Computação Científica com Python

Versão 1.0

Flávio Codeço Coelho

01/02/2011

Conteúdo

CAPÍTULO 1

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 frequentemente 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 programadores 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 MATLAB, Mathematica, Maple, 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 (MATLAB, 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 sistema 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.

1.1 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 distribuido 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 Python [1] 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 continou 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 CNRI [2], o Python atingiu a versão 1.6, que foi rápidamente seguida pela versão 2.0. A partir desta versão, o Python passa a ser distribuído sob a Python License, compatível com a GPL [3], 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 (Jython [4]), e depois na própria linguagem Python (PYPY [5]){Pypy}, e na plataforma `.NET` (IronPython [6]){IronPython}.

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 e celulares até supercomputadores com processamento paralelo, passando por todas as plataformas de computação pessoal.

[1] www.python.org
[2] Corporation for National Research Initiatives
[3] GNU General Public License
[4] www.jython.org
[5] pypy.org
[6] {www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython}

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 *capext*).

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

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. A 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 execucão 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:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly. Explicit is better than implicit.
Simple is better than complex. Complex is better than complicated.
Flat is better than nested. Sparse is better than dense.
Readability counts. Special cases aren't special enough to break
the rules. Although practicality beats purity. Errors should never
pass silently. Unless explicitly silenced. In the face of
ambiguity, refuse the temptation to guess. There should be one- and
preferably only one -obvious way to do it. Although that way may
not be obvious at first unless you're Dutch. Now is better than
never. Although never is often better than \*right\* now. If the
implementation is hard to explain, it's a bad idea. If the
implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea - let's do more of those!
```

1.2 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 apresentados [7]. 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 Windows, instalem o Linux em uma máquina virtual [8]! 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 Rio de Janeiro, 2010

[7] O código fonte do exemplos está disponível na seguinte URL: http://fccoelho.googlepages.com/CCP_code.zip
[8] Recomendo o VirtualBox (www.virtualbox.org), é software livre e fantástico!

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 *cap-obj*. **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 subseqüentes. Para uma introdução mais completa à linguagem, recomendamos ao leitor a consulta a livros e outros documentos voltados especificamente para programação em Python.

2.1 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:

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

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. Palavras reservadas não podem ser utilizadas como nomes de variáveis:

```
>>> for=1
  File "<stdin>", line 1
    for=1
        ^
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 elementos do módulo *__builtins__* deve ser redefinidofootnote{Atenção, os componentes de *__builtins__*, não geram erros de sintaxe ao ser redefinidos.

O console interativo do Python possui um sistema de ajuda integrado que pode ser usado para acessar a documentação de q

7.3 The for statement

The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object: ...

2.2 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 *sec_ipython*).

Tanto no uso interativo, como na execução a partir de scripts, o interpretador espera encontrar apenas uma expressão por l

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

No cabeçalho da shell do Python, acima (listagem *ex-conspy*), 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 com extensão *.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:

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

Note que os caracteres '`\#!`' devem ser os dois primeiros caracteres do arquivo (como na listagem ex-exec):

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

print "Alô Mundo!"
```

Depois, resta apenas ajustar as permissões do arquivo para que possamos executá-lo:

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

Nota: Aqui assume-se que a codificação do seu editor de texto é '`latin1`'. O importante e casar a codificação do seu editor de texto com a especificada no início do seu script.

No exemplo da listagem ex-exec, utilizamos o comando `print` para fazer com que nosso script produzisse uma string como saída, ou seja, para escrever no stdout [1]. 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.

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

Nas listagens que se seguem, alternaremos entre a utilização de scripts e a utilização do Python no modo interativo. A presença do símbolo >>>, característico da shell do Python será suficiente para diferenciar os dois casos. Exemplos de scripts virão dentro de caixas.

2.3 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:

```
>>> 2+2 #Comentário ...
4
>>> 2*2
4
>>> 2**2
4
>>> (50-5*6)/4 #Divisão de inteiros retorna "floor": ...
5
>>> 7/3
2
>>> 7/-3
-3
>>> 7/3.
2.3333333333333335
```

2.3.1 Operadores aritméticos

Nosso primeiro exemplo numérico (Listagem ex-arit) [2], 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 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:

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

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

```
>>> x = y = z = 0
>>> x
0
>>> y
0
>>> z
0
>>> a,b,c=1,2,3
>>> a
1
>>> b
2
>>> c
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

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

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":

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

2.3.2 Números complexos

Um Número complexo para o Python, é um objeto [3]. 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:

```
>>> a=3.0+3.0j
>>> a.real
3.0
>>> a.imag
3.0
>>> abs(a)
4.2426406871192848
>>> sqrt(a.real**2 + a.imag**2)
4.2426406871192848
```

[3] Assim como os outros tipos de números.

Nomes, Objetos e Espaços de Nomes

{espaço 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:

```
>>> a=3*2**7
>>> 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 ss:dict) 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:

```
>>> maria
Traceback (most recent call last): File "stdin", line 1, in ?
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 função [1]. Ao sair da função, o dicionário local torna-se igual ao global:

```
>>> a=1
>>> len(globals().items())
4
>>> len(locals().items())
4
>>> def fun():
... a='novo valor'
... print len(locals().items())
... print a
...
>>> fun()
```

[1] Mais quaisquer variáveis explicitamente definidas como globais

```
1
novo valor
>>> print a
1
>>> len(locals().items())
5
>>> locals()
'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 acima.

CAPÍTULO 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.

4.1 Listas

As listas formam o tipo de dados mais utilizado e versátil do Python. Listas são definidas como uma sequência de valores separados por vírgulas e delimitada por colchetes:

```
>>> lista=[1, 'a', 'pe']
>>> lista
[1, 'a', 'pe']
>>> lista[0]
1
>>> lista[2]
'pe'
>>> lista[-1]
'pe'
```

Na listagem ex-lista1, 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 ex-lista1, 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 assessar 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:

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

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:

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

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:

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

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[:]`:

```
>>> lista[:]
['a', 'pe', 'que', 1]
>>> lista.append(2) #adiciona 2 ao final
['a', 'pe', 'que', 1, 2]
>>> lista.insert(2,['a','b'])
>>> lista
['a', 'pe', ['a', 'b'], 'que', 1, 2]
```

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

Note que as operações *in situ* não alocam memória extra para a operação, ou seja, a inversão ou a ordenação descritas na tabela **:tab:'metlista'**, 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 ex-adlista, 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:

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

Já na listagem ex-extlista, os elementos da segunda lista são adicionados, individualmente, ao final da lista original:

```
>>> lista.index('que')
3
>>> lista.index('a')
0
>>> lista.index('z')
Traceback (most recent call last):
```

```
File "input", line 1, in ?
ValueError: list.index(x): x not in list 'z' in lista 0
```

Conforme ilustrado na listagem ex-buslista, 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` [1] como ilustrado na listagem ex-buslista. Caso o elemento faça parte da lista, este comando retornará `1`, caso contrário retornará `0` [2].

Existem dois métodos básicos para remover elementos de uma lista: `L.remove` e `L.pop` – listagem ex-remlista. 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:

