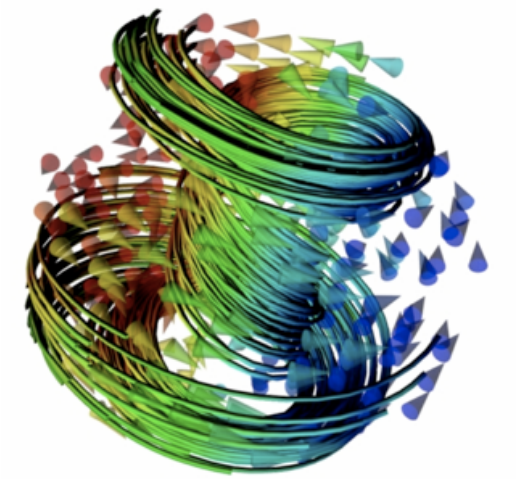
Introdução à Linguagem

Python para Ciências Computacionais e Engenharia

(Um guia para o iniciante em Python 3)



Hans Fangohr
Faculdade de Engenharia e Meio Ambiente
Universidade de Southampton

Traduzido por Gustavo Charles P. de Oliveira
Departamento de Computação Científica
Universidade Federal da Paraíba

1 de Junho de 2018

Conteúdo

1 Introdução

Este texto resume algumas ideias centrais relevantes para a Engenharia Computacional e Computação Científica usando Python. A ênfase está em apresentar alguns conceitos de programação em Python que são importantes para algoritmos numéricos. Os capítulos posteriores versam sobre bibliotecas numéricas, tais como `NumPy` e `SciPy`, as quais, individualmente, merecem muito mais espaço para serem abordadas do que o fornecido aqui. Temos o objetivo de capacitar o leitor para aprender de maneira independente a como usar as funcionalidades dessas bibliotecas através da documentação disponível tanto *online* quanto nos próprios pacotes.

1.1 Modelagem Computacional

1.1.1 Introdução

Cada vez mais, processos e sistemas são pesquisados ou desenvolvidos através de simulações computacionais: novos protótipos de aeronaves, como o recente A380, são primeiramente projetados e testados virtualmente através de simulações computacionais. Com o crescente poder computacional disponível através de supercomputadores, clusters de computadores e até mesmo de máquinas *desktop* e *laptops*, essa tendência provavelmente permanecerá.

Simulações computacionais são rotineiramente utilizadas na pesquisa básica para ajudar a entender medições experimentais e para substituir - por exemplo - o crescimento e a fabricação de amostras/experiências custosas, sempre que possível. Em um contexto industrial, o projeto de produtos e dispositivos geralmente pode ser feito muito mais eficazmente de modo virtual através de simulações do que através da construção e teste de protótipos. Isto ocorre, em particular, em áreas onde as amostras são custosas, como é o caso da nanociência (onde é custoso criar coisas pequenas) e da indústria aeroespacial (onde é custoso construir coisas grandes). Existem também situações em que certas experiências só podem ser realizadas virtualmente (variando da astrofísica ao estudo de efeitos de acidentes nucleares ou químicos de grande escala). A Modelagem Computacional, incluindo o uso de ferramentas computacionais para pós-processamento, análise e visualização de dados, tem sido utilizada na engenharia, física e química por muitas décadas, mas está se tornando mais importante devido à disponibilidade barata de recursos computacionais. A Modelagem Computacional também está começando a desempenhar um papel mais importante nos estudos de sistemas biológicos, economia, arqueologia, medicina, saúde e muitos outros domínios.

```
In [4]: oct(234)
```

```
Out[4]: '0o352'
```

1.1.2 Modelagem Computacional

Para estudar um processo com simulação computacional, distinguimos duas etapas: a primeira é desenvolver um *modelo* do sistema real; a segunda, resolver as equações envolvidas numericamente. Ao estudar o movimento de um pequeno objeto, como uma moeda, digamos, sob a influência da gravidade, podemos ignorar seu atrito com o ar: nosso modelo - que só pode considerar a força gravitacional e a inércia da moeda, ou seja, $a(t) = F/m = -9.81\,m/s^2$ - é uma aproximação do sistema real. O modelo normalmente permitir-nos-á expressar o comportamento do sistema (em alguma forma aproximada) através de equações matemáticas, que geralmente envolvem equações diferenciais ordinárias (EDOs) ou equações diferenciais parciais (EDPs).

Nas ciências naturais, como física, química e engenharia, muitas vezes não é tão difícil encontrar um modelo adequado, embora as equações resultantes tendam a ser muito difíceis de resolver e, na maioria dos casos, impossíveis de serem resolvidas analiticamente.

Por outro lado, em assuntos que não são tão bem descritos através de uma estrutura matemática e dependem do comportamento de objetos cujas ações são impossíveis de serem previstas de forma determinística (como os seres humanos), é muito mais difícil encontrar um modelo adequado para descrever a realidade. Como uma regra geral, nestas disciplinas, as equações resultantes são mais fáceis de serem resolvidas, mas difíceis de serem encontradas e a validade de um modelo precisa ser questionada muito mais. Exemplos típicos são as tentativas de simular a economia, o uso de recursos globais, o comportamento de uma multidão em pânico, etc.

Até agora, apenas discutimos o desenvolvimento de *modelos* para descrever a realidade. O uso desses modelos não envolve necessariamente computadores ou algum trabalho numérico. Na verdade, se a equação de um modelo puder ser resolvida de forma analítica, então deve-se fazer isso e registrar a solução para a equação.

Na prática, dificilmente uma equação-modelo de sistemas de interesse pode ser resolvida analiticamente, e é aí que o computador entra: usando métodos numéricos, podemos, pelo menos, estudar o modelo *para um determinado conjunto de condições de contorno*. Para o exemplo considerado acima, talvez não possamos ver facilmente a partir de uma solução numérica que a velocidade da moeda sob a influência da gravidade mudará linearmente com o tempo (o que podemos constatar facilmente a partir da solução analítica disponível para este sistema simples: $v(t) = 9.81t + v_0\, m/s$).

A solução numérica que pode ser calculada usando um computador consistiria de dados que mostram como a velocidade muda ao longo do tempo para uma velocidade inicial particular v_0 (v_0 é uma condição de contorno aqui). O programa computacional reportaria uma longa lista de dois números mantendo o valor do tempo t_i para o qual um determinado valor da velocidade v_i foi calculado. Ao plotarmos v_i *versus* t_i, ou ajustarmos uma curva a partir dos dados, poderemos entender a tendência do comportamento da moeda (que pode ser vista a partir da solução analítica, é claro).

Claramente, é desejável encontrar soluções analíticas sempre que possível, mas o número de problemas onde isso é possível é pequeno. Geralmente, a obtenção do resultado numérico de uma simulação computacional é bastante útil (apesar das deficiências dos resultados numéricos em comparação com uma expressão analítica), porque é a única maneira possível de estudar o sistema.

Em suma, o nome *modelagem computacional* deriva das duas etapas: (i) *modelagem*, em que se busca um modelo que descreva o sistema real, e (ii) a resolução das equações do modelo usando *métodos computacionais*.

1.1.3 Programação como suporte à modelagem computacional

Existe uma grande quantidade de pacotes que fornecem recursos para modelagem computacional. Se estes satisfizerem as necessidades de pesquisa ou de projeto, e o processamento e a visualização de dados sejam adequadamente suportados através de ferramentas existentes, pode-se realizar estudos de modelagem computacional sem nenhum conhecimento de programação mais aprofundado.

Em um ambiente de pesquisa - tanto na academia quanto na pesquisa por novos produtos e idéias na indústria -, muitas vezes se atinge um ponto em que os pacotes existentes não são capazes de realizar uma determinada tarefa de simulação que surge, ou onde se faz necessário o aprendizado de novas maneiras de analisar os dados existentes.

Nesse ponto, habilidades de programação tornam-se um requisito. Em geral, também é útil ter um entendimento amplo dos blocos de construção de software e idéias básicas de engenharia de software à medida que usamos mais e mais dispositivos que são controlados por software.

Esquecemos, com frequencia, que não há nada que o computador faça que nós, como seres humanos, não possamos fazer. O computador pode fazê-lo muito mais rápido, mas o fará com muitos erros. Portanto, não há mágica nos cálculos que um computador executa: os cálculos poderiam ter sido feitos por seres humanos - na verdade o foram durante muitos anos. Veja, por exemplo, o tópico *Computador Humano* da Wikipedia.

Compreender como construir uma simulação computacional reduz-se, grosso modo, a: (i) encontrar o modelo (muitas vezes, isto significa encontrar as equações corretas), (ii) saber como resolver estas equações numericamente, (iii) implementar os métodos para calcular essas soluções (este é o ponto na programação).

1.2 Por que Python para Computação Científica?

O foco do projeto da linguagem Python está na produtividade e legibilidade de código. Por exemplo, através de:

- console Python interativo;
- sintaxe muito clara e legível através de indentação;
- fortes capacidades de introspecção;
- plena modularidade e pacotes hierárquicos de suporte;
- manipulação de erros com base em exceções;
- tipos de dados dinâmicos e gerenciamento de memória automático.

Como Python é uma linguagem interpretada e, muitas vezes, executada mais lentamente do que código compilado, pode-se perguntar por que alguém deveria considerar essa linguagem "lenta"para simulações computacionais?

Há duas respostas para esta crítica:

1. *Tempo de implementação versus tempo de execução*: não é apenas o tempo de execução que contribui para o custo de um projeto computacional: também é necessário considerar o custo do trabalho de desenvolvimento e manutenção.

Nos primórdios da computação científica (digamos, nas decádas de 1960/70/80), o tempo de computação era tão proibitivo que fazia sentido investir muitos meses do tempo de um programador para melhorar o desempenho de um cálculo em qualquer ínfima porcentagem.

Hoje em dia, no entanto, os ciclos de uma unidade de processamento (CPU) tornaram-se muito mais baratos do que o tempo do programador. Para os códigos de pesquisa, os quais são executados, via de regra, apenas um pequeno número de vezes (antes que os pesquisadores passem para o próximo problema), pode ser econômico aceitar que o código seja executado apenas com 25% da velocidade esperada possível se isto economiza, digamos, um mês do tempo de um pesquisador (ou programador). Por exemplo: se o tempo de execução de parte do código for de 10 horas, com uma previsão de que ele seja executado cerca de 100 vezes, então o tempo total de execução é de aproximadamente 1000 horas. Seria ótimo se isto pudesse ser reduzido para 25%. Seria uma economia de 750 horas de processamento. Por outro lado, uma espera extra (cerca de um mês) e o custo de 750 horas de CPU valeria o investimento de um mês do tempo de uma pessoa.

Legibilidade e manutenção do código - código curto, menos erros: uma questão relacionada é que um código de pesquisa não é usado apenas para um projeto, mas continua sendo usado repetidas vezes, evolui, cresce, bifurca-se etc. Neste caso, o investimento em mais tempo para tornar o código mais rápido é uma justificativa frequente. Ao mesmo tempo, uma quantidade significativa de tempo do programador concentrar-se-á em (i) introduzir as mudanças necessárias, (ii) testá-las mesmo antes de o trabalho de otimização da velocidade de execução da versão alterada começar. Para manter, estender e modificar um código de modos ainda imprevistos, o uso de uma linguagem que seja fácil de ler e que possua grande poder expressivo é recomendado.

2. *Código Python bem escrito pode ser muito rápido* se partes críticas e demoradas do código forem executadas através de linguagens compiladas: normalmente, menos de 5% do código de um projeto de simulação precisam de mais de 95% do tempo de execução. Desde que esses cálculos sejam feitos de forma muito eficiente, não é necessário se preocupar com todas as outras partes do código, já que o tempo total de sua execução é insignificante.

A parte do programa onde há intensa computação deve ser ajustada para alcançar o melhor desempenho. Várias opções são oferecidas em Python:

- Por exemplo, a extensão NumPy fornece uma interface Python para as eficientes bibliotecas compiladas LAPACK, que são quase o padrão em álgebra linear numérica. Se os problemas em estudo puderem ser formulados de forma que, eventualmente, grandes sistemas de equações algébricas tenham que ser resolvidos, ou autovalores computados, etc., o código compilado na biblioteca LAPACK pode ser usado (através do pacote NumPy). Nesta etapa, os cálculos são realizados com o mesmo desempenho do Fortran/C, haja vista que o código usado é essencialmente escrito em Fortran/C. O Matlab, a propósito, explora exatamente isto: a linguagem de *script* Matlab é muito lenta (cerca de 10 vezes mais lenta do que Python), mas o Matlab ganha seu poder ao delegar operações matriciais às bibliotecas compiladas LAPACK.

- As bibliotecas numéricas C/Fortran existentes podem ser "interfaceadas"(para serem chamadas a partir do programa Python) através de, por exemplo, Swig, Boost.Python e Cython.

- O Python pode ser estendido através de linguagens compiladas se a parte do problema com grande demanda computacional não for algoritmicamente padronizada e não houver biblioteca existente que possa ser usada. As linguagens comumente utilizadas para implementar extensões rápidas são C, C++ e Fortran.

- Listamos algumas ferramentas que são usadas para usar código compilado a partir do Python:

- A extensão `scipy.weave` é útil se apenas uma expressão curta precisar ser expressa em C.

- A interface Cython está crescendo em popularidade para declarar semi-automaticamente os tipos de variáveis no código Python, para traduzir esse código para C (automaticamente) e depois usar o código C compilado a partir do Python. O Cython também é usado para encapsular rapidamente uma biblioteca C existente com uma interface para que a biblioteca C possa ser usada a partir do Python.

- Boost.Python é especializado para encapsular código C++ no Python.

