, 23
- try, 32

tuplas, 19
unidades, 155
unum, 159
urllib, 193
urllib2, 193
uso interativo, 5
verossimilhança, 228
weave, 100
web-spider, 190
while, 29
zip, 31

Colophon

Este livro foi formatado usando
L^AT_EX, criado por Leslie
Lamport e a classe `memoir`. O
corpo do texto utiliza fontes de
tamanho 10 Modern Roman,
criadas por Donald Knuth.
Outras fontes incluem Sans,
Smallcaps, Italic, Slanted and
Typewriter, todas da família
Computer Modern desenvolvida
por Knuth.