```
>>> lista.remove("que")
>>> lista
['a', 'pe', ['a', 'b'], 1, 2, 'a', 'b']
>>> lista.pop(2)
['a', 'b']
>>> lista
['a', 'pe', 1, 2, 'a', 'b']
```

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 ex-oplista.

> lista=['a', 'pe', 1, 2, 'a', 'b'] lista = lista + ['novo', 'elemento'] lista ['a', 'pe', 1, 2, 'a', 'b', 'novo', 'elemento'] lista += 'dois' lista ['a', 'pe', 1, 2, 'a', 'b', 'd', 'o', 'i', 's'] lista += ['dois'] lista ['a', 'pe', 1, 2, 'a', 'b', 'd', 'o', 'i', 's', 'dois'] li=[1,2] li*3 [1, 2, 1, 2, 1, 2]

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 ex-oplista. 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 ex-range). O comando `range` aceita 1, 2 ou três argumentos: início, fim e passo, respectivamente (ver exemplo ex-range).

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

4.2 Tuplas

{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 ex-criatupla).

> tu = ('Genero', 'especie', 'peso', 'estagio') tu[0] 'Genero' tu[1:3] ('especie', 'peso')

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.

> tu=() tu () tu='casa', -Repare na vírgula ao final! tu ('casa',) tu=1,2,3,4 tu (1, 2, 3, 4) var =w,x,y,z var (w,x,y,z) var = tu w 1 x 2 y 3 z 4

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

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

Conforme exemplificado em ex-criatupla2, 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 ex-criatupla2, apenas mais longo.

Na sétima linha da listagem ex-criatupla2, 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 sequênca 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 ex-formstring.

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 ex-criatupla2, 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.

4.3 Strings

{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.

> st='123 de oliveira4' len(st) 16 min(st) ' ' max(st) 'v' texto = """primeira linha segunda linha terceira linha""" print texto primeira linha segunda linha terceira linha

Conforme ilustrado na listagem ex-string, uma string é uma sequência de quaisquer caracteres alfanuméricos, incluindos 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 caracter. 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}.

> animal='Hamster 1' peso=98 '
>
> 'Hamster 1: 98 gramas'

{strings!formatando}

Na listagem ex-formstring, 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:

> animal='Hamster 1' peso=98 '
>
> 'Hamster 1: 98 gramas' animal+': '+peso+' gramas' Traceback (most recent call last): File "input", line 1, in ? TypeError: cannot concatenate 'str' and 'int' objects

Pelo erro apresentado na listagem ex-concstring, 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}(ss:dict){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.

> Z='C':12, 'O':16, 'N':12, 'Na':40 Z['O'] 16 Z['H']=1 Z 'Na': 40, 'C': 12, 'H': 1, 'O': 16, 'N': 12 Z.keys() ['Na', 'C', 'H', 'O', 'N'] Z.haskey('N') 1

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

Os conjuntos (chave:valor) são chamados de ítens do dicionários. Esta terminologia é importante pois podemos acessar, separadamente, chaves, valores ou ítens 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 ex-criadic, pode-se adicionar novos ítens 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.

{dicionários!métodos}

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

> Z.items() [('Na', 40), ('C', 12), ('H', 1), ('O', 16), ('N', 12)] i=Z.iteritems() i dictionary-iterator object at 0x8985d00 i.next() ('Na', 40) i.next() ('C', 12) e assim por diante... k=Z.iterkeys() k.next() 'Na' k.next() 'C' k.next() 'H' k.next() 'O' k.next() 'N' k.next() Traceback (most recent call last): File "input", line 1, in ? 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`.

4.4 Conjuntos

{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 ex-conjuntos).

> a = set('pirapora') b = set('paranapanema') a letras em a set(['i', 'p', 'r', 'a', 'o']) a - b Letras em a mas não em b set(['i', 'o']) a b letras em a ou b set(['a', 'e', 'i', 'm', 'o', 'n', 'p', 'r']) a b letras em a e b set(['a', 'p', 'r']) a b letras em a ou b mas não em ambos set(['i', 'm', 'e', 'o', 'n'])

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

a - b: $A - B$ [3]

[a | b:] $A \cup B$

[a & b:] $A \cap B$

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

[3] Por convenção representa-se conjuntos por letras maiúsculas.

CAPÍTULO 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 ex-ifelif).

> if a == 1: este bloco é executado se a for 1 pass elif a == 2: este bloco é executado se a for 2 pass
> else: este bloco é executado se se se nenhum dos blocos anteriores tiver sido executado pass

{if}{elif}{else}

No exemplo ex-ifelif, 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 ex-brdict). As condições são as chaves de um dicionário cujos valores são funções. Esta solução não contempla o `else`, porém.

> desfechos = 1:fun1,2:fun2 desfechos[a]

5.1 Iteração

{iteração} 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}:}{while} O laço `while` repete uma tarefa enquanto uma condição for verdadeira (Listagem ex-loops). 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.

> while True: passrepete indefinidamente i=0 while i 10: i +=1 print i saida omitida for i in range(1):
> print i

{O laço *for}:}{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 ex-iterdic nós vimos que dicionários possuem métodos

específicos para iterar sobre seus componentes. Agora suponhamos que desejássemos iterar sobre uma lista e seu índice?

for n,e in enumerate(['a','b','c','d','e']): print "

0: a 1: b 2: c 3: d 4: e

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

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

perguntas = ['nome','cargo','partido'] respostas = ['Lula','Presidente','PT'] for p,r in zip(perguntas,respostas): print "qual o seu

qual o seu nome? Lula qual o seu cargo? Presidente qual o seu partido? PT

{zip}

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

for i in reversed(range(5)): print i 4 3 2 1 0

for i in sorted(set(l)): print i laranja leite manga ovos 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`.{break}

5.2 Lidando com erros: Exceções

{try}{except}{finally}{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 erros [1].

1/0 Traceback (most recent call last): File "stdin", line 1, in ? 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 ex-exception. Caso você queira continuar com a execução do programa apesar do erro, poderíamos solucionar o problema conforme o exposto na listagem ex-try

for i in range(5): ... try: ... q=1./i ... print q ... except ZeroDivisionError: ... print "Divisão por zero!" ... Divisão por zero! 1.0 0.5 0.333333333333 0.25

A construção {try:\ldots 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.

[1] Os erros tratados nesta seção não são erros de sintaxe mas erros que ocorrem durante a execução de programas sintaticamente corretos. Estes erros serão denomidados `exceções`

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 ex-multexc).

try: f = open('arq.txt') s = f.readline() i = int(s.strip()) except IOError, (errno, strerror): print "Erro de I/O (

except ValueError: print "Não foi possível converter o dado em Inteiro." except: print "Erro desconhecido."

A construção {try:\ldots 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 util para tarefas que precisam ser executadas de qualquer forma, como fechar arquivos ou conexões de rede. {Funções}{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 nomes). 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 método [2] chamado {__call__}. Todos os objetos que possuam este método poderão ser chamados [3].

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:

def nome(par1, par2, par3=valordefault, *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 ex-funbas).

def fun(a,b=1): ... print a,b ... fun(2) 2 1 fun(2,3) 2 3 fun(b=5,2) SyntaxError: non-keyword arg after keyword arg

{funçoes!argumentos opcionais} 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 ex-kwargs).

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

{funções!lista de argumentos variável} Funções são chamadas conforme ilustrado na linha 3 da listagem ex-kwargs. 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 ex-funbas).

Muitas vezes é conveniente "desempacotar" os argumentos passados para uma função a partir de uma tupla ou dicionário. {funções!passando argumentos}

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

Argumentos passados dentro de um dicionário podem ser utilizados simultâneamente para argumentos de passagem obrigatória (declarados no cabeçalho da função sem valor "default") e para argumentos opcionais, declarados ou não(Listagem ex-passdic).

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

[3] O 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".

def fun2(a, b=1,**outros): ... print a, b, outros ... dic = 'a':1,'b':2,'c':3,'d':4 fun2(**dic) 1 2 'c': 3, 'd': 4

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

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

def soma(a,b): return a+b print "ignorado!" soma (3,4) 7

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

5.3 Funções lambda

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

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

Observe no exemplo (ex-lamb), 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} 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()`.

def letras(palavra): for i in palavra: yield i for L in letras('gato'): print L g a t o

Como vemos na listagem ex-ger um gerador é uma função sobre a qual podemos iterar. {Decoradores}{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.

def faznada(f): def novaf(*args,**kwargs): print "chamando...",args,kwargs return f(*args,**kwargs) novaf.name = f.name novaf.doc = f.doc novaf.dict.update(f.dict) return novaf

Na listagem ex-dec, 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 ex-decuso, nos mostra como utilizar o decorador.

@faznada def soma(a,b): return a+b

soma(1,2) chamando... (1, 2) Out[5]:3

O decorador da listagem ex-dec, na verdade adiciona uma linha de código à função que decora: {print "chamando...",args,kwargs}.

Repare que o decorador da listagem ex-dec, 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 funçao 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.

from functools import wraps def meuDecorador(f): ... @wraps(f) ... def novaf(*args, **kwds): ... print 'Chamando funcao decorada ' ... return f(*args, **kwds) ... return novaf ... @meuDecorador ... def exemplo(): ... """Docstring""" ... print 'funcao exemplo executada!' ... exemplo() Chamando funcao decorada funcao exemplo executada! exemplo.name 'exemplo' exemplo.doc 'Docstring'

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

> def soma(a,b): “”” Esta funcao soma dois numeros: soma(2,3) 5 “”” return a+b help(soma) Help on function soma in module main:
>
> soma(a, b) Esta funcao soma dois numeros: soma(2,3) 5

No exemplo ex-docst, 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. {Módulos e Pacotes}{módulos} 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”. {import} 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 cap:obj) 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 global [4] 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`.

[float,frame=trBL, caption=Módulo exemplo, label=ex-modfib] {code/fibo.py}

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

> import fibo fibo.fib(50) 1 1 2 3 5 8 13 21 34 fibo.name ‘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 ex-runmod. 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.

> :math:‘$ python fibo.py 60

__main__ [’fibo.py’, ‘60’] 1 1 2 3 5 8 13 21 34 55 end{lstlisting}

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.

[4] Dicionário de nomes de variáveis e funções válidos durante a execução de um script

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}footnote{Além de *__init__.py}, naturalmente.}. Um 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 :ref:‘ex-importing}). begin{lstlisting}[caption=importing a package,label=ex-importing] >>> import pacote >>> dir(pacote) [’modulo1’,’modulo2’] end{lstlisting} .. index:: pacotes;

subsection{Pacotes Úteis para Computação Científica} subsubsection{*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: url{http://www.scipy.org/Numpy_Example_List}

Na listagem :ref:‘ex-det}, 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 algebra linear, são ainda oferecidas como funções através do subpacote linalg (Listagem :ref:‘ex-det}).

begin{lstlisting}[caption=Calculando e mostrando o determinante de uma matriz. ,label=ex-det] >>> from numpy import * >>> a = arange(9) >>> print a [0 1 2 3 4 5 6 7 8] >>> a.shape =(3,3) >>> print a [[0 1 2]

[3 4 5] [6 7 8]]