A conclusão é que *Python é "rápida o suficiente" para a maioria das tarefas computacionais e sua linguagem de alto nível amigável ao usuário frequentemente é compensada em velocidade reduzida quando comparada a linguagens compiladas de baixo nível. A combinação de Python com códigos compilados escritos sob medida para desempenho de partes críticas do código resulta em velocidades virtualmente ótimas na maioria dos casos.*

1.2.1 Estratégias de otimização

Em geral, entendemos a redução do tempo de execução quando discutimos "otimização de código"no contexto da modelagem computacional e, essencialmente, realizamos os cálculos necessários o mais rápido possível. (Às vezes, precisamos reduzir a quantidade de memória RAM, bem como a quantidade de entrada e saída de dados para o disco ou para a rede). Ao mesmo tempo, precisamos ter certeza de que não investimos quantidades inapropriadas de tempo de programação para acelerar o código. Como sempre, deve haver um equilíbrio entre o tempo dos programadores e a melhoria que podemos obter com isso.

1.2.2 Primeiro faça certo, depois otimize

Para escrever códigos rápidos de forma eficaz, observamos que a ordem correta é: (i) escrever um programa que realize o cálculo correto. Para isso, escolha uma linguagem/abordagem que lhe permita *escrever o código rapidamente e fazê-lo funcionar rapidamente* - independentemente da velocidade

de execução. Então (ii) altere o programa ou reescreva-o do zero na mesma linguagem para tornar a execução mais rápida. Durante o processo, continue comparando os resultados com a versão lenta primeiramente escrita para garantir que a otimização não introduza erros. Uma vez que estivermos familiarizados com o conceito de testes de regressão, eles devem ser usados para comparar o código novo (e esperançosamente mais rápido) com o original.

Um padrão comum em Python é começar a escrever o código Python puro e, em seguida, começar a usar bibliotecas Python que usam código compilado internamente (como os *arrays* rápidos fornecidas pelo NumPy e as rotinas do SciPy, que voltam-se para códigos numéricos já estabelecidos, tais como ODEPACK, LAPACK e outros). Se necessário, pode-se - depois de uma análise de desempenho (*profiling*) cuidadosa - começar a substituir partes do código Python por uma linguagem compilada, como C ou Fortran, a fim de melhorar a velocidade de execução ainda mais (conforme discutido acima).

1.2.3 Prototipagem em Python

Verifica-se que - mesmo se um código específico tiver que ser escrito em, digamos, C++ - é (frequentemente) mais eficiente prototipar o código em Python, e uma vez que um projeto apropriado (estrutura de classes) for encontrado, transferir o código para C++.

1.3 Literatura

Enquanto este texto inicia-se com uma introdução de alguns aspectos básicos da linguagem de programação Python, você pode achar necessário - dependenddo de sua experiência prévia - buscar fontes secundárias para entender algumas ideias completamante. Nesse caso, os seguintes documentos são indicados:

- Allen Downey, *Pense em Python*. Disponível online em html e pdf em https://penseallen.github.io/PensePython2e/.
- A documentação oficial Python: `http://www.python.org/doc/` e
- O tutorial Python (`http://docs.python.org/tutorial/`)

Os seguintes links também podem ser úteis para você:

- Homepage do `numpy` (`http://numpy.scipy.org/`)
- Homepage do `scipy` (`http://scipy.org/`)
- Homepage do `matplotlib` (`http://matplotlib.sourceforge.net/`).
- Guia de estilos Python (`http://www.python.org/dev/peps/pep-0008/`

1.3.1 Vídeo-aulas sobre Python para iniciantes

Você gosta de ouvir/seguir vídeo-aulas? Há uma série de 24 vídeo-aulas intituladas *Introduction to Computer Science and Programming* dadas por Eric Grimsom e John Guttag do MIT disponíveis em `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/`. Estas vídeo-aulas são destinadas a estudantes com pouca ou nenhuma experiência em programação. Elas proporcionam aos alunos uma compreensão do papel que a computação pode desempenhar na resolução de problemas. Elas também têm o objetivo de ajudar os alunos, independentemente de sua formação, a se sentirem confiantes em sua capacidade de escrever pequenos programas que lhes permitam atingir objetivos úteis.

1.3.2 Listas de discussão sobre Python

Existe também uma lista de discussão (http://mail.python.org/mailman/listinfo/tutor) onde iniciantes são bem-vindos para fazer perguntas sobre Python. Tanto usar os arquivos quanto postar suas próprias questões (o que, de fato, ajuda outros também) podem ajudar você a entender a linguagem. Use a etiqueta padrão da lista de discussão (por exemplo, seja educado, conciso, etc.). Você pode ter interesse em ler http://www.catb.org/esr/faqs/smart-questions.html para obter algumas orientações sobre como fazer perguntas em listas de discussão.

1.4 Versões do Python

Existem duas versões de Python por aí: Python 2.x and Python 3.x. Elas são levemente diferentes — as mudanças na Python 3.x foram introduzidas para corrigir algumas deficiências no projeto da linguagem identificados desde o seu início. Uma decisão que foi tomada assumiu que alguma incompatibilidade deveria ser aceita para se atingir o objetivo maior de uma linguagem melhor para o futuro.

Para computação científica, é crucial fazer uso de bibliotecas numéricas tais como NumPy, SciPy e o pacote de plotagem Matplotlib.

Todos eles estão disponíveis para Python 3, versão que usaremos aqui. No entanto, há muito código ainda em uso que foi escrito para Python 2. Para tanto, é útil saber das diferenças. O exemplo mais proeminente é que em Python 2.x, o comando `print` é especial, ao passo que em Python 3 é uma função ordinária. Por exemplo, em Python 2.7, podemos escrever:

```
print "Hello World"
```

ao passo que, em Python 3, isto causaria um erro de sintaxe. A maneira correta de usar `print` em Python 3 seria como uma função, i.e.

```
In [1]: print("Hello World")
```

```
Hello World
```

Felizmente, a notação de função (i.e. com os parênteses) também é permitida em Python 2.7. Assim, nossos exemplos devem funcionar tanto em Python 2.x, quanto em Python 3.x. (Existem outras diferenças não mencionadas aqui.)

1.5 Estes documentos

Este material foi convertido do `LaTex` para um conjunto de `Jupyter Notebooks`, tornando os exemplos nele contidos interativos. Você pode executar qualquer bloco de código iniciado por `In [ ]:` clicando na respectiva célula e pressionando `shift-enter`, ou clicando no botão da barra de ferramentas.

1.6 Seu *feedback*

Seu *feedback* é desejado. Caso você encontre algum erro neste texto, ou tiver sugestões sobre como alterá-lo ou estendê-lo, fique à vontade para entrar em contato com o Hans pelo e-mail `fangohr@soton.ac.uk`.

Se você encontrar uma URL que não está funcionando (ou apontando para o material errado), informe ao Hans também. Como o conteúdo da internet muda rapidamente, é difícil acompanhar essas mudanças sem um *feedback*.

2 Uma calculadora poderosa

==========

2.1 O prompt do Python e o Ciclo Ler-Avaliar-Imprimir (LAI)

Python é uma linguagem *interpretada*. Podemos coletar sequencias de comandos em texto e salvá-los em um arquivo como um *programa Python*. A convenção é que estes arquivos tenham a extensão `.py`, como, por exemplo, `hello.py`.

Podemos também entrar com comandos individuais no `prompt` Python que são imediatamente avaliados e executados pelo interpretador Python. Isto é muito útil para o programador/aprendiz entender como usar certos comandos e depois estendê-los para obter um programa maior. O papel da linguagem Python pode ser descrito como segue: *Ler* o comando, *Avaliar* este comando, *Imprimir* o valor avaliado e repetir o ciclo (*loop*) - isto dá origem à abreviatura LAI. O *prompt* Python é um terminal básico onde você introduz comandos após o marcador `>>>`, como no exemplo a seguir:

```
>>> 2 + 2
4
```

A interface LAI que estamos usando é um *notebook Jupyter*. Blocos de código aparecem com um `In` à esquerda deles.

```
In [2]: 4 + 5

Out[2]: 9
```

Para editar o código, clique na área de código (célula). Uma borda verde indica que a célula está selecionada (consulte o menu de ajuda, `Help`, do Jupyter Notebook para saber mais sobre os modos de *comando* e *edição*). Em seguida, pressione `shift-enter`.

2.2 Calculadora

Operações básicas tais como adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`) e exponenciação (`**`) funcionam (na maioria das vezes) como esperado:

```
In [2]: 10 + 10000

Out[2]: 10010

In [3]: 42 - 1.5

Out[3]: 40.5

In [3]: 47 * 11

Out[3]: 517

In [5]: 10 / 0.5

Out[5]: 20.0

In [6]: 2**2   # O operador de exponenciação ('à potência de') é **, e NÃO ^
```

```
Out[6]: 4

In [7]: 2**3

Out[7]: 8

In [8]: 2**4

Out[8]: 16

In [9]: 2 + 2

Out[9]: 4

In [10]: # Linhas começando com a tralha (#) indicam um comentário
         2 + 2

Out[10]: 4

In [11]: 2 + 2  # é um comentário na mesma linha de código

Out[11]: 4
```

e, usando o fato que $\sqrt[n]{x} = x^{1/n}$, podemos computar $\sqrt{3} = 1.732050\ldots$ usando **:

```
In [12]: 3**0.5

Out[12]: 1.7320508075688772
```

Parentêses podem ser usados para agrupamento:

```
In [7]: 2 * 10 + 5

Out[7]: 25

In [14]: 2 * (10 + 5)

Out[14]: 30
```

2.3 Divisão inteira

Em Python 3, a divisão funciona como você esperaria:

```
In [15]: 15/6

Out[15]: 2.5
```

Em Python 2, no entanto, `15/6` retornará `2`.

Este fenômeno é conhecido (em muitas linguages de programação, incluindo C) como *divisão inteira*: pelo fato de termos fornecido dois números inteiros (`15` e `6`) para o operador de divisão, a hipótese é que o valor de retorno seja também do tipo inteiro. A resposta matematicamente correta é um número em ponto flutuante.

A convenção para divisão inteira é truncar os dígitos fracionários e retornar a parte inteira apenas (i.e., `2` neste exemplo). Ela também pode ser chamada de "divisão por baixo"(*floor division*).

2.3.1 Como evitar a divisão inteira

Há duas maneiras de evitar o problema da divisão inteira:

1. Use o estilo de divisão do Python 3: isto está disponível mesmo no Python 2 com uma declaração especial de importação:

```
python    >>> from __future__ import division    >>> 15/6    2.5
```

Caso você queira usar `from __future__ import division` em um programa Python, a declaração seria incluída normalmenteno no início do arquivo.

2. Alternativamente, se você assegurar que, pelo menos um número (numerador ou denominador) seja do tipo `float` (ou `complex`), o operador de divisão retornará um número em ponto flutuante. Isto pode ser feito escrevendo `15.` em vez de `15`, ou forçando a conversão do número para `float`, i.e. usando `float(15)` em vez de, simplesmente, `15`:

```
python    >>> 15./6    2.5    >>> float(15)/6    2.5    >>> 15/6.    2.5    >>>
15/float(6)    2.5    >>> 15./6.    2.5
```

Se você realmente quiser a divisão inteira, poderá usar `//`. Por exemplo, `1//2` retornará `0` tanto em Python 2 quanto em Python 3.

2.3.2 Por que devo me importar com este problema de divisão?

A divisão inteira pode resultar em problemas surpreendentes: suponha que você esteja escrevendo código para calcular a média $m = (x + y)/2$ de dois números x e y. A primeira tentativa de escrever isto pode ser dada como:

```
m = (x + y) / 2
```

Suponha que isto seja testado com $x = 0.5$,$y = 0.5$. Então, a linha acima computaria a resposta correta $m = 0.5$ (porque `0.5 + 0.5 = 1.0`, i.e. 1.0 é um número de ponto flutuante, e assim `1.0/2` seria avaliado como `0.5`). Além disso, poderíamos usar $x = 10$,$y = 30$, e porque `10 + 30 = 40` e `40/2` seria `20`, obtemos a resposta correta $m = 20$. Entretanto, se tentássemos com os inteiros $x = 0$ e $y = 1$, então o código retornaria $m = 0$ (porque `0 + 1 = 1` e `1/2` seria avaliado como `0`), quando, na verdade, $m = 0.5$ seria a resposta correta.

Temos muitas possibilidades para fazer a linha de código acima funcionar seguramente, incluindo estas três formas:

```
m = (x + y) / 2.0
```

```
m = float(x + y) / 2
```

```
m = (x + y) * 0.5
```

Este comportamento de divisão inteira é comum entre a maioria das linguagens de programação (incluindo as importantes C, C++ e Fortran). Portanto, devemos estar cientes deste fato.

2.4 Funções matemáticas

Pelo fato de Python ser uma linguagem de programação com propósitos gerais, funções matemáticas comumente usadas, tais como seno, cosseno, exponencial, logaritmo e muitas outras, estão localizadas no módulo de matemática chamado `math`. Podemos fazer uso delas assim que importarmos este módulo. Por exemplo:

```
In [17]: import math
         math.exp(1.0)

Out[17]: 2.718281828459045
```

Usando a função `dir`, podemos listar o diretório de objetos disponíveis no módulo `math`:

```
In [17]: dir(math)

Out[17]: ['__doc__',
          '__file__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          'acos',
          'acosh',
          'asin',
          'asinh',
          'atan',
          'atan2',
          'atanh',
          'ceil',
          'copysign',
          'cos',
          'cosh',
          'degrees',
          'e',
          'erf',
          'erfc',
          'exp',
          'expm1',
          'fabs',
          'factorial',
          'floor',
          'fmod',
          'frexp',
          'fsum',
          'gamma',
          'gcd',
          'hypot',
          'inf',
          'isclose',
          'isfinite',
          'isinf',
          'isnan',
          'ldexp',
          'lgamma',
          'log',
          'log10',
          'log1p',
```

```
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

Como de costume, a função `help` pode fornecer mais informação acerca do módulo (use `help(math)`) ou de objetos individuais:

```
In [20]: help(math.exp)

Help on built-in function exp in module math:

exp(...)
    exp(x)

    Return e raised to the power of x.
```

O módulo de matemática define as constantes π and e:

```
In [19]: math.pi

Out[19]: 3.141592653589793

In [20]: math.e

Out[20]: 2.718281828459045

In [21]: math.cos(math.pi)

Out[21]: -1.0

In [22]: math.log(math.e)

Out[22]: 1.0
```

2.5 Variáveis

Uma *variável* pode ser usada para armazenar um certo valor ou objeto. Em Python, todos os números (e todas as outras coisas mais, incluindo funções, módulos e arquivos) são objetos. Uma variável é criada por atribuição:

```
In [22]: x = 0.5
```

Uma vez que a variável x tiver sido criada através da atribuição de 0.5 neste exemplo, podemos fazer uso dela:

```
In [24]: x*3

Out[24]: 1.5

In [25]: x**2

Out[25]: 0.25

In [26]: y = 111
         y + 222

Out[26]: 333
```

Uma variável é substituída se um novo valor for atribuído a ela:

```
In [23]: y = 0.7
         math.sin(y) ** 2 + math.cos(y) ** 2

Out[23]: 1.0000000000000002
```

O sinal de igual (=) é usado para atribuir um valor à uma variável.

```
In [28]: largura = 20
         altura = 5 * 9
         largura * altura

Out[28]: 900
```

Um valor pode ser atribuído a várias variáveis simultaneamente:

```
In [31]: x = y = z = 0  # inicializa x, y e z com 0
         x

In [30]: y

Out[30]: 0

In [31]: z

Out[31]: 0
```

Variáveis devem ser criadas com atribuição de valor antes de serem usadas, senão um erro ocorrerá:

```
In [32]: # tenta acessar uma variável indefinada
         n
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-32-a18bccbd09ad> in <module>()
      1 # tenta acessar uma variável indefinada
----> 2 n


NameError: name 'n' is not defined
```

Em modo interativo, a última expressão impressa é atribuída à variável `_`. Isto significa que quando você está usando Python como uma calculadora de mesa, é um tanto fácil continuar os cálculos. Por exemplo:

```
In [33]: taxa = 12.5 / 100
         preco = 100.50
         preco * taxa

Out[33]: 12.5625

In [34]: preco + _

Out[34]: 113.0625
```

Esta variável deve ser tratada como "somente leitura"pelo usuário. Não atribua, explicitamente, um valor a ela - você criaria uma variável local independente com o mesmo nome assim mascarando o comportamento "mágico"da variável pré-construída.

2.5.1 Terminologia

Estritamente falando, o seguinte acontece quando escrevemos algo como:

```
In [35]: x = 0.5
```

Primeiro, o Python cria o objeto `0.5`. Tudo em Python é um objeto, e assim o é o número em ponto flutuante 0.5. Este objeto é armazenado em algum lugar na memória. Em seguida, o Python *vincula um nome ao objeto*. O nome é `x`, e nos referimos a `x` casual e frequentemente como uma variável, um objeto, ou mesmo o valor 0.5. Entretanto, tecnicamente, `x` é um nome que é limitado ao objeto `0.5`. Outro modo de dizer isto é que `x` é uma *referência* para o objeto.

Enquanto é frequentemente suficiente pensar em atribuir 0.5 à uma variável `x`, existem situações nas quais precisamos lembrar o que realmente ocorre. Em particular, quando passamos referências de objetos para funções, precisamos ter em mente que a função pode operar sobre o objeto (em vez de uma cópia do objeto).

2.6 Equações impossíveis

Em programas computacionais, frequentemente encontramos declarações como

```
In [36]: x = x + 1
```

Se lêssemos esta equação do modo como estamos acostumados da matemática, poderíamos subtrair x de ambos os lados de $x = x + 1$ para descobrir que $0 = 1$. Todavia, sabemos que isto não é verdadeiro. Então, algo está errado aqui...

A resposta é que "equações" em códigos computacionais não são realmente equações, mas *atribuições*. Elas tem de ser lidas em dois passos:

1. Avaliando o valor no membro direito do sinal de igual;
2. Atribuindo este valor à variável cujo nome é mostrado no membro esquerdo. (Em Python: vincula-se o nome à esquerda ao objeto mostrado à direita).

Na literatura de ciência da computação, a notação seguinte é usada para expressar atribuições a fim de se evitar confusão com equações matemáticas:

$$x \leftarrow x + 1$$

Vamos aplicar a nossa regra de dois passos à atribuição `x = x + 1` dada acima:

1. Avalie o valor no membro direito do sinal de igual: para isto, precisamos saber o valor atual de `x`. Assumamos que o valor atual de `x` é `4`. Neste caso, o membro direito, `x+1` é avaliado para `5`.
2. Atribua este valor (i.e. `5`) à variável cujo nome é mostrado no membro esquerdo (`x`).

Confirmemos com o *prompt* do Python que esta é a interpretação correta:

```
In [37]: x = 4
         x = x + 1
         x

Out[37]: 5
```

2.6.1 A notação +=

Por ser uma operação bastante comum aumentar uma variável `x` por alguma quantidade fixa `c`, podemos escrevê-la como:

```
x += c
```

em vez de

```
x = x + c
```

Nosso exemplo inicial acima poderia, assim, ter sido escrito como:

```
In [38]: x = 4
         x += 1
         x

Out[38]: 5
```

Os mesmos operadores são definidos para multiplicação por uma constante (`*=`), subtração de uma constante (`-=`) e divisão por uma constante (`/=`).

Note que a ordem de `+` e `=` importa:

```
In [36]: x = 1
         x += 4
         x

Out[36]: 5
```

aumentará a variável x de um, ao passo que

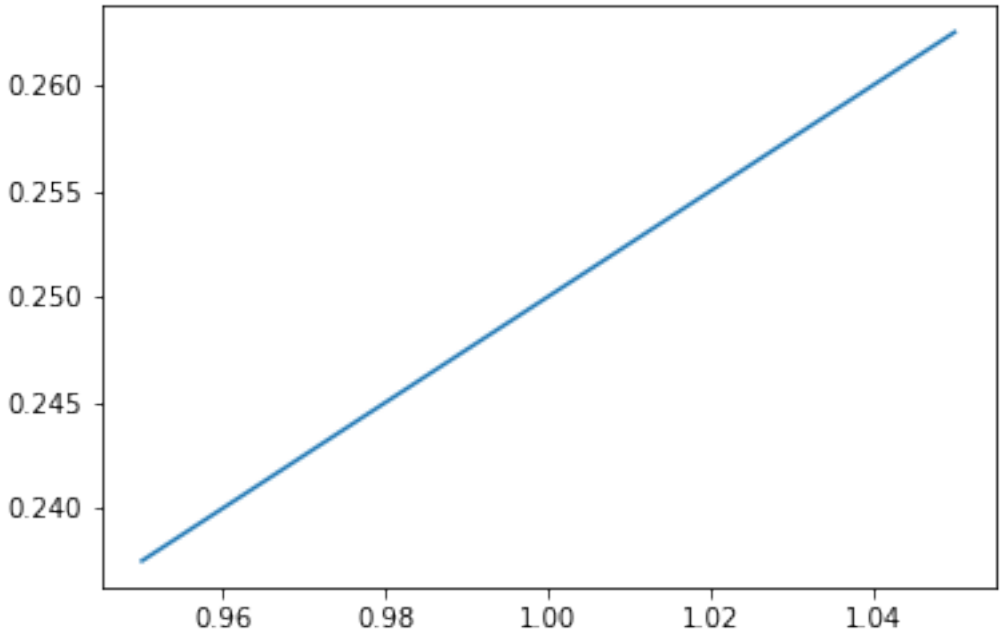
```
In [40]: x =+ 1
```

atribuirá o valor +1 à variável x.
Meu modelo de compressão de rocha

$$\mathbf{T} = \begin{bmatrix} T_{xx} & T_{xy} \\ T_{yx} & T_{yy} \end{bmatrix}$$

$K = c_p \, tr(\mathbf{T}) = c_p \, (T_{xx} + T_{yy})$

```
In [35]: %matplotlib inline

         import math as mt
         import numpy as np
         import matplotlib.pyplot as plt

         T = np.array([[0.1,0.0],[0.0,0.15]])
         print(T)

         CP = np.linspace(0.95,1.05)

         rock = {'cp':CP,'txx':0.1,'txy':0.0,'tyx':0.0,'tyy':0.15}

         # fator
         K = rock.get('cp')*(rock.get('txx') + rock.get('tyy')) + 0.22*(rock.get('txx')**2 -

         plt.plot(CP,K)

[[ 0.1   0.  ]
 [ 0.    0.15]]


Out[35]: [<matplotlib.lines.Line2D at 0x106e76c50>]
```




```
In [32]: CP = np.linspace(0.95,1.05)
         print(CP)
```

```
[ 0.95        0.95204082  0.95408163  0.95612245  0.95816327  0.96020408
  0.9622449   0.96428571  0.96632653  0.96836735  0.97040816  0.97244898
  0.9744898   0.97653061  0.97857143  0.98061224  0.98265306  0.98469388
  0.98673469  0.98877551  0.99081633  0.99285714  0.99489796  0.99693878
  0.99897959  1.00102041  1.00306122  1.00510204  1.00714286  1.00918367
  1.01122449  1.01326531  1.01530612  1.01734694  1.01938776  1.02142857
  1.02346939  1.0255102   1.02755102  1.02959184  1.03163265  1.03367347
  1.03571429  1.0377551   1.03979592  1.04183673  1.04387755  1.04591837
  1.04795918  1.05      ]
```

3 Tipos de Dados e Estruturas de Dados

3.1 O que é um tipo de dado?

A linguagem Python reconhece diferentes tipos de dados. Para encontrar o tipo de uma variável, use a função `type()`:

```
In [5]: a = '45'
        b = 45

        int(a)+b

        print(a)
        print(b)
        type(a),type(b)
```

```
45
45
```

```
Out[5]: (str, int)
```

```
In [9]: b = 'Isto é uma string'
        type(b)
```

```
Out[9]: str
```

Devo me lembrar que um número complexo é composto de uma parte real e um a parte imaginária. Em Python, eu escrevo um NC assim:

```
In [1]: c = 2 + 1j
        type(c)
```

```
Out[1]: complex
```

```
In [16]: d = [1, 3, 56]
         type(d)
```

```
Out[16]: list
```

3.2 Números

Informação adicional

- Introdução informal aos números. Python tutorial, seção 3.1.1
- Python Library Reference: visão geral formal de tipos numéricos, `http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex`
- Pense em Python, Seção 2.1

Os tipos de dados predefinidos são números inteiros, ponto flutuante e ponto flutuante complexos.

3.2.1 Inteiros

Se precisarmos converter uma *string* contendo um número inteiro para um inteiro, podemos usar a função `int()`:

```
In [22]: a = '34'       # a é uma string contendo os caracteres 3 e 4
         x = int(a)     # x é um número inteiro
```

A função `int()` também converterá números em ponto flutuante para inteiros:

```
In [23]: int(7.0)
```

```
Out[23]: 7
```

```
In [24]: int(7.9)
```

```
Out[24]: 7
```

Note que `int` truncará qualquer parte não inteira de um número em ponto flutuante. Para arredondar um número em ponto flutuante para inteiro, use o comando `round()`:

```
In [25]: round(7.9)
```

```
Out[25]: 8
```

3.2.2 Limites de Inteiro

Inteiros em Python 3 são ilimitados; o interpretador Python automaticamente atribuirá tanta memória quanto necessária à medida que os números ficarem maiores. Isto significa que calculamos números muito grandes sem passos especiais.