```
>>> from numpy.linalg import det
>>> det(a)
0.0
>>>
\end{lstlisting}
```

Na primeira linha do exemplo :ref:‘ex-det}, 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 textbf{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$‘:math:‘$3}, basta atribuir o valor *(3,3)} a *shape}. Conforme já vimos, atributos e métodos de objetos são referenciados usando-se esta notação de pontofootnote{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}. subsubsection{*Scipy}} .. index:: scipy .. index:: pair:módulo;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:url{http://www.scipy.org/Documentation}.

section{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.

subsection{Pydoc} .. index:: 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. begin{lstlisting}[caption= ,label=] $‘

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 demontrada 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 (ex-epdh).

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 , 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 fig:epydoc).

Dada toda esta funcionalidade, vale apena conferir o Epydoc [5].

[5] http://epydoc.sourceforge.net

Exercícios

1. Repita a iteração do exemplo ex-enumerate 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 ex-brdict utilizando exceções.
3. Escreva um exemplo de iteração empregando `break`, `continue` e ‘‘else‘‘(ao final).

Programação Orientada a Objetos

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

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 popularidade, 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. {objetos} {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 gramatical 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 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.

7.1 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.

class Objeto: pass

{classe} Na listagem ex:classe1, temos uma definição mínima de uma classe de objetos. Criamos uma classe chamada 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.

class pessoa: idade=20 altura=170 sexo='masculino' peso=70

{classe!atributos} Na listagem ex:classe2, definimos alguns atributos para a classe pessoa. Agora, podemos criar instâncias do objeto pessoa e estas instâncias herdarão estes atributos.

maria = pessoa() maria.peso 70 maria.sexo 'masculino' maria main.pessoa instance at 0x402f196c

Entretanto, os atributos definidos para o objeto pessoa (listagem ex:classe2), 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.

class pessoa: ... def init(self,idade,altura,sexo,peso): ... self.idade=idade ... self.altura=altura ... self.sexo=sexo ... self.peso=70 maria = pessoa() Traceback (most recent call last): File "stdin", line 1, in ? TypeError: init() takes exactly 5 arguments (1 given) maria=pessoa(35,155,'feminino',50) maria.sexo 'feminino'

A função {__init__} que definimos na nova versão da classe pessoa (listagem ex:classe4), 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 ex:classe4, linha 7), recebemos uma mensagem de erro. Na linha 11 da listagem ex:classe4 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. {classe!métodos} 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 ex:classe4). O argumento `self` também deve ser o primeiro argumento a ser declarado na lista de argumentos de um método.

7.2 Herança

{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 ex:her).

class Vertebrado: vertebra = True class Mamifero(Vertebrado): mamas = True class Carnivoro(Mamifero): longoscaninos = True bicho = Carnivoro() dir(bicho) ['doc', 'module', 'longoscaninos', 'mamas', 'vertebra'] issubclass(Carnivoro,Vertebrado) True bicho.class class main.Carnivoro at 0xb7a1d17c isinstance(bicho,Mamifero) True

Na listagem ex:her, vemos um exemplo de criação de um objeto, instância da classe Carnivoro, herdando os atributos dos ancestrais desta. Vemos também que é possivel 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.

7.3 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:

> class Cachorro: pass rex=Cachorro() rex.dono = 'Pedro' rex.raca = 'Pastor' rex.peso=25 rex.dono 'Pedro' laika = Cachorro() laika.dono AttributeError: Cachorro instance has no attribute 'dono'

No exemplo ex:classbag, a classe `Cachorro` é criada vazia, mas ainda assim, atributos podem ser atribuidos a suas instâncias, sem alterar a estrutura da classe. {Exercícios}

1. Utilizando os conceitos de herança e os exemplos de classes apresentados, construa uma classe `Cachorro` que herde atributos das classes `Carnivoro` 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__}.

{Criando Gráficos em Python}(ch:plot) {Introdução à produção de figuras de alta qualidade utilizando o pacote matplotlib. \textbf{Pré-requisitos:} Capítulo \ref{cap:intro}.}

{E} {xiste} 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 ex:fil): simplicidade, elegância, versatilidade, etc. À época, a aplicação destes critérios nos deixou apenas uma opção: o módulo matplotlib [1].

[1] http://matplotlib.sourceforge.net

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 interativa 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 ex:instmat). Basta baixar o pacote **tar.gz** do sítio, descompactar e executar o comando de instalação.

{lstlisting} [caption=Instalando o matplotlib ,label=ex:instmat] :math:‘$ python setup.py install end{lstlisting}

O MPL procura tornar simples tarefas de plotagem, simples e tarefas complexas, possíveis (listagemref{ex:hist}, figura ref{fig:hist}). 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. begin{lstlisting}[caption=Criando um histograma no modo interativo ,label=ex:hist] >>> from pylab import * >>> from numpy.random import * >>> x=normal(0,1,1000) >>> hist(x,30) ... >>> show() end{lstlisting}

begin{figure} centering includegraphics[width=10cm]{hist.png} caption{Histograma simples a partir da listagem ref{ex:hist}} label{fig:hist}

end{figure}

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 instaladosfootnote{Veja no sitio do MPL os pré-requisitos para cada uma das GUIs}.

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).

subsection{Configurando o MPL} O MPL possui valores textit{default} para propriedades genéricas dos gráficos gerados. Estas configurações ficam em um arquivo texto chamado textbf{matplotlibrc}, que deve ser copiado da distribuição do MPL, editado conforme as preferências do usuário e renomeado para texttt{$‘:math:‘$/.matplotlibrc}, ou seja, deve ser colocado como um arquivo oculto no diretório textbf{home} do usuário.

A utilização de configurações padrão a partir do textbf{matplotlibrc} é mais útil na utilização interativa do MPL, pois evita a necessidade de configurar cada figura de acordo com as nossas preferências, a cada vez que usamos o MPLfootnote{Para uma descrição completa das características de gráficos que podem ser configuradas, veja o ememplo de textbf{matplotlibrc} que é fornecido com a distribuição do MPL.}.

subsection{Comandos Básicos} Os comandos relacionados diretamente à geração de gráficos são bastante numerosos(tabela ref{tab:plot}); mas, além destes, existe um outro conjunto ainda maior de comandos, voltados para o ajuste fino de detalhes dos gráficos (ver tabela ref{tab:lineprop}, para uma amostra), tais como tipos de linha, símbolos, cores, etc. begin{table} centering begin{tabular}{lll} hline texttt{bar} & Gráfico de barras \ hline texttt{cohere} & Gráfico da função de coerência \ hline texttt{csd} & Densidade espectral cruzada \ hline texttt{errorbar} & Gráfico com barras de erro \ hline texttt{hist} & Histograma \ hline texttt{imshow} & Plota

imagens \ hline texttt{pcolor} & Gráfico de pseudocores \ hline texttt{plot} & Gráfico de linha \ hline texttt{psd} & Densidade espectral de potência \ hline texttt{scatter} & Diagrama de espalhamento \ hline texttt{specgram} & Espectrograma \ hline texttt{stem} & Pontos com linhas verticais \ hline end{tabular} caption{Principais comandos de plotagem do MPL}label{tab:plot} end{table} Uma explicação mais detalhada dos comandos apresentados na tabela ref{tab:plot}, será dada nas próximas seções no contexto de exemplos.

section{Exemplos Simples} subsection{O comando texttt{plot}} O comando plot é um comando muito versátil, pode receber um número variável de argumentos, com diferentes saídas. begin{lstlisting}[frame=trBL, caption=Gráfico de linha ,label=ex:linha] from pylab import * plot([1,2,3,4]) show() end{lstlisting} begin{figure}

centering includegraphics[width=10cm]{line.png} caption{Reta simples a partir da listagem ref{ex:linha}} label{fig:line}

end{figure} Quando texttt{plot} recebe apenas uma sequência de números (lista, tupla ou array), ele gera um gráfico (listagem ref{ex:linha}) utilizando os valores recebidos como valores de textbf{y} enquanto que os valores de textbf{x} são as posições destes valores na sequência.

Caso duas sequências de valores sejam passadas para texttt{plot} (listagem ref{ex:ponto}), a primeira é atribuida a textbf{x} e a segunda a textbf{y}. Note que, neste exemplo, ilustra-se também a especificação do tipo de saída gráfica como uma sequência de pontos. O parâmtero texttt{'ro'} indica que o símbolo a ser usado é um círculo vermelho. begin{lstlisting}[frame=trBL, caption=Gráfico de pontos com valores de textbf{x} e textbf{y} especificados. ,label=ex:ponto] from pylab import * plot([1,2,3,4], [1,4,9,16], 'ro') axis([0, 6, 0, 20]) savefig('ponto.png') show() end{lstlisting} begin{figure}

centering includegraphics[width=10cm]{ponto.png} caption{Gráfico com símbolos circulares a partir da listagem ref{ex:ponto}} label{fig:ponto}

end{figure} Na linha 3 da listagem ref{ex:ponto} especifica-se também os limites dos eixos como uma lista de quatro elementos: os valores mínimo e máximo dos eixos textbf{x} e textbf{y}, respectivamente. Na linha 4, vemos o comando texttt{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 texttt{plot}. Uma das maneiras mais diretas é através dos argumentos listados na tabela ref{tab:lineprop}. Nos diversos exemplos apresentados neste capítulo, alguns outros métodos serão apresentados e explicadosfootnote{Para maiores detalhes consulte a documentação do MPL (http://matplotlib.sourceforge.net).}.