```
In [28]: 35**42

Out[28]: 709345573078604437117360980259891332480037817731499671936035156 25
```

Em muitas outras linguagens de programação, tais como C e FORTRAN, inteiros ocupam um tamanho fixo de 4 bytes, permitindo 2^{32} valores diferentes, mas tipos diferentes estão disponíveis com diferentes tamanhos. Para números que cabem dentro desses limites, os cálculos podem ser mais rápidos, mas você pode ter de verificar que os números não excedam os limites. Calcular um número além dos limites é o mesmo que incorrer em *overflow de inteiro*, e pode produzir resultados estranhos.

Mesmo em Python, precisamos estar cientes disto quando usamos o NumPy. O NumPy usa inteiros com um tamanho fixo porque ele armazena muitos deles juntos e precisa calculá-los eficientemente. A documentação sobre tipos de dados do NumPy inclui uma gama de tipos inteiros denominados pelo tamanho deles. Então, `int16`, por exemplo, é um inteiro de 16 bits, com 2^{16} valores possíveis.

Inteiros podem também ser do tipo *signed*, quando permitem valores positivos ou negativos, ou do tipo *unsigned*, quando permitem apenas valores positivos. Por exemplo:

- `uint16` (*unsigned*) varia de 0 a $2^{16} - 1$
- `int16` (*signed*) varia de -2^{15} a $2^{15} - 1$

3.2.3 Números em Ponto Flutuante

Uma string contendo um número em ponto flutuante pode ser convertida em um ponto flututante usando o comando `float()`:

```
In [29]: a = '35.342'
         b = float(a)
         b

Out[29]: 35.342

In [30]: type(b)

Out[30]: float
```

3.2.4 Números Complexos

A linguagem Python (assim como Fortran e Matlab) possui números complexos predefinidos. Aqui estão alguns exemplos sobre como usá-los:

```
In [31]: x = 1 + 3j
         x

Out[31]: (1+3j)

In [32]: abs(x)                # calcula o valor absoluto
```

```
Out[32]: 3.1622776601683795

In [33]: x.imag

Out[33]: 3.0

In [34]: x.real

Out[34]: 1.0

In [35]: x * x

Out[35]: (-8+6j)

In [36]: x * x.conjugate()

Out[36]: (10+0j)

In [37]: 3 * x

Out[37]: (3+9j)
```

Note que se você desejar executar operações mais complicadas (tais como calcular a raiz quadrada, etc.) você terá que usar o módulo `cmath` (Complex MATHematics):

```
In [39]: import cmath

         cmath.sqrt(x)

Out[39]: (1.442615274452683+1.0397782600555705j)
```

3.2.5 Funções aplicáveis a todos os tipos de números

A função `abs()` retorna o valor absoluto (módulo) de um número:

```
In [50]: a = -45.463
         abs(a)
```

Note que `abs()` também funciona para números complexos (veja acima).

3.3 Sequencias

Sequencias de caracteres (`string`), listas (`lists`) e tuplas (`tuples`) são *sequencias*. Elas podem ser *indexadas* e *fatiadas* da mesma maneira.

Tuplas e *strings* são "imutáveis" (*immutable*). Basicamente, isto significa que não podemos alterar elementos individuais dentro da tupla, nem alterar caracteres individuais dentro de uma *string*, ao passo que listas são "mutáveis"(*mutable*), ou seja, podemos alterar seus elementos.

As sequencias compartilham as seguintes operações:

`a[i]`
retorna o *i*-ésimo elemento de `a`
`a[i:j]`
retorna elementos de *i* até *j*1
`len(a)`

retorna o número de elementos na sequencia
`min(a)`
retorna o menor valor na sequencia
`max(a)`
retorna o maior valor na sequencia
`x in a`
retorna verdadeiro (`True`) se `x` for um elemento de `a`
`a + b`
concatena `a` e `b`
`n * a`
cria `n` cópias da sequencia `a`

3.3.1 Tipo de Sequencia 1: String

Informação adicional

- Introdução às strings, Python tutorial 3.1.2

Uma *string* (imutável) pode ser definida usando aspas simples:

```
In [7]: a = 'Hello World'
```

aspas duplas:

```
In [ ]: a = "Hello World"
```

ou aspas triplas da mesma espécie

```
In [ ]: a = """Hello World"""
        a = '''Hello World'''
```

O tipo de uma *string* é `str` e uma *string* vazia é dada por :

```
In [ ]: a = "Hello World"
        type(a)

In [ ]: b = ""
        type(b)

In [ ]: type("Hello World")

In [ ]: type("")
```

O número de caracteres em uma *string* (isto é, seu *comprimento*) pode ser obtido usando a função `len()`:

```
In [8]: a = "Hello Moon"
        len(a)

Out[8]: 10

In [9]: a = 'teste'
        len(a)
```

```
Out[9]: 5

In [10]: len('outro teste')

Out[10]: 11
```

Você pode combinar ("concatenar") duas *strings* usando o operador +:

```
In [ ]: 'Hello ' + 'World'
```

Strings possuem um número de métodos úteis, incluindo, por exemplo `upper()`, que retorna a *string* em maiúsculas:

```
In [13]: a = "Esta é uma sentença de teste."
         a.upper()

Out[13]: 'ESTA É UMA SENTENÇA DE TESTE.'
```

Uma lista de métodos disponíveis para *strings* pode ser encontrada na documentação de referência. Se um *prompt* Python estiver disponível, deve-se usar as funções `dir` e `help` para recuperar esta informação, i.e. `dir()` fornece a lista de métodos; `help` pode ser usada para aprender sobre cada método.

Um método particularmente útil é `split()`, o qual converte uma string em uma lista de strings:

```
In [15]: a = "Esta é uma sentenca de teste."
         a.split()

Out[15]: ['Esta', 'é', 'uma', 'sentenca', 'de', 'teste.']
```

O método `split()` separará a string onde ele encontrar um *espaço em branco*. Um espaço em branco signifca qualquer caracter que é impresso como um espaço em branco, tais como um espaço, vários espaços ou uma indentação de parágrafo (tab).

Se um caracter separador for passaado como argumento para o método `split()`, uma string pode ser dividida em diferentes partes. Suponha que, por exemplo, queiramos obter uma lista de sentenças completas:

```
In [16]: a = "O cão está com fome. O gato está entendiado. A cobra está acordada."
         a.split(".")

Out[16]: ['O cão está com fome',
          ' O gato está entendiado',
          ' A cobra está acordada',
          '']
```

O método oposto a `split` é `join`, o qual pode ser usado como segue:

```
In [17]: a = "O cão está com fome. O gato está entendiado. A cobra está acordada."
         s = a.split('.')
         s

Out[17]: ['O cão está com fome',
          ' O gato está entendiado',
          ' A cobra está acordada',
          '']
```

```
In [18]: ".".join(s)

Out[18]: 'O cão está com fome. O gato está entendiado. A cobra está acordada.'

In [19]: " PARE".join(s)

Out[19]: 'O cão está com fome PARE O gato está entendiado PARE A cobra está acordada PARE'
```

3.3.2 Tipo de sequencia 2 : Lista

Informação adicional

- Introdução a Listas, Python tutorial, seção 3.1.4

Uma lista é uma sequencia de objetos. Os objetos podem ser de qualquer tipo. Por exemplo, inteiros:

```
In [ ]: a = [34, 12, 54]
```

ou *strings*:

```
In [ ]: a = ['cão', 'gato', 'rato']
```

Uma lista vazia é contruída com `[]`:

```
In [ ]: a = []
```

O tipo é `list`:

```
In [ ]: type(a)

In [ ]: type([])
```

Assim como *strings*, o número de elementos em uma lista pode ser obtido usando a função `len()`:

```
In [ ]: a = ['cão', 'gato', 'rato']
        len(a)
```

Também é possível *misturar* diferentes tipos de dados em uma mesma lista:

```
In [ ]: a = [123, 'pato', -42, 17, 0, 'elefante']
```

Em Python, uma lista é um objeto. Portanto, é possível que uma lista contenhaoutras listas (porque uma lista guarda uma sequencia de objetos):

```
In [ ]: a = [1, 4, 56, [5, 3, 1], 300, 400]
```

Você pode combinar ("concatenar") duas listas usando o operator +:

```
In [20]: [3, 4, 5] + [34, 35, 100]

Out[20]: [3, 4, 5, 34, 35, 100]
```

Ou você pode adicionar um objeto ao fim de uma lista usando o método `append()`:

```
In [21]: a = [34, 56, 23]
         a.append(42)
         a

Out[21]: [34, 56, 23, 42]
```

Você pode deletar um objeto a partir de uma lista chamando o método `remove()` e passando o objeto a ser deletado. Por exemplo:

```
In [22]: a = [34, 56, 23, 42]
         a.remove(56)
         a

Out[22]: [34, 23, 42]
```

O comando range() Um tipo especial de lista é frequentemente requerido (na maioria das vezes, junto com laços `for`) e, portanto, existe um comando para gerar essa lista: o comando `range(n)` gera inteiros começando de 0 e indo até n, exclusive (n não entra). Aqui estão alguns exemplos:

```
In [23]: list(range(3))

Out[23]: [0, 1, 2]

In [ ]: list(range(10))
```

Este comando é frequentemente usando com laços `for`. Por exemplo, para imprimir os números 02,12,22,32,...,102, o seguinte programa pode ser usado:

```
In [24]: for i in range(11):
             print(i ** 2)

0
1
4
9
16
25
36
49
64
81
100
```

O comando `range` toma um parâmetro opcional para ser o início da sequencia de inteiros (start) e outro parâmetro opcional para o tamanho do passo. Isto é frequentemente escrito como `range([start],stop,[step])`, onde os argumentos entre colchetes (*i.e.* start e step) são opcionais. Aqui estão alguns exemplos:

```
In [28]: list(range(3, 10))            # start=3

Out[28]: [3, 4, 5, 6, 7, 8, 9]
```

```
In [29]: list(range(3, 10, 2))          # start=3, step=2

Out[29]: [3, 5, 7, 9]

In [30]: list(range(10, 0, -1))         # start=10,step=-1

Out[30]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Por que estamos invocando `list(range())`?

Em Python 3, `range()` gera os números sob demanda. Quando você usa `range()` em um laço `for`, isto é mais eficiente porque ele não utiliza memória com uma lista de números. Passando-o para `list()`, nós o forçamos a gerar todos os seus números, assim vendo o que ele faz.

Para obter o mesmo comportamento eficiente em Python 2, use `xrange()` em vez de `range()`.

3.3.3 Tipo de Sequencia 3: Tuplas

Uma tupla (*tuple*) é uma sequencia (imutável) de objetos. Tuplas são muito similares em comportamento a listas com a exceção de que não podem ser modificadas (i.e. são imutáveis).

Por exemplo, os objetos em uma sequencia podem ser de qualquer tipo:

```
In [33]: a = (12, 13, 'cão')
         a

Out[33]: (12, 13, 'cão')

In [32]: a[0]

Out[32]: 12
```

Os parênteses não são necessários para definir uma tupla: apenas uma sequencia de objetos separadas por vírgulas é suficiente para definir uma tupla:

```
In [ ]: a = 100, 200, 'pato'
        a
```

embora seja boa prática de programação incluir os parênteses onde eles ajudam a mostrar que uma tupla está definida. Tuplas podem também ser usadas para fazer duas atribuições ao mesmo tempo:

```
In [ ]: x, y = 10, 20
        x

In [ ]: y
```

Isto pode ser usado para fazer um *swap* dos objetos. Por exemplo,

```
In [34]: x = 1
         y = 2
         x, y = y, x
         x

Out[34]: 2
```

```
In [35]: y

Out[35]: 1
```

A tupla vazia é dada por `()`

```
In [36]: t = ()
         len(t)

Out[36]: 0

In [37]: type(t)

Out[37]: tuple
```

A notação para uma tupla contendo um valor pode parecer um pouco estranha, em princípio:

```
In [38]: t = (42,)
         type(t)

Out[38]: tuple

In [ ]: len(t)
```

A vírgula adicional é necessária para distinguir `(42,)` de `(42)`, em que, no segundo caso, os parênteses seriam interpretados como um operador de precedência: `(42)` simplifica-se para 42, que é apenas um número:

```
In [39]: t = (42)
         type(t)

Out[39]: int
```

Este exemplo mostra a imutabilidade de uma tupla:

```
In [42]: a = (12, 13, 'cão')
         a[0]

Out[42]: 12

In [41]: a[0] = 1 # tenta-se alterar o primeiro elemento da tupla de 12 para 1

         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-41-edc0eb1f0e19> in <module>()
          ----> 1 a[0] = 1 # tenta-se alterar o primeiro elemento da tupla de 12 para 1


         TypeError: 'tuple' object does not support item assignment
```

A imutabilidade é a principal diferença entre uma tupla e uma lista, que é mutável. Devemos usar tuplas quando não quisermos que o conteúdo seja alterado.

Note que as funções em Python que retornam mais do que um valor, os retornam em tuplas (isto faz sentido porque você não deseja que estes valores sejam alterados).

3.3.4 Indexando sequencias

Informação adicional

- Introdução a *strings* e indexação: Python tutorial, seção 3.1.2, a seção relevante está começando depois que as strings foram apresentadas.

Objetos individuais em listas podem ser acessados usando o índice do objeto e colchetes (`[` e `]`):

```
In [43]: a = ['cão', 'gato', 'rato']
         a[0]

Out[43]: 'cão'

In [44]: a[1]

Out[44]: 'gato'

In [45]: a[2]

Out[45]: 'rato'
```

Note que em Python (assim como em C, mas diferentemente de Fortran e Matlab), o *índice começa a contar a partir de zero!*

Python fornece um atalho bastante útil para recuperar o último elemento em uma lista: para isto, usa-se o índice "-1", onde o sinal de menos indica que é um elemento *do final* da lista. Semelhantemente, o índice "-2" retornará o penúltimo elemento:

```
In [46]: a = ['cão', 'gato', 'rato']
         a[-1]

Out[46]: 'rato'

In [47]: a[-2]

Out[47]: 'gato'
```

Se você preferir, você pode pensar no índice `a[-1]` como uma notação compacta para `a[len(a) - 1]`.

Lembre-se que *strings* (assim como listas) também são um tipo de sequencia e podem ser indexadas da mesma maneira:

```
In [48]: a = "Hello World!"
         a[0]

Out[48]: 'H'

In [49]: a[1]

Out[49]: 'e'

In [50]: a[10]

Out[50]: 'd'

In [51]: a[-1]

Out[51]: '!'

In [52]: a[-2]

Out[52]: 'd'
```

3.3.5 Fatiando sequencias

Informação adicional

- Introdução a *strings*, indexação e fatiamento no Python tutorial, seção 3.1.2

O fatiamento (*slicing*) de sequencias pode ser usado para recuperar mais do que um elemento. Por exemplo:

```
In [53]: a = "Hello World!"
         a[0:3]

Out[53]: 'Hel'
```

Escrevendo `a[0:3]`, buscamos os 3 primeiros elementos começando do índice 0. Semelhantemente:

```
In [54]: a[1:4]

Out[54]: 'ell'

In [55]: a[0:2]

Out[55]: 'He'

In [56]: a[0:6]

Out[56]: 'Hello '
```

Podemos usar índices negativos para nos referirmos ao fim da sequencia:

```
In [62]: a[0:-1]

Out[62]: 'Hello World'
```

Também é possível ignorar índices do início ou fim da sequencia, assim retornando todos os elementos exceto os ignorados. Aqui estão alguns exemplos para tornar isto mais claro:

```
In [63]: a = "Hello World!"
         a[:5]

Out[63]: 'Hello'

In [64]: a[5:]

Out[64]: ' World!'

In [65]: a[-2:]

Out[65]: 'd!'

In [66]: a[:]
         b = a

Out[66]: 'Hello World!'
```

Note que `a[:]` gerará uma *cópia* de `a`. O uso de índices no fatiamento é feito por pessoas experientes em contagem intuitiva. Se você se sentir desconfortável com o fatiamento, dê uma olhada nesta citação do Python tutorial (seção 3.1.2):

> A melhor maneira de lembrar como as fatias funcionam é pensar como se os índices apontassem entre os caracteres, com a aresta esquerda do primeiro caracter numerada de 0. Então, a aresta direita do último caracter de uma *string* de 5 caracteres tem índice 5. Por exemplo:
>
> ```
>  +---+---+---+---+---+
>  | H | e | l | l | o |
>  +---+---+---+---+---+
>  0   1   2   3   4   5   <-- use para INDEXAÇÃO
> -5  -4  -3  -2  -1       <-- use para INDEXAÇÃO a partir do final
> ```
>
> A primeira linha de números fornece a posição dos índices de fatiamento 0...5 na string; a segunda linha fornece os índices negativos correspondentes. A fatia de i até j consiste de todos os caracteres entre as arestas rotuladas de i e j, respectivamente.

Assim, o ponto importante é que para o fatiamento (*slicing*), devemos pensar os índices como se apontassem entre os caracteres.

Para indexação (*indexing*), é melhor pensar como se os índices apontassem para os caracteres. Aqui está um pequeno resumo gráfico dessas regras:

```
   0   1   2   3   4    <-- use para INDEXAÇÃO
  -5  -4  -3  -2  -1    <-- use para INDEXAÇÃO a partir do final
 +---+---+---+---+---+
 | H | e | l | l | o |
 +---+---+---+---+---+
 0   1   2   3   4   5  <-- use para FATIAMENTO
-5  -4  -3  -2  -1      <-- use para FATIAMENTO a partir do final
```

Se você não estiver certo sobre qual é o índice correto, é sempre uma boa técnica brincar um pouco com um pequeno exemplo no *prompt* do Python para testar as coisas antes e/ou durante a escrita de seus programas.

3.3.6 Dicionários

Dicionários são também chamados de "arrays associativos" ou "tabelas hash"("hash tables"). Dicionários são conjuntos *não-ordenados* de pares *palavra-chave/valor* (*key/value*).

Um dicionário vazio pode ser criado usando chaves:

```
In [68]: d = {}
```

Pares palavra-chave/valor podem ser adicionados como:

```
In [69]: d['hoje'] = '22 graus C'    # 'hoje' é a palavra-chave

In [70]: d['ontem'] = '19 graus C'
```

`d.keys()` retorna uma lista de todas as palavras-chave:

```
In [71]: d.keys()

Out[71]: dict_keys(['hoje', 'ontem'])
```

Podemos recuperar valores usando a palavra-chave como índice:

```
In [72]: d['hoje']

Out[72]: '22 graus C'
```

Outras maneiras de preencher um dicionário se os dados são conhecidos no momento da criação são:

```
In [73]: d2 = {2:4, 3:9, 4:16, 5:25}
         d2

Out[73]: {2: 4, 3: 9, 4: 16, 5: 25}

In [74]: d3 = dict(a=1, b=2, c=3)
         d3

Out[74]: {'a': 1, 'b': 2, 'c': 3}
```

A função `dict()` cria um dicionário vazio.

Outros métodos úteis de dicionário incluem `values()`, `items()` e `get()`. Você pode usar `in` para verificar a presença de valores.

```
In [75]: d.values()

Out[75]: dict_values(['22 graus C', '19 graus C'])

In [76]: d.items()

Out[76]: dict_items([('hoje', '22 graus C'), ('ontem', '19 graus C')])

In [77]: d.get('hoje','desconhecido')

Out[77]: '22 graus C'

In [80]: d.get('amanhã','desconhecido')

Out[80]: 'desconhecido'

In [81]: 'hoje' in d

Out[81]: True

In [82]: 'amanhã' in d

Out[82]: False
```

O método `get(key,default)` fornecerá o valor para uma dada `key` se essa palavra-chave existir, caso contrário ela retornará o objeto `default`. Aqui está um exemplo mais complexo:

```
In [ ]: order = {}         # cria um dicionário vazio

        #adiciona pedidos à medida que chegam
        order['Pedro'] = 'Uma lata de cerveja'
        order['Paulo'] = 'Um suco de laranja'
        order['Maria'] = 'Uma lata de agua tônica'

        #entrega pedidos no bar
        for person in order.keys():
            print(person, "solicita", order[person])
```

Algumas tecnicalidades adicionais:

- A palavra-chave pode ser qualquer objeto Python (imutável). Nisto incluem:
  - ▷ números
  - ▷ strings
  - ▷ tuplas.
- dicionários são muito rápidos na recuperação de valores (uma vez dada a palavra-chave)

Um outro exemplo para demonstrar a vantagem de se usar dicionários em vez de pares de listas:

```
In [ ]: dic = {}                          # cria dicionário vazio

        dic["Helio"]    = "sala 1033"     # preenche dicionário
        dic["Anderson C"] = "sala 1031" # "Anderson C" é a chave
        dic["Kennedy"]    = "sala 1027" # "sala 1027" é o valor

        for key in dic.keys():
            print(key, "trabalha na", dic[key])
```

Sem o dicionário:

```
In [ ]: pessoas = ["Helio","Anderson C","Kennedy"]
        salas  = ["sala 1033","sala 1031","sala 1027"]

        # possivel inconsistencia aqui visto que temos duas listas
        if not len( pessoas ) == len( salas ):
            raise RuntimeError("pessoas e salas diferem em comprimento")

        for i in range( len( salas ) ):
            print(pessoas[i],"trabalha na",salas[i])
```

3.4 Passando argumentos para funções

Esta seção contém algumas ideias mais avançadas e faz uso de conceitos que serão somente apresentados mais tarde neste texto. A seção pode ser mais facilmente acessível em um estágio posterior.

Quando objetos são passados para uma função, o Python sempre passa (o valor da) referência do objeto para a função. Efetivamente, isto significa chamar a função *por referência*, embora nos refiramos a isto como chamar por *valor* (da referência).

Revisaremos a passagem de argumentos por valor e por referência antes de discutirmos outras situação em Python com maiores detalhes.

3.4.1 Passagem de argumento por valor

Pode-se esperar que, se passarmos um objeto por valor para uma função, as modificações desse valor dentro da função não afetarão o objeto (porque não passamos o objeto em si, mas apenas seu valor, que é uma cópia). Aqui está um exemplo desse comportamento (em C):

```
#include <stdio.h>

void pass_by_value(int m) {
  printf("in pass_by_value: received m=%d\n",m);
  m=42;
  printf("in pass_by_value: changed to m=%d\n",m);
}

int main(void) {
  int global_m = 1;
  printf("global_m=%d\n",global_m);
  pass_by_value(global_m);
  printf("global_m=%d\n",global_m);
  return 0;
}
```

juntamente com a saída correspondente:

```
global_m=1
in pass_by_value: received m=1
in pass_by_value: changed to m=42
global_m=1
```

O valor 1 da variável global `global_m` não é modificado quando a função `pass_by_value` altera seu argumento de entrada para 42.

3.4.2 Passagem de argumento por referência

Chamar uma função por referência, por outro lado, significa que o objeto fornecido a uma função é uma referência ao objeto. Isso significa que a função verá o mesmo objeto que no código de chamada (porque eles estão referenciando o mesmo objeto: podemos pensar na referência como um ponteiro para o local na memória onde o objeto está localizado). Qualquer alteração que atue sobre o objeto dentro da função, será visível no objeto no nível de chamada (porque a função realmente opera no mesmo objeto, e não em uma cópia dele).

Aqui está um exemplo que mostra isso usando ponteiros em C:

```
#include <stdio.h>

void pass_by_reference(int *m) {
  printf("in pass_by_reference: received m=%d\n",*m);
  *m=42;
  printf("in pass_by_reference: changed to m=%d\n",*m);
}

int main(void) {
  int global_m = 1;
```

```
  printf("global_m=%d\n",global_m);
  pass_by_reference(&global_m);
  printf("global_m=%d\n",global_m);
  return 0;
}
```

juntamente com a saída correspondente:

```
global_m=1
in pass_by_reference: received m=1
in pass_by_reference: changed to m=42
global_m=42
```

O C++ fornece a capacidade de passar argumentos como referências adicionando um "&"à frente do nome do argumento na definição da função:

```
#include <stdio.h>

void pass_by_reference(int &m) {
  printf("in pass_by_reference: received m=%d\n",m);
  m=42;
  printf("in pass_by_reference: changed to m=%d\n",m);
}

int main(void) {
  int global_m = 1;
  printf("global_m=%d\n",global_m);
  pass_by_reference(global_m);
  printf("global_m=%d\n",global_m);
  return 0;
}
```

juntamente com a saída correspondente:

```
global_m=1
in pass_by_reference: received m=1
in pass_by_reference: changed to m=42
global_m=42
```

3.4.3 Passagem de argumento em Python

Em Python, os objetos são passados como o valor de uma referência (ponteiro) para o objeto. Dependendo da forma como a referência é usada na função e, dependendo do tipo de objeto que ele referencia, isso pode resultar em um comportamento de "passagem por referência"(onde qualquer alteração no objeto recebido como argumento de função é imediatamente refletida no nível de chamada).

Aqui estão três exemplos para discutir isso. Começamos passando uma lista para uma função que itera através de todos os elementos na sequência e dobra o valor de cada elemento:

```
In [ ]: def double_the_values(l):
            print("in double_the_values: l = %s" % l)
```

```
        for i in range(len(l)):
            l[i] = l[i] * 2
        print("in double_the_values: changed l to l = %s" % l)

    l_global = [0, 1, 2, 3, 10]
    print("In main: s=%s" % l_global)
    double_the_values(l_global)
    print("In main: s=%s" % l_global)
```

A variável `l` é uma referência ao objeto lista. A linha `l[i] = l[i] * 2` avalia primeiro o lado direito e lê o elemento com o índice `i`, depois multiplica por dois. Uma referência a este novo objeto é então armazenada no objeto lista `l` na posição com o índice `i`. Por isso, modificamos o objeto lista, que é referenciado através de `l`.

A referência ao objeto lista nunca muda: a linha `l[i] = l[i] * 2` muda os elementos `l[i]` da lista `l` mas nunca a referência `l` para a lista. Assim, tanto a função como o nível de chamada estão operando no mesmo objeto através das referências `l` e`global_l`, respectivamente.

Em contraste, aqui está um exemplo que não modifica os elementos da lista dentro da função, o que produz esta saída:

```
In [ ]: def double_the_list(l):
            print("in double_the_list: l = %s" % l)
            l = l + l
            print("in double_the_list: changed l to l = %s" % l)

        l_global = "Hello"
        print("In main: l=%s" % l_global)
        double_the_list(l_global)
        print("In main: l=%s" % l_global)
```

O que acontece aqui é que, durante a avaliação de `l = l + l`, um novo objeto é criado que contém `l + l`, e que, em seguida, vinculamos o nome `l` a ele. No processo, perdemos as referências à lista `l` que foi passada à função (e, portanto, não alteramos a lista passada à função).

Finalmente, vejamos o que essa saída produz:

```
In [ ]: def double_the_value(l):
            print("in double_the_value: l = %s" % l)
            l = 2 * l
            print("in double_the_values: changed l to l = %s" % l)

        l_global = 42
        print("In main: s=%s" % l_global)
        double_the_value(l_global)
        print("In main: s=%s" % l_global)
```

Neste exemplo, também duplicamos o valor (de 42 a 84) dentro da função. No entanto, quando vinculamos o objeto 84 ao nome `l` (que é a linha `l = l * 2`), criamos um novo objeto (84), e nós vinculamos o novo objeto a `l`. No processo, perdemos a referência ao objeto 42 dentro da função. Isso não afeta o próprio objeto 42, nem a referência `l_global` para ele.

Em resumo, o comportamento em Python de passar argumentos para uma função pode parecer variável (se o visualizarmos a partir do ponto de vista de *passagem por valor* versus *passagem por referência*). No entanto, ele é sempre feito como uma chamada por valor, onde o valor é uma referência ao objeto em questão, e o comportamento pode ser explicado pelo mesmo raciocínio em todos os casos.

3.4.4 Considerações de desempenho

As chamadas de função por valor requerem a cópia do valor antes de ele ser passado à função. Sob o ponto de vista de desempenho (tempo de execução e requisitos de memória), isso pode ser um processo caro se o valor for grande. (Imagine que o valor é um objeto `numpy.array` que pode ser de vários Megabytes ou Gigabytes de tamanho.)

Geralmente se preferem chamadas por referência para objetos grandes, pois neste caso apenas um ponteiro para os objetos é passado, independentemente do tamanho real do objeto e, portanto, isso geralmente é mais rápido do que a chamada por valor.

A abordagem em Python de (efetivamente) chamar por referência é, portanto, eficiente. No entanto, precisamos ter cuidado para que nossa função não modifique os dados que ela receber quando isso não é desejado.