Vale a pena ressaltar que o comando texttt{plot} aceita, tanto listas, quanto arrays dos módulos texttt{Numpy}, texttt{Numeric} ou texttt{numarray}. Na verdade todas a sequências de números passadas para o comando texttt{plot} são convertidas internamente para texttt{arrays}. begin{table} centering caption{Argumentos que podem ser passados juntamente com a função plot para controlar propriedades de linhas.}label{tab:lineprop} begin{tabular}{|l|l|} Propriedade & Valores \ hline texttt{alpha} & transparência (0-1) \ texttt{antialiased} & true | false \ texttt{color} & Cor: b,g,r,c,m,y,k,w \ texttt{label} & legenda \

texttt{linestyle} & verb|– : -. -| \

texttt{linewidth} & Espessura da linha (pontos) \ texttt{marker} & verb|+ o . s v x > < ^|\ texttt{markeredgewidth} & Espessura da margem do símbolo \ texttt{markeredgecolor} & Cor da margem do símbolo \ texttt{markerfacecolor} & Cor do símbolo \ texttt{markersize} & Tamanho do símbolo (pontos) \ hline end{tabular} end{table} subsection{O Comando texttt{subplot}} O MPL trabalha com o conceito de textit{figura} independente do de textit{eixos}. O comando texttt{gcf()} retorna a figura atual, e o comando texttt{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 seja, o usúário não precisa tomar conhecimento deles. A listagem ref{ex:subplot} apresenta uma figura com dois eixos feita de maneira bastante simples. lstinputlisting[frame=trBL, caption=Figura com dois gráficos utilizando o comando subplot. ,label=ex:subplot]{code/subplot.py} %begin{lstlisting}[float,frame=trBL, caption=Figura com dois gráficos utilizando o comando subplot. ,label=ex:subplot]

%end{lstlisting} begin{figure}

centering includegraphics[width=10cm]{subplot.png} caption{Figura com dois gráficos utilizando o comando subplot, a partir da listagem ref{ex:subplot}} label{fig:subplot}

end{figure}

O comando texttt{figure(1)}, na linha 11 da listagem ref{ex:subplot}, é opcional, mas pode vir a ser importante quando se deseja criar múltiplas figuras, antes de dar o comando texttt{show()}. Note pelo primeiro comando texttt{plot} da listagem ref{ex:subplot}, que o comando texttt{plot} aceita mais de um par textbf{(x,y)}, cada qual com seu tipo de linha especificado independentemente. subsection{Adicionando Texto a Gráficos} O MPL nos oferece quatro comandos para a adição de texto a figuras: texttt{title}, texttt{xlabel}, texttt{ylabel}, e texttt{text}. O três primeiros adicionam título e nomes para os eixos textbf{x} e textbf{y}, respectivamente.

Todos os comandos de inserção de texto aceitam argumentos (tabela ref{tab:texto}) adicionais para formatação do texto. begin{table} caption{Argumentos opcionais dos comandos de inserção de texto.}label{tab:texto} begin{tabular}{lll} textbf{Propriedades} & textbf{Valores} \ hline 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 \ hline end{tabular} end{table} O MPL também nos permite utilizar um subconjunto da linguagem TeX para formatar expressões matemáticas (Listagem ref{ex:mathtext} e figura ref{fig:mathtext}. Para inserir expressões em TeX, é necessário que as strings contendo as expressões matemáticas sejam ‘‘raw strings’‘footnote{exemplo: r’raw string’}, e delimitadas por cifrões($‘). [frame=trBL, caption=Formatando texto e expressões matemáticas ,label=ex:mathtext] {code/mathtext.py}

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.

9.1 Mapas

O matplotlib pode ser extendido para plotar mapas. Para isso precisamos instalar o Basemap toolkit. Se você já instalou o matplotlib, 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 ex:mapa).

[frame=trBL, caption=Plotando o globo terrestre,label=ex:mapa] {code/mapa.py}

Na listagem fig:mapa, 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 $1000km^2$.

{Ferramentas de Desenvolvimento} {Exposição de ferramentas voltadas para o aumento da produtividade em um ambiente de trabalho em computação científica. \textbf{Pré-requisitos:} Capítulos \ref{cap:intro} e \ref{cap:obj}}

{C}omo 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. {Ipython}{Ipython} (sec:ipython) A utilização interativa do Python é de extrema valia. Outros ambientes voltados para computação científica, tais como Matlab, R, Mathematica 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:

{lstlisting} [language=csh, caption= ,label=] :math:‘$ ipython [opções] arquivos end{lstlisting} Muita das opções que controlam o funcionamento do Ipython não são passadas na linha de comando, estão especificadas no arquivo texttt{ipythonrc} dentro do diretório texttt{~/.ipython}.

Quatro opções do Ipython são consideradas especiais e devem aparecer em primeiro lugar, antes de qualquer outra opção: texttt{-gthread, -qthread, -wthread, -pylab}. As três primeiras opções são voltadas para o uso interativo de módulos na construção de GUIs (interfaces gráficas), respectivamente texttt{GTK}, texttt{Qt}, texttt{WxPython}. Estas opções iniciam o Ipython em um ‘‘thread” separado, de forma a permitir o controle interativo de elementos gráficos. A opção texttt{-pylab} permite o uso interativo do pacote matplotlib (Ver capítulo ref{ch:plot}). Esta

opção executará lstinline{from pylab import *} ao iniciar, e permite que gráficos sejam exibidos sem necessidade de invocar o comando texttt{show()}, mas executará scripts que contém texttt{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. subsection{Comandos Mágicos}index{Ipython!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, texttt{%autoindent} liga a indentação automática dentro do Ipython.

Existe uma opção que vem ativada por default no texttt{ipythonrc}, denomidada texttt{automagic}. Com esta função, os comandos mágicos podem ser chamados sem o %, ou seja texttt{autoindent} é entendido como texttt{%autoindent}. Variáveis definidas pelo usuário podem mascarar comandos mágicos. Portanto, se eu definir uma variável lstinline{autoindent = 1}, a palavra texttt{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 texttt{%magic} retorna um explicação dos comandos mágicos existentes.

begin{description} item[texttt{%Exit}] Sai do console Ipython.

item [texttt{%Pprint}] Liga/desliga formatação do texto. item [texttt{%Quit}] Sai do Ipython sem pedir confirmação. item [texttt{%alias}] Define um sinônimo para um comando.

end{description}

Você pode usar texttt{%1} para representar a linha em que o comando texttt{alias foi chamado}, por exemplo: begin{lstlisting}[caption= ,label=] In [2]: alias all echo "Entrada entre parênteses: (%l)" In [3]: all Ola mundo Entrada entre parênteses: (Ola mundo) end{lstlisting}

begin{description} item[texttt{%autocall}] Liga/desliga modo que permite chamar funções sem os parênteses. Por exemplo: texttt{fun 1} vira fun(1).

item [texttt{%autoindent}] Liga/desliga auto-indentação. item [texttt{%automagic}] Liga/desliga auto-mágica. item [texttt{%bg}] Executa um comando em segundo plano, em um thread separado. Por exemplo: texttt{%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 texttt{jobs.results[5]}

end{description}

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

begin{description} item[texttt{%bookmark}]Gerencia o sistema de marcadores do Ipython. Para saber mais sobre marcadores digite texttt{%bookmark?}.

item [texttt{%cd}] Muda de diretório. item [texttt{%colors}]Troca o esquema de cores. item [texttt{%cpaste}]Cola e executa um bloco pré-formatado da área de transferência (clipboard). O bloco tem que ser terminado por uma linha contendo lstinline{–}. item [texttt{%dhist}]Imprime o histórico de diretórios. item [texttt{%ed}]Sinônimo para texttt{%edit} item [texttt{%edit}] Abre um editor e executa o código editado ao sair. Este comando aceita diversas opções, veja a documentação. end{description}

O editor a ser aberto pelo comando texttt{%edit} é o que estiver definido na variável de ambiente texttt{$'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 funcão `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.

In [6]:

IPython will make a temporary file named: /tmp/ipythoneditGuUWr.py done. Executing edited code... Out[6]:"def fun(): print 'fun' funa(): print 'funa"'

In [7]:fun() fun

In [8]:funa() funa

In [9]:

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.

In [1]:

1: ip.magic("

In [2]:

1: ip.magic("

2: ip.magic("

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 MATLAB (veja capítulo ch:plot). O pacote principal do `numpy` também fica exposto no modo `pylab`. Subpacotes do numpy precisam ser importados manualmente. {Editores de Código}{editores} 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

{Editores}

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

Emacs: Editor incrivelmente completo e versátil, funciona como ambiente integrado de desenvolvimento (figura fig:emacs). 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. {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 fig:scite).{Scite}

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

[Jedit:] Incluí o Jedit nesta lista, pois oferece suporte ao desenvolvimento em Jython (ver Seção sec:jython). Afora isso, é um editor bastante poderoso para java e não tão pesado quanto o Eclipse (figura fig:jedit).{Jedit}

[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/

CAPÍTULO 11

Editores Especializados

{IDEs} 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 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 fig:boa).

[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 de versão, geradores de documentação, etc.(Figura fig:eric).

[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 fig:pydev). Em termos de funcionalidade, equipara-se ao Eric e ao Boa-constructor.

11.1 Controle de Versões em Software

{Controle de Versões} 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 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 em Python: `Mercurial` [1]. 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