3.4.5 Modificação de dados por desatenção

Geralmente, uma função não deve modificar os dados passados como entrada para ela.

Por exemplo, o código a seguir demonstra a tentativa de determinar o valor máximo de uma lista, e uma modificação - desatenta - da lista no processo:

```
In [ ]: def mymax(s):  # demonstrando efeito colateral
            if len(s) == 0:
                raise ValueError('mymax() arg is an empty sequence')
            elif len(s) == 1:
                return s[0]
            else:
                for i in range(1, len(s)):
                    if s[i] < s[i - 1]:
                        s[i] = s[i - 1]
                return s[len(s) - 1]

        s = [-45, 3, 6, 2, -1]
        print("in main before caling mymax(s): s=%s" % s)
        print("mymax(s)=%s" % mymax(s))
        print("in main after calling mymax(s): s=%s" % s)
```

O usuário da função `mymax()` não esperaria que o argumento de entrada fosse modificado quando a função fosse executada. Em geral, devemos evitar isso. Existem várias maneiras de encontrar melhores soluções para o problema dado:

- Neste caso particular, poderíamos usar a função predefinida `max()` para obter o valor máximo de uma sequência.
- Se sentirmos que precisamos nos ater ao armazenamento de valores temporários dentro da lista [isso na verdade não é necessário], podemos criar uma cópia da lista de entrada `s` primeiro e, em seguida, proceder com o algoritmo (veja adiante sobre cópia de objetos).
- Escolha outro algoritmo, o qual use uma variável temporária extra ao invés de abusar da lista para isso. Por exemplo:
- Podemos passar uma tupla (em vez de uma lista) para a função: uma tupla é *imutável* e, portanto, nunca pode ser modificada (isso resultaria em uma exceção sendo gerada quando a função tentasse escrever nos elementos da tupla).

3.4.6 Copiando objetos

Python fornece a função `id()`, a qual retorna um número inteiro que é exclusivo para cada objeto. (Na implementação atual do CPython, este é o endereço da memória.) Podemos usá-la para identificar se dois objetos são iguais.

Para copiar um objeto cujo tipo é uma sequência (incluindo listas), podemos fatiá-lo, i.e. se `a` for uma lista, então `a[:]` retornará uma cópia de `a`. Aqui está uma demonstração:

```
In [ ]: a = list(range(10))
        a

In [ ]: b = a
        b[0] = 42
        a               # alterando-se b, altera-se a

In [ ]: id(a)

In [ ]: id(b)

In [ ]: c = a[:]
        id(c)           # c é um objeto diferente

In [ ]: c[0] = 100
        a               # altera-se c, mas a permanece inalterado
```

A biblioteca padrão em Python fornece o módulo `copy`, o qual provê funções de cópia que podem ser usadas para criar cópias de objetos. Poderíamos ter usado `import copy; c = copy.deepcopy(a)` em vez de `c = a[:]`.

3.5 Igualdade e Identidade

Uma questão relacionada diz respeito à igualdade de objetos.

3.5.1 Igualdade

Os operadores `<`, `>`, `==`, `>=`, `<=`, e `!=` comparam os *valores* de dois objetis. Os objetos não precisam ter o mesmo tipo. Por exemplo:

```
In [ ]: a = 1.0; b = 1
        type(a)

In [ ]: type(b)

In [ ]: a == b
```

Então, o operador `==` verifica se os valores de dois objetos são iguais.

3.5.2 Identidade/Semelhança

Para verificar se dois objetos `a` e `b` são os mesmos (i.e. se `a` e `b` fazem referência ao mesmo local na memória), podemos usar o operator `is` (continuação do exemplo anterior):

```
In [ ]: a is b
```

Certamente, eles são diferentes aqui, já que não são do mesmo tipo.

Também podemos lançar mão da função `id` que, de acordo com a documentação da versão Python 2.7: "*Retorna a identidade de um objeto, única entre objetos existindo simultaneamente. (Dica: é o endereço do objeto na memória.)*"

```
In [ ]: id(a)

In [ ]: id(b)
```

que mostra que `a` e `b` são armazenados em diferentes locais na memória.

3.5.3 Exemplo: Igualdade e identidade

Fechamos com um exemplo envolvendo listas:

```
In [ ]: x = [0, 1, 2]
        y = x
        x == y

In [ ]: x is y

In [ ]: id(x)

In [ ]: id(y)
```

Aqui, `x` e `y` são referências ao mesmo endereço na memória. Eles são idênticos e o operador `is` confirma isso. O ponto importante a lembrar é que a linha 2 (`y = x`) cria uma nova referência `y` para a mesma lista da qual `x` já é uma referência.

Consequentemente, podemos alterar os elementos de `x` que `y` mudará simultaneamente, pois `x` e `y` se referem ao mesmo objeto:

```
In [ ]: x

In [ ]: y

In [ ]: x is y

In [ ]: x[0] = 100
        y

In [ ]: x
```

Em contraste, se usarmos `z = x [:]` (em vez de `z = x`) para criar um novo objeto `z`, então a operação de fatiamento `x[:]` criará uma cópia da lista `x`, e a nova referência `z` apontará para esta cópia. O *valor* de `x` e `z` será o mesmo, mas `x` e `z` não serão o mesmo objeto (eles não serão idênticos):

```
In [ ]: x
```

```
In [ ]: z = x[:]              # cria cópia de x antes de atribui-lo a z
        z == x               # mesmo valor

In [ ]: z is x               # nao sao o mesmo objeto

In [ ]: id(z)                # confirmacao pela id()

In [ ]: id(x)

In [ ]: x

In [ ]: z
```

Consequentemente, podemos alterar `x` sem que `z` seja alterado. Por exemplo (continuação):

```
In [ ]: x[0] = 42
        x

In [ ]: z
```

4 Introspecção

Um código Python pode fazer e responder perguntas sobre si mesmo e sobre os objetos que ele manipula.

4.1 dir()

`dir()` é uma função predefinida que retorna uma lista de todos os nomes pertencentes a algum espaço de nomes (*namespace*).

- Se nenhum argumento for passado para `dir` (isto é, `dir()`), ele inspeciona o *namespace* no qual foi chamado.
- Se `dir` receber um argumento (ou seja, `dir(<object>)`, ele inspeciona o *namespace* do objeto que foi passado.

Por exemplo:

```
In [18]: dir()

Out[18]: ['In',
          'Out',
          '_',
          '_1',
          '_10',
          '_11',
          '_12',
          '_13',
          '_17',
          '_2',
          '_3',
          '_5',
```

```
 '_6',
 '_7',
 '_8',
 '_9',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i10',
 '_i11',
 '_i12',
 '_i13',
 '_i14',
 '_i15',
 '_i16',
 '_i17',
 '_i18',
 '_i2',
 '_i3',
 '_i4',
 '_i5',
 '_i6',
 '_i7',
 '_i8',
 '_i9',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'a',
 'b',
 'exit',
 'get_ipython',
 'macas',
 'math',
 'my_int',
 'nome',
 'quit']
```

```
In [19]: nome = "Pedro"
         dir(nome)
```

```
Out[19]: ['__add__',
```

```
'__class__',
'__contains__',
'__delattr__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__getitem__',
'__getnewargs__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
```

```
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

4.1.1 Nomes Mágicos

Você encontrará muitos nomes que começam e terminam com um sublinhado duplo (por exemplo, `__name__`). Estes são chamados de *nomes mágicos*. Funções com nomes mágicos fornecem a implementação de funcionalidades particulares da linguagem Python.

Por exemplo, a aplicação de `str` a um objeto `a`, ou seja `str(a)`, resultará - internamente - na chamada do método `a.__str__()`. O método `__str__` geralmente precisa retornar uma *string*. A ideia é que o método `__str__()` deve ser definido para todos os objetos (incluindo aqueles que derivam de novas classes que um programador pode criar) de modo que todos os objetos (independentemente de seu tipo ou classe) pode ser impresso usando a função `str()`. A conversão real de algum objeto `x` para a *string* é então feita através do método específico do objeto `x.__str__()`.

Podemos demonstrar isso criando uma classe `my_int` que herda da classe base de número inteiro do Python e substitui o método `__str__`. (Isto requer mais conhecimento de Python do que o fornecido até este ponto no texto para poder entender este exemplo.)

```
In [20]: class my_int(int):
             """Herdada de int"""
             def __str__(self):
                 """ Representacao adaptada de str para a classe my_int"""
                 return "my_int: %s" % (int.__str__(self))

         a = my_int(3)
         b = int(4)            # equivalente a b = 4
         print("a * b = ", a * b)
```

```
        print("Type a = ", type(a), "str(a) = ", str(a))
        print("Type b = ", type(b), "str(b) = ", str(b))
```

```
a * b =  12
Type a =  <class '__main__.my_int'> str(a) =  my_int: 3
Type b =  <class 'int'> str(b) =  4
```

Leitura complementar Veja Documentação Python, Modelos de Dados

4.2 Tipo (*type*)

O comando `type(<object>)` retorna o tipo de um objeto:

```
In [21]: type(1)
```

```
Out[21]: int
```

```
In [22]: type(1.0)
```

```
Out[22]: float
```

```
In [23]: type("Python")
```

```
Out[23]: str
```

```
In [24]: import math
         type(math)
```

```
Out[24]: module
```

```
In [25]: type(math.sin)
```

```
Out[25]: builtin_function_or_method
```

4.3 isinstance

`isinstance(<object>, <typespec>)` retorna verdadeiro (`True`) se o objeto passado é uma instância do tipo de dado passado, ou de qualquer uma de suas superclasses. Use `help(isinstance)` para a sintaxe completa.

```
In [26]: isinstance(2,int)
```

```
Out[26]: True
```

```
In [27]: isinstance(2.,int)
```

```
Out[27]: False
```

```
In [28]: isinstance(a,int)     # a é uma instância de my_int
```

```
Out[28]: True
```

```
In [29]: type(a)
```

```
Out[29]: __main__.my_int
```

4.4 Obtendo ajuda com `help`

- A função `help` reportará o *docstring* (atributo mágico com nome `__doc__` do objeto que é passado, às vezes complementado com informação adicional. No caso de funções, `help` também mostrará a lista de argumentos que a função aceita (mas não fornecerá o valor de retorno).
- `help()` inicializa um ambiente interativo de ajuda.
- É comum usar o comando `help` muitas vezes para se lembrar da sintaxe e semântica dos comandos.

```
In [30]: help(isinstance)
```

```
Help on built-in function isinstance in module builtins:

isinstance(obj, class_or_tuple, /)
    Return whether an object is an instance of a class or of a subclass thereof.

    A tuple, as in ``isinstance(x, (A, B, ...))``, may be given as the target to
    check against. This is equivalent to ``isinstance(x, A) or isinstance(x, B)
    or ...`` etc.
```

```
In [31]: import math
         help(math.sin)
```

```
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

A função `help` depende do nome de um objeto (que deve existir no espaço de nomes corrente). Por exemplo, help(math.sqrt) não funcionará se o módulo `math` não tiver sido importado antes.

```
In [33]: del math # desligando o modulo math (omp anteriormente) para destacar o erro.
         help(math.sqrt)
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-33-e3a778a8af20> in <module>()
          1 del math
    ----> 2 help(math.sqrt) # reinicie o kernel e execute esta célula antes das demais para


        NameError: name 'math' is not defined
```

```
In [34]: import math
         help(math.sqrt)

Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Em vez de importar o módulo, poderíamos ter também passado a *string* `math.sqrt` para a função `help`, i.e.:

```
In [35]: help('math.sqrt')

Help on built-in function sqrt in math:

math.sqrt = sqrt(...)
    sqrt(x)

    Return the square root of x.
```

- `help` é uma função que fornece informações sobre o objeto que é passado como seu argumento. A maioria das coisas em Python (classes, funções, módulos, etc.) são objetos e, por isso, podem ser passados para a função `help`. Há, no entanto, algumas coisas para as quais você gostaria de ajuda, mas que não são objetos existentes em Python. Nesses casos, muitas vezes é possível passar uma *string* contendo o nome da coisa ou conceito para a função `help`, por exemplo.
- `help('modules')` gerará uma lista de todos os módulos que podem ser importados para o interpretador corrente. Note que `help(modules)` (note a ausência de aspas) resultará em um `NameError` (a menos que você tenha a má sorte de ter uma variável chamada módulos em seu ambiente de nomes, caso em que você obterá ajuda a respeito dessa variável).
- `help('algum_modulo')`, onde `algum_modulo` é um módulo que ainda não foi importado (e ainda não é um objeto), lhe dará as informações de ajuda desse módulo.
- `help('alguma_keyword')`: por exemplo 'and', 'if' ou 'print', isto é, `help('and')`, `help('if')` e `help('print')`. Estas são palavras-chave especiais reconhecidas em Python: elas não são objetos e, portanto, não podem ser passadas como argumentos para `help`. Passe o nome da palavra-chave como uma *string* para `help` funcionar, mas somente se você tiver a documentação em `html` instalada e se o interpretador Python tiver sido informado da localização da documentação ao se definir a variável de ambiente PYTHONDOCS.

4.5 *Docstrings* (*strings* de documentação)

O comando `help(<object>)` acessa as *strings* de documentação de objetos.

Qualquer *string* literal aparecendo como o primeiro item na definição de uma classe, função, método ou módulo, é considerada como sua *docstring*.

`help` inclui a docstring na informação que ela exibe sobre o objeto. Além da *docstring*, ela pode exibir algumas outras informações. Por exemplo, no caso de funções, ela exibe a assinatura da função. A *docstring* é armazenada no atributo `__doc__` do objeto.

```
In [36]: help(math.sin)
         # Help on built-in function sin in module math (<-- texto da docstring)

Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).