[1] http://www.selenic.com/mercurial

operações. Além do mais, o uso diário de sistema de controle de versões envolve apenas dois ou três comandos. {Entendendo o Mercurial} {Mercurial} {Controle de Versões!Mercurial} O Mercurial é um sistema de controle de versões descentralizado, ou seja, não há nenhuma noção de um servidor central onde fica 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 fig:mercrep 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 fig:mercrep, um repositório é composto por um Arquivo [2] 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.

Ao final de cada ciclo de trabalho, o pesquisador envia suas modificações para o arquivo numa operação denominada "commit"(figura fig:commit) [3].

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 uma revisão. Estas revisões recebem uma numeração sequencial (figura fig:commit). 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 fig:ana1.

Nosso segundo desenvolvedor, Bruno, acabou de se juntar ao time e clona o repositório Ana [4].

:math:‘$ hg clone ssh://maquinadana/projeto meuprojeto

requesting all changes adding changesets adding manifests adding file changes added 4 changesets with 4 changes to 2 files end{lstlisting} begin{leftbar} textbf{URLs válidas:}\file://\ http://\ https://\ ssh://\ static-http:// end{leftbar}

Após o comando acima, Bruno receberá uma cópia completa do arquivo de Ana, mas seu diretório de trabalho, texttt{meu projeto}, permanecerá independente. Bruno está ansioso para começar a trabalhar e logo faz dois texttt{commits} (figura ref{fig:bruno1}). begin{figure}

centering includegraphics[width=10cm]{bruno1.png} % bruno1.png: 410x60 pixel, 72dpi, 14.46x2.12 cm, bb=0 0 410 60 caption{Modificações de Bruno.}

label{fig:bruno1} end{figure}

Enquanto isso, em paralelo, Ana também faz suas modificações (figura ref{fig:ana2}). begin{figure}

centering includegraphics[width=10cm]{ana2.png} % ana2.png: 340x60 pixel, 72dpi, 11.99x2.12 cm, bb=0 0 340 60 caption{Modificações de Ana.}

label{fig:ana2} end{figure}

[2] Doravante grafado com "A" maiúsculo para diferenciar de arquivos comuns(files).

[3] Vou 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.

[4] Assumimos aqui que a máquina da ana está executando um servidor `ssh`

Bruno então decide ‘‘puxar” o repositório de Ana para sincronizá-lo com o seu. begin{lstlisting}[language=csh, caption= ,label=] $‘

> hg pull pulling from ssh://maquinadaana/projeto searching for changes adding changesets adding manifests adding file changes added 1 changesets with 1 changes to 1 files (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 fig:bruno2. 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.

> :math:‘$ hg update

this update spans a branch affecting the following files: hello.py (resolve)

aborting update spanning branches! (use ‘hg merge’ to merge across branches or ‘hg update -C’ to lose changes) end{lstlisting}

Devido à ramificação no Arquivo de Bruno, o comando texttt{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. begin{lstlisting}[language=csh, caption= ,label=] $‘

> hg merge tip 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 fig:bruno3.

Agora, se Ana puxar de Bruno, receberá todas as moficações de Bruno e seus repositórios estarão plenamente sincronizados, como a figura fig:bruno3. {Criando um Repositório}

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

> 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.

11.1.1 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 descobrí-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.
>
> [`addremove`] 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.

[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.

[update] Atualiza ou funde o diretório de trabalho.

[verify] Verifica a integridade do repositório.

[version] Retorna versão e informação de copyright.

{Interagindo com Outras Linguagens}(ch:capext) {Introdução a vários métodos de integração do Python com outras linguagens. \textbf{Pré-requisitos:} Capítulos \ref{cap:intro} e \ref{cap:obj}.}

CAPÍTULO 12

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

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. Coincidentemente, o primeiro método que vamos explorar para incorporar código C em programas Python, é oferecido como parte do pacote *Scipy*. {Weave}{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 ex:weaveloop).

[frame=trBL, caption=Otimização de loops com o \texttt{weave}, label=ex:weaveloop] {code/weaveloop.py}

No exemplo ex:weaveloop 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.

[frame=trBL, caption=Calculando iterativamente a série de Fibonacci em Python e em \texttt{C}(\texttt{weave.inline}), label=ex:weavefib] {code/weavefib.py}

No exemplo ex:weavefib, o ganho de performance do weave.inline já não é tão acentuado.

:math:‘$ python weavefib.py

Tempo médio no python: 1.69277e-05 segundos Tempo médio no weave: 1.3113e-05 segundos Aceleração média: 1.49554 +- 0.764275 end{lstlisting}

subsection{Ctypes}index{Ctypes} O pacote texttt{ctypes}, parte integrante do Python a partir da versão 2.5, é um módulo que nos permite invocar funções de bibliotecas em texttt{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 usuario tenha um bom conhecimento da linguagem texttt{C}.

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

Dado seu estágio atual de desenvolvimento, o texttt{ctypes} não é a ferramenta mais indicada ao cientista que deseja fazer uso das conveniências do texttt{C} em seus programas Python. Portanto, vamos apenas dar dois exemplos básicos para que o leitor tenha uma ideia de como funciona o texttt{ctypes}. Para maiores informações recomendamos o tutorial do ctypes (url{ http://python.net/crew/theller/ctypes/tutorial.html})

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

begin{lstlisting}[frame=trBL, caption=Carregando bibliotecas em texttt{C}, label=ex:ctypes1] >>> from ctypes import * >>> libc = cdll.LoadLibrary(“libc.so.6”) >>> libc <CDLL ‘libc.so.6’, handle ... at ...> end{lstlisting}

Uma vez importada uma biblioteca (listagem ref{ex:ctypes1}), podemos chamar funções como atributos das mesmas. begin{lstlisting}[frame=trBL, caption=Chamando fuções em bibliotecas importadas com o ctypes, label=ex:ctypes2] >>> libc.printf <_FuncPtr object at 0x...> >>> print libc.time(None) 1150640792 >>> printf = libc.printf >>> printf(“Ola, %sn”, “Mundo!”) Ola, Mundo! end{lstlisting}

subsection{Pyrex}index{Pyrex} O Pyrex é uma linguagem muito similar ao Python feita para gerar módulos em texttt{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: begin{itemize}

item Não é permitido definir funções dentro de outras funções;

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

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

item Definições de classes e funções não podem ocorrer dentro de estruturas de controle (if, elif, etc.);

item Operadores textit{in situ} (+=, *=, etc.) não são suportados pelo Pyrex; item List comprehensions não são suportadas; item Não há suporte a Unicode. end{itemize}

Para exemplificar o uso do Pyrex, vamos implementar uma função geradora de números primos em Pyrex (listagem ref{ex:pyrex}).

lstinputlisting[frame=trBL, caption=Calculando números primos em Pyrex, label=ex:pyrex]{code/primes.pyx}

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 texttt{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 texttt{C}, mais eficiente será o código texttt{C} gerado pelo Pyrex. A variável texttt{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 texttt{while} ao invés de um laço do tipo texttt{for i in range(x)}. Este ultimo seria mais lento devido a incluir a função texttt{range} do Python.

O próximo passo é gerar a versão em texttt{C} da listagem ref{ex:pyrex}, compilar e linká-lo, transformando-o em um módulo Python. begin{lstlisting}[language=csh,frame=trBL, caption=Gerando Compilando e linkando, label=ex:pyrexc] $‘

pyrexc primes.pyx :math:‘$ gcc -c -fPIC -I/usr/include/python2.4/ primes.c

$‘ gcc -shared primes.o -o primes.so

Agora vamos comparar a performance da nossa função com uma função em Python razoávelmente bem implementada (Listagem ex:pyrex2). Afinal temos que dar uma chance ao Python, não?

[frame=trBL, caption=Calculando números primos em Python, label=ex:pyrex2] {code/primes2.py}

Comparemos agora a performance das duas funções para encontrar todos os números primos menores que 100000. Para esta comparação utilizaremos o ipython que nos facilita esta tarefa através da função mágica {\%timeit}.

> In [1]:from primes import primes In [2]:from primes2 import primes as primesp In [3]:%timeit primes(100000) 10 loops, best of 3: 19.6 ms per loop In [4]:%timeit primesp(100000) 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 distribuido 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 ex:setupix. Uma vez criado o script (batizado, por exemplo, de `setupyx.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 incluida na distribuição do programa, o setuptools passa então para a etapa de compilação e linkagem do código `C`.

[frame=trBL, caption=Automatizando a compilação de extensões em \texttt{Pyrex} por meio do setuptools, label=ex:setupix] {code/setupyx.py}

> import setuptools from setuptools.extension import Extension

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}{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 fig:trapezio.

Matemáticamente, podemos expressar a regra trapezoidal da seguinte fórmula.

:math:‘$int_a^b f(x);dx approx frac{h}{2}(f(a)+f(b))+hsum_{i=1}^{n-1}f(a+ih),;;;h=frac{b-a}{n}

$‘

(eq:trapezio)

Na listagem ex:trapintloop podemos ver uma implementação simples da regra trapezoidal.

[frame=trBL, caption=implementação da regra trapezoidal(utilizando laço for) conforme especificada na equação \ref{eq:trapezio}, label=ex:trapintloop] {code/trapintloop.py}

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

:math:‘$ python trapintloop.py

resultado: 16 Tempo: 11.68 seg resultado: 49029 Tempo: 26.96 seg end{lstlisting}

Para converter o script ref{ex:trapintloop} em texttt{C++} tudo o que precisamos fazer é: begin{lstlisting}[language=csh,frame=trBL, caption=Verificando o tempo de execução em texttt{C++}] $‘

ss trapintloop.py *** SHED SKIN Python-to-C++ Compiler 0.0.15 *** Copyright 2005, 2006 Mark Dufour; License GNU GPL version 2 or later (See LICENSE) (If your program does not compile, please mail me at mark.dufour@gmail.com!!)

WARNING trapintloop:13: ‘xrange’, ‘enumerate’ and ‘reversed’ return lists for now [iterative type analysis..] ** iterations: 2 templates: 55 [generating c++ code..] :math:‘$ make run

g++ -O3 -I ... ./trapintloop resultado: 16 Tempo: 0.06 seg resultado: 49029 Tempo: 1.57 seg end{lstlisting}

Com estes dois comandos, geramos, compilamos e executamos uma versão texttt{C++} do programa listado em ref{ex:trapintloop}. O código texttt{C++} gerado pelo Shed-Skin pode ser conferido na listagem ref{ex:trapintloop_C}

Como pudemos verificar, o ganho em performance é considerável. Lamentávelmente, 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.

lstinputlisting[language=c++,frame=trBL, caption=Código texttt{C++} gerado pelo Shed-skin a partir do script trapintloopy.py,label=ex:trapintloop_C]{code/trapintloop.cpp} section{Integração com a Linguagem texttt{Fortran}}index{FORTRAN} A linguagem texttt{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 texttt{Fortran} deriva de ‘‘IBM mathematical FORmula TRANslation system’‘. Dada a sua longa existência, existe uma grande quantidade de código científico escrito em texttt{Fortran} disponível para uso. Felizmente, a integração do texttt{Fortran} com o Python pode ser feita de forma extremamente simples, através da ferramenta texttt{f2py}, que demonstraremos a seguir. subsection{texttt{f2py}} Esta ferramenta está disponível como parte do Pacote numpy (url{www.scipy.org}). Para ilustrar o uso do texttt{f2py}, vamos voltar ao problema da integração pela regra trapezoidal (equação ref{eq:trapezio}). Como vimos, a implementação deste algoritmo em Python, utilizando um laço texttt{for}, é ineficiente. Para linguagens compiladas como texttt{C++} ou texttt{Fortran}, laços são executados com grande eficiência. Vejamos a performance de uma implementação em texttt{Fortran} da regra trapezoidal (listagem ref{ex:trapintf}).

lstinputlisting[language=fortran,frame=trBL, caption=implementação em texttt{Fortran} da regra trapezoidal. label=ex:trapintf]{code/trapint.f}

A listagem ref{ex:compfor} nos mostra como compilar e executar o código da listagem ref{ex:trapintf}. Este comando de compilação pressupõe que você possua o texttt{GCC} (Gnu Compiler Collection) versão 4.0 ou superior. No caso de versões mais antigas deve-se substituir texttt{gfortran} por texttt{g77} ou texttt{f77}. begin{lstlisting}[language=csh,frame=trBL, caption= Compilando e executando o programa da listagem ref{ex:trapintf},label=ex:compfor] $‘

gfortran -o trapint trapint.f :math:‘$ time ./trapint

Resultado: 16.01428 Resultado: 48941.40

real 0m2.028s user 0m1.712s sys 0m0.013s end{lstlisting}

Como em texttt{Fortran} não temos a conveniência de uma função para ‘‘cronometrar” nossa função, utilizamos o comando texttt{time} do Unix. Podemos constatar que o tempo de execução é similar ao obtido com a versão em texttt{C++} (listagem ref{ex:trapintloop_C}).

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

Antes que possamos ‘‘importar” nossas funções em texttt{Fortran} para uso em um programa em Python, precisamos compilar nossos fontes em texttt{Fortran} com o texttt{f2py}. A listagem ref{ex:compf2py} nos mostra como fazer isso. begin{lstlisting}[language=csh,frame=trBL, caption= Compilando com texttt{f2py},label=ex:compf2py] $‘

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 ex:trapintloop) para verificar os ganhos de performance. A listagem

ex:trapintloopcomp contém nosso script de teste. [frame=trBL, caption=Script para comparação entre Python e \texttt{Fortran} via \texttt{f2py},label=ex:trapintloopcomp] {code/trapintloopcomp.py}

A listagem ex:trapintloopcomp 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 integrar 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 ex:comp). 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.

:math:‘$ python trapintloopcomp.py

resultado: 16 Tempo: 13.29 seg resultado: 49029 Tempo: 29.14 seg tempo do Fortran com funcoes em Python resultado: 16 Tempo: 7.31 seg resultado: 48941 Tempo: 24.95 seg tempo do Fortran com funcoes em Fortran resultado: 16 Tempo: 4.85 seg resultado: 48941 Tempo: 6.42 seg end{lstlisting}

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

lstinputlisting[frame=trBL, caption=Implementação vetorizada da regra trapezoidal,label=ex:trapintvect]{code/trapintvect.py}

Executando a listagem ref{ex:trapintvect}, vemos que a implementação vetorizada em Python ganha (0.28 e 2.57 segundos)de nossas soluções utilizando texttt{f2py}.

Da mesma forma que com o texttt{Pyrex}, podemos distribuir programas escritos em Python com extensões escritas em texttt{Fortran}, com a ajuda do pacote setuptools. Na listagem ref{ex:setupf2py} vemos o exemplo de como escrever um setup.py para este fim. Neste exemplo, temos um texttt{setup.py} extrememente limitado, contendo apenas os parâmetros necessarios para a compilação de uma extensão denominada texttt{flib}, a partir de uma programa em texttt{Fortran}, localizado no arquivo texttt{flib.f}, dentro do pacote ‘‘texttt{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. begin{lstlisting}[frame=trBL, caption=texttt{setup.py} para distribuir programas com extensões em texttt{Fortran} ,label=ex:setupf2py] import ez_setup ez_setup.use_setuptools() import setuptools from numpy.distutils.core import setup, Extension flib = Extension(name=’meupacote.flib’,

libraries=[], library_dirs=[], f2py_options=[], sources=[’meupacote/flib.f’])

setup(name = ‘mypackage’,

version = ‘0.3.5’, packages = [’meupacote’], ext_modules = [flib]

)

end{lstlisting}

section{A Píton que sabia Javanês — Integração com Java}index{Java} Peço 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 texttt{Java}.

A linguagem texttt{Java} tem, todavia, contra si uma série de fatores: A complexidade de sua sintaxe rivaliza com a do texttt{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 texttt{Java} tem de bom, sem levar como ‘‘brinde” seus aspectos negativos? É aqui que entra o texttt{Jython}.

O texttt{Jython} é uma implementação completa do Python 2.2footnote{O desenvolvimento do texttt{Jython continua, mas não se sabe ainda quando alcançará o CPython (implementação em texttt{C} do Python).}} em texttt{Java}. Com o texttt{Jython} programadores texttt{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 texttt{Java}. Além destas vantagens, O texttt{Jython} também é uma linguagem Open Source, ou seja de código aberto.

subsection{O interpretador Jython}label{sec:jython} index{Jython}

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

Vamos tentar usá-lo como usaríamos o interpretador Python e ver se notamos alguma diferença. begin{lstlisting}[frame=trBL, caption=Usando o interpretador texttt{Jython} ,label=lst:int-jython] $‘

> jython Jython 2.1 on java1.4.2-01 (JIT: null) Type “copyright”, “credits” or “license” for more information. print ‘hello world’ hello world import math() dir(math) [’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’] math.pi 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`.

> import javax.swing.JOptionPane; class testDialog public static void main (String[] args) javax.swing.JOptionPane.showMessageDialog (null, “Isto e um teste.”);

A versão apresentada na listagem lst:Swingjava está escrita em `Java`. Vamos ver como ficaria a versão em `Jython`.

> import javax.swing.JOptionPane javax.swing.JOptionPane.showMessageDialog(None,”Isto e um teste.”)

Podemos observar, na listagem lst:Swingjython, 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, utilizado-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 lst:Swingjython, seu resultado encontra-se na figura fig:Swing-jython.

14.1 Criando “Applets” em Jython

Para os conhecedores 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 lst:applet-jython).

> import java.applet.Applet; class appletp (java.applet.Applet): def paint (self, g): g.drawString (“Eu sou um Applet 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 lst:applet-jython para que a máquina virtual `Java` possa executá-lo. Para isso, salvamos o código e utilizamos o compilador do `Jython`, o `jythonc`.

> jythonc -deep -core -j appletp.jar appletp.py processing appletp
>
> Required packages: java.applet
>
> Creating adapters:
>
> Creating .java files: appletp module appletp extends java.applet.Applet
>
> Compiling .java to .class... Compiling with args: [’/opt/blackdown-jdk-1.4.2.01/bin/javac’, ‘-classpath’, ‘/usr/share/jython/lib/jython-2.1.jar:/usr/share/libreadline-java/lib/libreadline-java.jar:.:./jpywork::/usr/share/jython/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 lst:htmlapp. Então, basta apontar nosso navegador para este documento e veremos o applet ser executado.

html head meta content="text/html; charset=ISO-8859-1" http-equiv="content-type" titlejython applet/title /head body Este eacute; o seu applet em Jython:br br br center applet code="appletp" archive="appletp.jar" name="Applet em Jython" alt="This browser doesn't support JDK 1.1 applets." align="bottom" height="50" width="160" PARAM NAME="codebase" VALUE="." h3Algo saiu errado ao carregar este applet./h3 /applet /center br br /body /html

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

Exercícios

1. Compile a listagem ex:weaveloop 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 ex:pyrex), 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 `append` do laço `while`. Compare a performance desta nova versão com a da versão original.