In [37]: print(math.sin.__doc__)

sin(x)

Return the sine of x (measured in radians).
```

Para funções, classes tipos, módulos, etc. definidos pelo usuário, deve-se sempre fornecer uma *docstring*.

Documentando uma função definida pelo usuário:

```
In [38]: def power2and3(x):
             """Retorna a tupla (x**2, x**3)"""
             return x**2 ,x**3

         power2and3(2)

Out[38]: (4, 8)

In [39]: power2and3(4.5)

Out[39]: (20.25, 91.125)

In [40]: power2and3(0+1j)

Out[40]: ((-1+0j), (-0-1j))

In [41]: help(power2and3)

Help on function power2and3 in module __main__:

power2and3(x)
    Retorna a tupla (x**2, x**3)



In [42]: print(power2and3.__doc__)

Retorna a tupla (x**2, x**3)
```

5 Entrada e Saída de Dados

Nesta seção, descrevemos o processo de impressão na tela, o qual inclui o uso da função `print`, o especificador de formato do estilo antigo `%` e o especificador de formato do novo estilo `{}`.

5.1 Imprimindo na saída padrão (normalmente a tela)

A função `print` é o comando mais usado para imprimir informações para o "dispositivo de saída padrão", que normalmente é a tela.

Existem dois modos de usar a impressão.

5.1.1 Impressão simples

A maneira mais fácil de usar o comando de impressão é listando as variáveis a serem impressas, separadas por vírgulas. Aqui estão alguns exemplos:

```
In [1]: a = 10
        b = 'texto de teste'
        print(a)
```

```
10
```

```
In [2]: print(b)
```

```
texto de teste
```

```
In [3]: print(a, b)
```

```
10 texto de teste
```

```
In [4]: print("A resposta é", a)
```

```
A resposta é 10
```

```
In [5]: print("A resposta é", a, "e a string contém", b)
```

```
A resposta é 10 e a string contém texto de teste
```

```
In [6]: print("A resposta é", a, "e a string lida como", b)
```

```
A resposta é 10 e a string lida como texto de teste
```

Em Python, um espaço é adicionado entre cada objeto que está sendo impresso. Uma nova linha é impressa após cada chamada. Para suprimir isso, use o parâmetro `end =`:

```
In [7]: print("Imprimindo na linha", end ='')
        print("... ainda imprimindo na mesma linha.")
```

```
Imprimindo na linha... ainda imprimindo na mesma linha.
```

5.1.2 Impressão formatada

A forma mais sofisticada de formatar a saída usa uma sintaxe muito semelhante ao `fprintf` do Matlab (e, portanto, também, semelhante ao `printf` da linguagem C). A estrutura geral é a de uma *string* contendo especificadores de formato, seguida por um sinal de porcentagem e uma tupla que contém as variáveis a serem impressas no lugar dos especificadores de formato.

```
In [8]: print("a = %d b = %d" % (10,20))

a = 10 b = 20
```

Uma *string* pode conter identificadores de formato (ex: `%f` para formatar como `float`,`%d` para formatar como número inteiro e `%s` para formatar como *string*, isto é, uma sequência de caracteres):

```
In [9]: from math import pi
        print("Pi = %5.2f" % pi)

Pi =  3.14


In [10]: print("Pi = %10.3f" % pi)

Pi =      3.142


In [11]: print("Pi = %10.8f" % pi)

Pi = 3.14159265


In [12]: print("Pi = %d" % pi)

Pi = 3
```

O especificador de formato do tipo `%W.Df` significa que um `float` deve ser impresso com uma largura total de W caracteres e D digitos após o ponto decimal. (Isto é idêntico ao Matlab e C, por exemplo).

Para imprimir mais de um objeto, forneça vários especificadores de formato e liste vários objetos na tupla:

```
In [13]: print("Pi = %f, 142*pi = %f e pi^2 = %f." % (pi,142*pi,pi**2))

Pi = 3.141593, 142*pi = 446.106157 e pi^2 = 9.869604.
```

Note que a conversão de um especificador de formato e uma tupla de variáveis para uma *string* não depende do comando `print`:

```
In [14]: from math import pi
         "pi = %f" % pi

Out[14]: 'pi = 3.141593'
```

Isso significa que podemos converter objetos em *strings* onde quer que precisemos, e podemos decidir imprimir as *strings* mais tarde - não há necessidade de acoplar a formatação ao código que faz a impressão.

Visão geral dos especificadores de formato comumente usados (exemplo: unidade astronômica):

```
In [15]: AU = 149597870700  # unidade astronomica [m]
         "%f" % AU          # linha1 na tabela

Out[15]: '149597870700.000000'
```

especificador	estilo	examplo da saída para AU
`%f`	ponto flutante	`149597870700.000000`
`%e`	notação exponencial	`1.495979e+11`
`%g`	mais curta entre `%e` ou `%f`	`1.49598e+11`
`%d`	inteiro	`149597870700`
`%s`	`str()`	`149597870700`
`%r`	`repr()`	`149597870700L`

5.1.3 `str` e `__str__`

Todos os objetos em Python devem fornecer um método `__str__` que retorne uma boa representação de *string* do objeto. Este método `a.__str__()` é chamado quando aplicamos a função `str` ao objeto `a`:

```
In [16]: a = 3.14
         a.__str__()

Out[16]: '3.14'

In [17]: str(a)

Out[17]: '3.14'
```

A função `str` é extremamente conveniente já que nos permite escrever objetos mais complicados, tais como

```
In [18]: b = [3, 4.2, ['maçã', 'banana'], (0, 1)]
         str(b)

Out[18]: "[3, 4.2, ['maçã', 'banana'], (0, 1)]"
```

A forma como isto é impresso em Python é usando o método `__str__` do objeto da lista. O interpretador imprimirá o colchete de abertura `[` e depois chamará o método `__str__` do primeiro objeto, ou seja, o inteiro 3, produzindo `3`. Em seguida, o método `__str__` do objeto da lista imprime a vírgula `,`, move-se para chamar o método `__str__` do próximo elemento da lista (ou seja,`4.2`) para imprimi-lo. Desta forma, qualquer objeto composto pode ser representado como uma *string*, pedindo aos objetos que ele guarda para convertê-los em *strings*.

O método *string* do objeto `x` é chamado de forma implícita quando

- usamos o especificador de formato `%s` para imprimir `x`;

- passamos o objeto x diretamente para o comando de impressão:

```
In [19]: print(b)

[3, 4.2, ['maçã', 'banana'], (0, 1)]


In [20]: print("%s" % b)

[3, 4.2, ['maçã', 'banana'], (0, 1)]
```

5.1.4 repr e __repr__

Uma segunda função, repr, deve converter um determinado objeto em uma representação de string *para que essa sequencia de caracteres possa ser usada para recriar o objeto usando a função* `eval`. A função repr geralmente fornecerá uma string mais detalhada do que str. A aplicação de repr ao objeto x tentará chamar x.__repr__().

```
In [21]: from math import pi as a1
         str(a1)

Out[21]: '3.141592653589793'

In [22]: repr(a1)

Out[22]: '3.141592653589793'

In [23]: numero_como_string = repr(a1)
         a2 = eval(numero_como_string)  # avalia a string
         a2

Out[23]: 3.141592653589793

In [24]: a2-a1                         # -> repr é uma representação exata

Out[24]: 0.0

In [25]: a1-eval(repr(a1))

Out[25]: 0.0

In [26]: a1-eval(str(a1))              # -> str perdeu alguns dígitos

Out[26]: 0.0
```

Podemos converter um objeto para a sua forma str() ou repr() usando os especificadores de formato %s e %r, respectivamente.

```
In [27]: import math
         "%s" % math.pi

Out[27]: '3.141592653589793'

In [28]: "%r" % math.pi

Out[28]: '3.141592653589793'
```

5.1.5 Nova formatação

Um novo sistema predefinido de formatação permite mais flexibilidade para casos complexos, com o custo de ser um pouco mais longo.

Idéias básicas em exemplos:

```
In [29]: "{} precisa de {} chocolates".format('Pedro', 4)      # insere valores na ordem

Out[29]: 'Pedro precisa de 4 chocolates'

In [30]: "{0} precisa de {1} chocolates".format('Pedro', 4)    # indexa elemento

Out[30]: 'Pedro precisa de 4 chocolates'

In [31]: "{1} precisa de {0} chocolates".format('Pedro', 4)

Out[31]: '4 precisa de Pedro chocolates'

In [32]: # referencia elemento para impressão por nome
         "{nome} precisa de {numero} chocolates".format(nome='Pedro',numero=4)

Out[32]: 'Pedro precisa de 4 chocolates'

In [33]: "Pi é aproximadamente {:f}.".format(math.pi)      # podemos usar opções de formataç

Out[33]: 'Pi é aproximadamente 3.141593.'

In [34]: "Pi é aproximadamente {:.2f}.".format(math.pi)    # e precisao

Out[34]: 'Pi é aproximadamente 3.14.'

In [35]: "Pi é aproximadamente {:6.2f}.".format(math.pi)   # e largura

Out[35]: 'Pi é aproximadamente   3.14.'
```

Esta é uma maneira poderosa e elegante de formatação de *strings* que está tendo seu uso crescendo gradualmente.

Outras informações

- Exemplos
- Python Enhancement Proposal 3101
- Biblioteca Python, String Formatting Operations
- Formatação antiga de *strings*
- Introdução à formatação de saída mais elegante, tutorial Python, seção 7.1

5.1.6 Mudanças de Python 2 para Python 3: `print`

Uma (talvez a mais óbvia) mudança da versão Python 2 para a 3 é perda do status especial do comando `print`. Em Python 2, poderíamos imprimir "Hello world"usando:

```
print "Hello world"              # válida em Python 2.x
```

Efetivamente, chamamos a função `print` com o argumento `Hello World`. Todas as outras funções em Python são chamadas com o argumento acompanhado por parênteses, i.e.

```
In [36]: print("Hello World")               # válida em Python 3.x

Hello World
```

Esta é a nova convenção *necessária* em Python 3 e *permitida* para a versão recente de Python 2.x.

Tudo o que aprendemos sobre a formatação de *strings* usando o operador de porcentagem ainda funciona da mesma maneira:

```
In [37]: import math
         a = math.pi
         "meu pi = %f" % a           # formatação de string

Out[37]: 'meu pi = 3.141593'

In [38]: print("meu pi = %f" % a)    # impressão valida em Python 2.7 e 3.x

meu pi = 3.141593


In [39]: "Curto pi = %.2f, mais longo pi = %.12f." % (a, a)

Out[39]: 'Curto pi = 3.14, mais longo pi = 3.141592653590.'

In [40]: print("Curto pi = %.2f, mais longo pi = %.12f." % (a, a))

Curto pi = 3.14, mais longo pi = 3.141592653590.


In [41]: print("Curto pi = %.2f, mais longo pi = %.12f." % (a, a))

Curto pi = 3.14, mais longo pi = 3.141592653590.
```

5.2 Leitura e escrita de arquivos

Aqui está um programa que

1. escreve algum texto para um arquivo com o nome `teste.txt`,
2. e depois lê o texto novamente e
3. imprime na tela.

Os dados armazenados no arquivo `teste.txt` são:

```
Escrevendo texto no arquivo. Esta é a primeira linha.
E a segunda linha.
```

```
In [42]: # 1. Escrever um arquivo
         out_file = open("teste.txt", "w")          # 'w' significa Writing (modo de escrita,
         out_file.write("Escrevendo texto no arquivo. Esta é a primeira linha.\n"+\
                        "E a segunda linha.")
         out_file.close()                           # fecha o arquivo

         # 2. Ler um arquivo
         in_file = open("teste.txt", "r")           # 'r' significa Reading (modo de leitura,
         text = in_file.read()                      # lê todo o arquivo em na string 'text'

         in_file.close()                            # fecha o arquivo

         # 3. Mostrar os dados
         print(text)
```

```
Escrevendo texto no arquivo. Esta é a primeira linha.
E a segunda linha.
```

Mais detalhadamente, abrimos um arquivo com o comando `open` e atribuimos este objeto de abertura de arquivo à variável `out_file`. Depois, escrevemos dados no arquivo usando o método `out_file.write`. Observe que, no exemplo acima, passamos uma *string* para o método `write`. Podemos, naturalmente, usar toda a formatação que discutimos antes. Por exemplo, para escrever este arquivo com o nome `tabela.txt`, podemos usar este programa Python. É uma boa prática fechar (`close()`) arquivos quando terminamos de operar em modo de leitura ou escrita. Se um programa Python for encerrado de forma controlada (ou seja, não por causa de um corte de energia ou de um *bug* improvável profundo na linguagem Python ou no sistema operacional), ele fechará todos os arquivos abertos assim que os objetos de manipulação de arquivo forem destruídos. No entanto, fechá-los ativamente o mais rápido possível é um estilo melhor.

5.2.1 Exemplos de leitura de arquivos

Usamos um arquivo chamado `meu-arquivo.txt` contendo as seguintes 3 linhas de texto para os exemplos abaixo:

Esta é a primeira linha. Esta é a segunda linha. Esta é uma terceira e última linha.

```
In [43]: f = open('meu-arquivo.txt', 'w')
         f.write('Esta e a primeira linha.\n'
                 'Esta e a segunda linha.\n'
                 'Esta e uma terceira e ultima linha.')
         f.close()
```

fileobject.read() O método `fileobject.read()` lê o arquivo inteiro e o retorna como uma *string* (incluindo caracteres de nova linha - `\n`).