{Jython: A python que sabia Javanês} {P} {eço} 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 contra si uma série de fatores: A complexidade de sua sintaxe rivaliza com a do C++, é uma linguagem proprietária, e não é eficiente como esperaríamos que uma linguagem compilada, com tipagem estática fosse. Mas todos estes aspectos negativos não são suficientes para anular as vantagens da sua grande biblioteca 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 2.1 [1] em Java. Com o Jython programadores Java podem embutir o python em seus aplicativos e applets e nós, programadores Python podemos utilizar misturar 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.

[1] Ao final de 2005 está prometida compatibilidade com o Python 2.3

O interpretador Jython

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

Vamos tentar usá-lo como usaríamos o interpretador python e ver se notamos alguma diferença.

:math:‘$ jython

Jython 2.1 on java1.4.2-01 (JIT: null) Type “copyright”, “credits” or “license” for more information. >>> print ‘hello world’ hello world >>> import math() >>> dir(math) [’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’] >>> math.pi 3.141592653589793 end{lstlisting}

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. begin{lstlisting}[language=Java,frame=trBL, caption=Um programa simples em Java usando a classe Swing. ,label=lst:Swing-java] import javax.swing.JOptionPane; class testDialog {

public static void main (String[] args) { javax.swing.JOptionPane.showMessageDialog (null, “Isto e um teste.”);

}

} end{lstlisting}

A versão apresentada na listagem ref{lst:Swing-java} está escrita em Java. Vamo ver como ficaria a versão em Jython. begin{lstlisting}[frame=trBL, caption=O mesmo programa da listagem ref{lst:Swing-java} em Jython.,label=lst:Swing-jython] >>> import javax.swing.JOptionPane >>> javax.swing.JOptionPane.showMessageDialog(None,”Isto e um teste.”) end{lstlisting}

Podemos observar na listagem ref{lst:Swing-jython} que eliminamos a verborragia característica do Java, e o programa em Jython ficou bem mais “pitônico”. Outro detalhe que vale a pena comentar, é que não precisamos compilar o código em Jython, com seria necessário com o Java. Só isto já é uma grande vantagem do Jython. Em suma, utilizado-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 ref{lst:Swing-jython}, Seu resultado encontra-se na figura ref{fig:Swing-jython}.

begin{figure} centering includegraphics[bb=0 0 269 127]{jyswing.eps}

% jyswing.jpg: 72dpi, width=9.49cm, height=4.48cm, bb=0 0 269 127 caption{Saída da listagem ref{lst:Swing-jython}.} label{fig:Swing-jython}

end{figure}

section{Criando ‘‘Applets” em Jython} Para os conhecedores de Java, o Jython pode ser utilizado para criar ‘‘servlets”, ‘‘beans” e ‘‘applets” com a mesma facilidade com que criamos um aplicativo em Jython. Vamos

ver um exemplo de ‘‘applet”(listagem ref{lst:applet-jython}). begin{lstlisting}[frame=trBL, caption=Criando um applet em Jython ,label=lst:applet-jython] import java.applet.Applet; class appletp (java.applet.Applet):

def paint (self, g): g.drawString (“Eu sou um Applet Jython!”, 5, 5)

end{lstlisting}

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 ref{lst:applet-jython} para que a máquina virtual Java possa executá-lo. Para isso, salvamos o código utilizaremos o compilador do Jython, texttt{jythonc}. begin{lstlisting}[language= Ksh,frame=trBL, caption= Compilando appletp.py ,label=lst:jythonc] $‘

jythonc -deep -core -j appletp.jar appletp.py processing appletp

Required packages: java.applet

Creating adapters:

Creating .java files: appletp module appletp extends java.applet.Applet

Compiling .java to .class... Compiling with args: [’/opt/blackdown-jdk-1.4.2.01/bin/javac’, ‘-classpath’, ‘/usr/share/jython/lib/jython-2.1.jar:/usr/share/libreadline-java/lib/libreadline-java.jar:.:./jpywork::/usr/share/jython/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 precisamos embuti-lo” em um documento HTML (listagem lst:htmlapp. Então basta apontar nosso navegador para este documento e veremos o applet ser executado.

html head meta content=”text/html; charset=ISO-8859-1” http-equiv=”content-type” titlejython applet/title /head body Este eacute; o seu applet em Jython:br br br center applet code=”appletp” archive=”appletp.jar” name=”Applet em Jython” alt=”This browser doesn’t support JDK 1.1 applets.” align=”bottom” height=”50” width=”160” PARAM NAME=”codebase” VALUE=”.” h3Algo saiu errado ao carregar este applet./h3 /applet /center br br /body /html

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

{Teoria de Grafos} {Grafos} {Breve introdução a teoria de grafos e sua representação computacional. Introdução ao Pacote \texttt{NetworkX}, voltado para a manipulação de grafos. \textbf{Pré-requisitos:} Programação orientada a objetos.}

CAPÍTULO 17

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 ex:grafdict mostra um dicionário representando o grafo da figura fig:g1. 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 ex:vlist).

g = ‘a’:[’c’,’d’,’e’],’b’:[’d’,’e’],’c’:[’a’,’d’],’d’:[’b’,’c’,’a’],’e’:[’a’,’b’]

Podemos utilizar o método `keys` para obter uma lista dos vértices de nosso grafo.

g.keys() [’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 possivel. Porém, para dar mais alcance aos nossos exemplos sobre teoria de grafos, vamos nos utilizar do pacote `NetworkX` [1] que já implementa uma representação bastante completa de grafos e redes em Python.

[1] https://networkx.lanl.gov/

NetworkX

{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 $V = \{W, X, Y, Z\}$:

$$‘G : A = (W, Z), (Z, Y), (Y, X), (X, Z) : math : ‘$$

[frame=trBL,caption=Definindo um grafo através de seus vértices,label=ex:graph1] {code/graph1.py} Executando o código acima, obtemos:

$$‘[’Y’,’ X’,’ Z’,’ W’][(’Y’,’ X’), (’Y’,’ Z’), (’X’,’ Z’), (’Z’,’ W’)] : math : ‘$$

Ao lidar com grafos, é conveniente representá-los graficamente. Vejamos como obter o diagrama do grafo da listagem ex:graph1: [frame=trBL,caption=Diagrama de um grafo,label=ex:graph2] {code/graph2.py}

A funcionalidade do pacote `NetworkX` é bastante ampla. A seguir exploraremos um pouco desta funcionalidade.

18.1 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.

18.2 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 $u \longrightarrow v$. 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. [1]

[G.subgraph(listadevertices)]Retorna subgrafo correspondente à lista de vértices.

18.3 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.

[cartesian_product(G1,G2)]Produto cartesiano de dois grafos (Figura fig:gpcc).

[compose(G1,G2)]Combina grafos, identificando vértices com mesmo nome.

[complement(G)]Retorna o complemento do grafo(Figura fig:gpcc).

[create_empty_copy(G)]Cópia vazia de G.

[convert_to_undirected(G)]Retorna uma cópia não direcionada de G.

[convert_to_directed(G)]Retorna uma cópia não direcionada de G.

[convert_node_labels_to_integers(G)]Retorna uma cópia com os vértices renomeados como números inteiros.

18.4 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. [frame=trBL,caption=Construindo um grafo dinamicamente,label=ex:cont] {code/grafodin.py}

{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.

[1] Uma 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.

{Módulos!threading}

O objeto grafo do `NetworkX` aceita qualquer objeto como um vértice. Na listagem ex:cont, nos valemos deste fato para colocar instâncias da classe `Contagio` como vértices do grafo G. O grafo G é contruído 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 infeccioso, 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 fig:cont)

18.5 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. [frame=trBL,caption=Construindo uma rede social a partir de e-mails,label=ex:mnet] {code/mnet.py}

Na Listagem ex:mnet, usamos dois módulos interessantes da biblioteca padrão do Python: O módulo `email` e o módulo mailbox. `mailbox`.

{Módulo email:} Módulo para decodificar, manusear, e compor emails.

{Módulos!email}

{Módulo mailbox:} Conjuto de classes para lidar com caixas de correio no formato Unix, MMDF e MH.

{Módulos!mailbox}

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 documentaçao 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 fig:mnet).

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.
4. Insira um print no laço for do exemplo ex:mnet para ver o formato de saída do iterador mbox.
5. Modifique o programa ex:mnet para associar apenas mensagens que contêm uma palavra em comum.

{Interação com Bancos de Dados}{bancos de dados} {Apresentação dos módulos de armazenamento de dados Pickle e Sqlite3 que fazem parte da distribuição padrão do Python. Apresentação do pacote SQLObject para comunicação com os principais sistemas de bancos de dados existentes. \textbf{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, visualizacão 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.

CAPÍTULO 20

O Módulo Pickle

{pickle} O módulo `pickle` e seu primo mais veloz `cPickle`, implementam algoritmos que permitem armazenar, em um arquivo, objetos implementados em Python.

> In [1]:import pickle In [2]:class oi: .2.: def digaoi(self): .2.: print "oi" In [3]:a= oi() In [4]:f = open('picteste','w') In [5]:pickle.dump(a,f) In [6]:f.close() In [7]:f = open('picteste','r') In [8]:b=pickle.load(f) In [9]:b.digaoi() oi

Como vemos na listagem ex:pickle, 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 ex:pickle. Quando um objeto é armazenado por meio do módulo `pickle`, nem o código da classe, nem seus dados, são incluidos, apenas os dados da instância.

> In [10]:class oi: .10.: def digaoi(self,nome='flavio'): .10.: print 'oi

> In [11]:f = open('picteste','r') In [12]:b=pickle.load(f) In [13]:b.digaoi() 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. {O módulo Sqlite3}{sqlite} 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 `SQL` [1].

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`.

> In [1]:import sqlite3

O próximo passo é a criação de um objeto "conexão", através do qual podemos executar comandos SQL.

> 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] SQL 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

In [3]:c.execute(‘‘‘create table especimes(nome text, altura real, peso real)’‘’) 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 necessáriamente 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.

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:

In [5]:t = (‘tom’,) In [6]:c.execute(‘select * from especimes where nome=?’,t) In [7]:c.fetchall() [(‘tom’, 12.5, 2.2999999999999998)]

No exemplo acima utilizamos o método `fetchall` para recuperar o resultado da operação. Caso desejássemos obter um único registro, usaríamos `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.

In [8]:t = ((‘jerry’,5.1,0.2),(‘butch’,42.4,10.3)) In [9]:for i in t: .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.

In [10]:c.execute(‘select * from especimes by peso’) In [11]: for reg in c: print reg (‘jerry’,5.1,0.2) (‘tom’, 12.5, 2.2999999999999998) (‘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”.

CAPÍTULO 21

O Pacote SQLObject

{SQLObject} O pacote SQLObject [1] 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.

21.1 Construindo um aranha digital

{aranha}{web-spider} {Módulos!BeautifulSoup} Neste exemplo, teremos a oportunidade de construir uma aranha digital que recolherá informações da web (Wikipedia [2]) 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. [linerange={1-6},frame=trBL, caption=Módulos necessários, label=ex:arimp] {code/aranha.py}

O pacote `BeautifulSoup` [3] 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`. [linerange={8-11},frame=trBL, caption=Especificando o banco de dados., label=ex:arbdset] {code/aranha.py}

Na listagem ex:arbdset, 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, utilizamos `os.path.expanduser` para substituir o {\~} por `/home/usuario` como aconteceria no console unix normalmente.

[1] http://www.sqlobject.org/
[2] http://pt.wikipedia.org
[3] http://www.crummy.com/software/BeautifulSoup/

Na linha 11 da listagem ex:arbdset, 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. [linerange={16-20},frame=trBL, caption=Especificando a tabela \texttt{ideia} do banco de dados., label=ex:arsql] {code/aranha.py}

A classe que representa nossa tabela é herdeira da classe `SQLObject`. Nesta classe, a cada atributo (coluna da tabela) deve ser atribuido 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 armazernar 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 rastejador 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.{urllib2}{urllib}

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 ex:arresto que fazemos toda esta filtragem sofisticada em duas linhas de código(55 e 56), graças ao `BeautifulSoup`. [linerange={15-107},frame=trBL, caption=Restante do código da aranha., label=ex:arresto] {code/aranha.py}

A listagem ex:arresto 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.

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 23

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", distribuido 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.

23.1 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 fig:shell, vemos um exemplo de uma sessão do bash rodando em uma janela.

23.1.1 Alguns Comando Úteis

ls Lista arquivos.

- cp

Copia arquivos.

- mv

Renomeia ou move arquivos.

- rm

Apaga arquivos (de verdade!).

- ln

Cria links para arquivos.

- pwd

Imprime o nome do diretório corrente (caminho completo).

- mkdir

Cria um diretório.

- rmdir

Remove um diretório. Para remover recursivamente toda uma árvore de diretórios use rm -rf(cuidado!).

- cat

Joga o arquivo inteiro na tela.

- less

Visualiza o arquivo com possibilidade de movimentação e busca dentro do mesmo.

- head

Visualiza o início do arquivo.

- tail

Visualiza o final do arquivo.

- nl

Visualiza com numeração das linhas.

- od

Visualiza arquivo binário em base octal.

- xxd

Visualiza arquivo binário em base hexadecimal.

- gv

Visualiza arquivos Postscript/PDF.

- xdvi

Visualiza arquivos DVI gerados pelo .

- stat

Mostra atributos dos arquivos.

- wc

Conta bytes/palavras/linhas.

- du

Uso de espaço em disco.

- file

Identifica tipo do arquivo.

- touch

Atualiza registro de última atualização do arquivo. Caso o arquivo não exista, é criado.

- chown

Altera o dono do arquivo.

- chgrp

Altera o grupo do arquivo.

- chmod

Altera as permissões de um arquivo.

- chattr

Altera atributos avançados de um arquivo.

- lsattr

Lista atributos avançados do arquivo.

- find

Localiza arquivos.

- locate

Localiza arquivo por meio de índice criado com updatedb.

- which

Localiza comandos.

- whereis

Localiza o binário (executável), fontes, e página man de um comando.

- grep

Busca em texto retornando linhas.

- cut

Extrai colunas de um arquivo.

- paste

Anexa colunas de um arquivo texto.

- sort

Ordena linhas.

- uniq

Localiza linhas idênticas.

- gzip

Compacta arquivos no formato GNU Zip.

- compress

Compacta arquivos.

- bzip2

Compacta arquivos(maior compactação do que o gzip, porém mais lento.

- zip

Compacta arquivos no formato zip(Windows).

- diff

Compara arquivos linha a linha.

- 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.

23.2 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 comunicaçao 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 padrao 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 àgua 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”.

23.2.1 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 saida de um programa para um arquivo, por exemplo.

:math:‘$ sort < nomes > nomes_ordenados

end{lstlisting}

O comando do exemplo ref{ex:redir}, cria um novo arquivo com o conteúdo do arquivo texttt{nomes}, ordenado.

subsection{‘‘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 texttt{$‘:math:‘$} conhecido como ‘‘pipe’‘. Qualquer linha de comando conectando dois ou mais comandos através de ‘‘pipes” é denominada de ‘‘pipeline’‘. begin{lstlisting}* language=csh, caption=Lista ordenada dos usuários do sistema. ,label=ex:pipe

$‘ cut -d: -f1 /etc/passwd sort ajaxterm avahi avahi-autoipd backup beagleindex bin boinc ...

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 desenvolvidos * 1

_ para serem

utilizados diretamente do console. Nesta seção, vamos ver alguns exemplos, uma vez que seria impossível explorar todos eles, neste simples apêndice.

23.2.2 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;

- **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 bi-dimensionais 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:}

{EDO}

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

{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.

:math:'$ graph -T png < arquivo_de_dados_ascii > plot.png

end{lstlisting}

Se o texttt{arquivo_de_dados_ascii} contiver duas colunas de números, o programa as atribuirá a texttt{x} e texttt{y}, respectivamente. Os pares ordenados que darão origem aos pontos do gráfico não precisam estar em linhas diferentes. por exemplo: begin{lstlisting}* language=csh, caption=Desenhando um quadrado. ,label=ex:graphq

$' echo .1 .1 .1 .9 .9 .9 .9 .1 .1 .1 graph -T X -C -m 1 -q 0.3

A listagem ex:graphq 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".

:math:'$ echo 0 0 1 1 2 0 | spline | graph -T X

end{lstlisting} begin{figure}

centering includegraphics* width=10cm

{spline.png}

% spline.png: 578x594 pixel, 72dpi, 20.39x20.95 cm, bb=0 0 578 594 caption{Usando texttt{spline}.} label{fig:spline}

end{figure}

Spline não serve apenas para interpolar funções, também pode ser usado para interpolar curvas em um espaço d-dimensional utilizando-se a opçao texttt{-d}. begin{lstlisting}* language=csh, caption=Interpolando uma curva em um plano. ,label=ex:splinec

echo 0 0 1 0 1 1 0 1 | spline -d 2 -a -s | graph -T X end{lstlisting}

O comando da listagem ref{ex:splinec} traçará uma curva passando pelos pontos texttt{(0,0)}, texttt{(1,0)}, texttt{(1,1)} e texttt{(0,1)}. A opção texttt{-d 2} indica que a variável dependente é bi-dimensional. A opção texttt{-a} indica que a variável independente deve ser gerada automáticamente e depois removida da saída (opção texttt{-s}). begin{figure}

centering includegraphics* width=10cm

{splinec.png}

% splinec.png: 578x594 pixel, 72dpi, 20.39x20.95 cm, bb=0 0 578 594 caption{Interpolando uma curva em um plano.} label{fig:splinec}

end{figure}

subsubsection{texttt{ode}}

O utilitário texttt{ode} é capaz de produzir uma solução numérica de sistemas de equações diferenciais ordinárias. A saída de texttt{ode} pode ser redirecionada para o utilitário texttt{graph}, que já discutimos anteriormente, de forma que as soluções sejam plotadas diretamente, à medida em que são calculadas.

Vejamos um exemplo simples: begin{equation}

dfrac{dy}{dt}=y(t)

end{equation}

A solução desta equação é:

begin{equation} y(t)=e^t

end{equation}

Se nós resolvermos esta equação numericamente, a partir do valor inicial $‘y(0)=1:math:‘$, até $‘t=1:math:‘$ esperaríamos obter o valor de $‘e:math:‘$ como último valor da nossa curva ($‘e1=2.718282:math:‘$, com 7 algarismos significativos). Para isso digitamos no console: begin{lstlisting}* language=csh, caption= Resolvendo uma equação diferencial simples no console do Linux. ,label=ex:ode1

$‘ ode y’=y y=1 print t,y step 0,1

Após digitar a ultima linha do exemplo ex:ode1, 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 ex:ode1 em um arquivo texto. Abra o seu editor favorito, e digite o seguinte modelo: * language=csh,frame=trBL, caption=Sistema de três equações diferenciais acopladas ,label=ex:lorenz

{code/lorenz}

Salve o arquivo com o nome `lorenz`. Agora digite no console a seguinte linha de comandos:

ode lorenz graph -T X -C -x -10 10 -y -20 20

E eis que surgirá a bela curva da figura fig:lorenz.

Indices and tables

- *genindex*
- *modindex*
- *search*

Índice