```
In [44]: f = open('meu-arquivo.txt', 'r')
         f.read()
```

```
Out[44]: 'Esta e a primeira linha.\nEsta e a segunda linha.\nEsta e uma terceira e ultima li
```

```
In [45]: f.close()
```

fileobject.readlines() O método `fileobject.readlines()` retorna uma lista de *strings*, onde cada elemento da lista corresponde a uma linha na *string*:

```
In [46]: f = open('meu-arquivo.txt', 'r')
         f.readlines()

Out[46]: ['Esta e a primeira linha.\n',
          'Esta e a segunda linha.\n',
          'Esta e uma terceira e ultima linha.']

In [47]: f.close()
```

Isto é frequentemente usado para iterar nas linhas, e para fazer alguma coisa com cada linha. Por exemplo:

```
In [48]: f = open('meu-arquivo.txt', 'r')
         for line in f.readlines():
             print("%d caracteres" % len(line))
         f.close()

25 caracteres
24 caracteres
35 caracteres
```

Observe que isso irá ler o arquivo completo em uma lista de *strings* quando o método `readlines()` for chamado. Isso não é problema se sabemos que o arquivo é pequeno e se encaixa na memória da máquina.

Se assim for, também podemos fechar o arquivo antes de processarmos os dados, ou seja:

```
In [49]: f = open('meu-arquivo.txt', 'r')
         lines = f.readlines()
         f.close()
         for line in lines:
             print("%d caracteres" % len(line))

25 caracteres
24 caracteres
35 caracteres
```

Iterando em linhas (objeto arquivo) Existe uma possibilidade mais legal de ler uma linha de arquivo por linha que (i) somente lerá uma linha por vez (e também é adequada para arquivos grandes) e (ii) resulta em código mais compacto:

```
In [50]: f = open('meu-arquivo.txt', 'r')
         for line in f:
             print("%d caracteres" % len(line))
         f.close()

25 caracteres
24 caracteres
35 caracteres
```

Aqui, o manipulador de arquivo `f` atua como um iterador e retornará a próxima linha em cada iteração subsequente do laço `for` até o final do arquivo ser alcançado (e então o laço `for` é encerrado).

Leitura adicional Métodos para arquivos, Tutorial, Seção 7.2.1

6 Fluxo de Controle

6.1 Básico

Para um dado arquivo com um programa Python, o interpretador Python começará no topo e depois processará o arquivo. Demonstramos isso com um programa simples. Por exemplo:

```
In [1]: def f(x):
            """funcao que computa e retorna x*x"""
            return x * x

        print("O Programa principal comeca aqui")
        print("4 * 4 = %s" % f(4))
        print("Ultima linha do programa -- adeus")
```

```
O Programa principal comeca aqui
4 * 4 = 16
Ultima linha do programa -- adeus
```

A regra básica é que os comandos em um arquivo (ou função ou qualquer sequência de comandos) são processados de cima para baixo. Se vários comandos forem dados na mesma linha (separados por ;), eles serão processados da esquerda para a direita (embora seja desencorajado ter múltiplas declarações por linha a fim de manter boa legibilidade de código.)

Neste exemplo, o interpretador começa no topo (linha 1). Ele encontra a palavra-chave `def` e lembra para o futuro que a função `f` está definida aqui. (Ainda não executará o corpo da função, ou seja, a linha 3 - isso só acontece quando chamamos a função). O interpretador pode ver a partir do recuo onde o corpo da função acaba: o recuo na linha 5 é diferente do da primeira linha no corpo da função (linha 2), e assim o corpo da função terminou, e a execução deve continuar com essa linha. (Linhas vazias não são importantes para essa análise).

Na linha 5, o intérprete imprimirá a saída `O programa principal comeca aqui`. Então, a linha 6 é executada. Esta linha contém a expressão `f(4)`, a qual chamará a função `f(x)`, definida na linha 1, em que `x` tomará o valor `4`. [Atualmente `x` é uma referência ao objeto `4`.] A função `f` é então executada, retornando `4*4` na linha 3. Este valor `16` é usado na linha 6 para substituir `f(4)`. Por fim, a representação *string* `%s` do objeto 16 é impressa como parte do comando de impressão na linha 6. O interpretador então passa para a linha 7 antes de o programa terminar.

Agora vamos aprender sobre diferentes possibilidades para direcionar esse fluxo de controle ainda mais.

6.1.1 Condicionais

Os valores `True` e `False` são objetos predefinidos especiais:

```
In [2]: a = True
        print(a)
```

```
True
```

```
In [3]: type(a)
```

```
Out[3]: bool
```

```
In [4]: b = False
        print(b)
```

```
False
```

```
In [5]: type(b)
```

```
Out[5]: bool
```

Podemos operar com esses dois valores lógicos usando a lógica booleana, por exemplo, a lógica e a operação (and):

```
In [6]: True and True          # lógica e operação
```

```
Out[6]: True
```

```
In [7]: True and False
```

```
Out[7]: False
```

```
In [8]: False and True
```

```
Out[8]: False
```

```
In [9]: True and True
```

```
Out[9]: True
```

```
In [10]: c = a and b
         print(c)
```

```
False
```

Há também o *ou* lógico (or) e a negação (not</ span>):

```
In [11]: True or False
```

```
Out[11]: True
```

```
In [12]: not True
```

```
Out[12]: False
```

```
In [13]: not False
```

```
Out[13]: True
```

```
In [14]: True and not False

Out[14]: True
```

Em códigos computacionais, muitas vezes precisamos avaliar alguma expressão que seja verdadeira ou falsa (às vezes chamada de "predicado"). Por exemplo:

```
In [15]: x = 30          # atribui 30 a x
         x > 15          # x é maior do que 15?

Out[15]: True

In [16]: x > 42

Out[16]: False

In [17]: x == 30         # x é igual a 30?

Out[17]: True

In [18]: x == 42

Out[18]: False

In [19]: not x == 42     # x não é o mesmo que 42?

Out[19]: True

In [20]: x != 42         # x não é o mesmo que 42?

Out[20]: True

In [21]: x > 30          # x é maior do que 30?

Out[21]: False

In [22]: x >= 30         # x é maior do que ou igual a 30?

Out[22]: True
```

6.2 Se-então-senão

Informação adicional

- Introdução à estrutura condicional `if-then` em Python tutorial, seção 4.1

A declaração `if` permite execução condicional de código. Por exemplo:

```
In [23]: a = 34
         if a > 0:
             print("a é positivo")

a é positivo
```

A declaração `if` também pode ter uma ramificação `else` que é executada se a condição estiver errada:

```
In [24]: a = 34
         if a > 0:
             print("a é positivo")
         else:
             print("a é nao-positivo (i.e. negativo ou zero)")
```

```
a é positivo
```

Finalmente, existe a palavra-chave `elif` (leia como "else if") que permite verificar várias possibilidades (exclusivas):

```
In [25]: a = 17
         if a == 0:
             print("a é zero")
         elif a < 0:
             print("a é negativo")
         else:
             print("a é positivo")
```

```
a é positivo
```

6.3 Laço `for`

Informação adicional

- Introdução a laços `for` Python tutorial, seção 4.2

O laço `for` permite-nos iterar sobre uma sequencia (isto poderia ser uma *string* ou lista, por exemplo). Aqui está um exemplo:

```
In [26]: for animal in ['cão','gato','rato']:
             print(animal, animal.upper())
```

```
cão CÃO
gato GATO
rato RATO
```

Juntamente com o comando `range()`, pode-se iterar sobre inteiros crescentes:

```
In [27]: for i in range(5,10):
             print(i)
```

```
5
6
7
8
9
```

6.4 Laço `while`

A palavra-chave `while` permite-nos repetir uma operação enquanto uma condição é verdadeira. Suponha que quiséssemos saber por quantos anos teremos que guardar 100 reais em uma poupança para alcançar 200 reais simplesmente por uma taxa de juros de 5% ao ano. Aqui está um programa para computar isso:

```
In [28]: montante = 100         # em BRL
         taxa = 1.05            # 5% a.a. juros
         anos = 0
         while montante < 200:  # repita até que 200 seja alcancado
             montante = montante * taxa
             anos = anos + 1
         print('Preciso de', anos, 'anos para alcançar', montante, 'reais.')
```

```
Preciso de 15 anos para alcançar 207.89281794113688 reais.
```

Operadores relacionais (comparações) nas declarações `if` e `while` ---------- ----------

A forma geral de `if` e `while` é a mesma: seguindo a palavra-chave `if` ou `while`, existe uma *condição* seguida de dois pontos. Na próxima linha, um novo (e, portanto, indentado!) bloco de comandos começa a ser executado se a condição for `True`).

Por exemplo, a condição pode ser a igualdade de duas variáveis `a1` e `a2`, expressa como `a1 == a2`:

```
In [29]: a1 = 42
         a2 = 42
         if a1 == a2:
             print("a1 e a2 são iguais")
```

```
a1 e a2 são iguais
```

Outro exemplo é testar se `a1` e `a2` não são os mesmos. Para isso, temos duas possibilidades. A opção número 1 usa o *operador de desigualdade* `!=`:

```
In [30]: if a1 != a2:
             print("a1 e a2 são diferentes")
```

A opção 2 usa a palavra-chave `not` à frente da condição:

```
In [31]: if not a1 == a2:
             print("a1 e a2 são diferentes")
```

Comparações para "maior"(>), "menor"(<) e "maior ou igual a"(>=) e "menor ou igual a"(<=) são diretos.

Finalmente, podemos usar os operadores lógicos "`and`"e "`or`"para combinar condições:

```
In [32]: a = 11
         b = -3
         if a > 10 and b > 20:
             print("A é maior do que 10 e b é maior do que 20")
         if a > 10 or b < -5:
             print("Ou a é maior do que 10, ou "
                   "b é menor do que -5, ou ambos.")
```

```
Ou a é maior do que 10, ou b é menor do que -5, ou ambos.
```

Use o *prompt* do Python para experimentar estas comparações e expressões lógicas. Por exemplo:

```
In [33]: T = -12.5
         if T < -20:
             print("muito frio")

         if T < -10:
             print("bastante frio")

bastante frio


In [34]: T < -20

Out[34]: False

In [35]: T < -10

Out[35]: True
```

6.5 Exceções

Mesmo que uma declaração ou expressão seja sintaticamente correta, ela pode causar um erro quando uma tentativa é feita para executá-la. Os erros detectados durante a execução são chamados *exceções* e não são necessariamente fatais: exceções podem ser *capturadas* e tratadas no programa. A maioria das exceções não são tratadas pelos programas, no entanto, e resultam em mensagens de erro como as mostradas aqui:

```
In [36]: 10 * (1/0)

        ---------------------------------------------------------------------------

        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-36-9ce172bd90a7> in <module>()
    ----> 1 10 * (1/0)


        ZeroDivisionError: division by zero


In [37]: 4 + spam*3

        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)
```

```
    <ipython-input-37-6b1dfe582d2e> in <module>()
----> 1 4 + spam*3


    NameError: name 'spam' is not defined
```

```
In [38]: '2' + 2
```

```
    ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

    <ipython-input-38-4c6dd5170204> in <module>()
----> 1 '2' + 2


    TypeError: must be str, not int
```

Excepção esquemática capturando todas as opções

```
In [39]: try:
             # corpo de código
         except ArithmeticError:
             # o que fazer se houver erro de aritmética
         except IndexError, the_exception:
             # the_exception refere-se à exceção neste bloco
         except:
             # o que fazer para qualquer outra exceção
         else:  # opcional
             # o que fazer se nenhuma exceção for lançada

         try:
             # corpo de código
         finally:
             # o que fazer SEMPRE


          File "<ipython-input-39-1f62ec36e0ce>", line 3
        except ArithmeticError:
            ^
    IndentationError: expected an indented block
```

A partir do Python 2.5, você pode usar a instrução `with` para simplificar a escrita do código para algumas funções predefinidas, em particular a função `open` para abrir arquivos: veja http://docs.python.org/tutorial/errors.html#predefined-clean-up-actions.

Exemplo: tentaremos abrir um arquivo que não existe e o Python criará uma exceção do tipo `IOError`, que significa erro de entrada/saída:

```
In [40]: f = open("filenamethatdoesnotexist", "r")

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)

<ipython-input-40-11dd75e3491b> in <module>()
----> 1 f = open("filenamethatdoesnotexist", "r")


FileNotFoundError: [Errno 2] No such file or directory: 'filenamethatdoesnotexist'
```

Se estivéssemos escrevendo um aplicativo com uma interface onde o usuário deve digitar ou selecionar um nome de arquivo, não gostaríamos que o aplicativo parasse se o arquivo não existisse. Em vez disso, precisamos capturar esta exceção e agir para tratá-la (informando, por exemplo, o usuário que um arquivo com tal nome não existe e perguntar se ele quer tentar outro nome de arquivo). Aqui está o esqueleto para capturar esta exceção:

```
In [41]: try:
             f = open("arquivo_inexistente","r")
         except IOError:
             print("Arquivo não pode ser aberto.")

Arquivo não pode ser aberto.
```

Há muito mais para ser dito sobre exceções e o uso delas em programas maiores. Comece lendo o Capítulo 8 do Python Tutorial: Erros e Exceções se você tiver interesse.

6.5.1 Lançando Exceções

Possibilidades de lançar uma exceção:

- `raise OverflowError`
- `raise OverflowError, Bath is full` (estilo antigo, desencorajado)
- `raise OverflowError(Bath is full)`
- `e = OverflowError(Bath is full); raise e`

Hierarquia de exceções As exceções padrão são organizadas em uma hierarquia de herança, e.g. `OverflowError` é uma subclasse de `ArithmeticError`. Isso pode ser visto ao examinarmos `help('exceptions')`, por exemplo. Você pode derivar suas próprias exceções de qualquer um dos padrões. É um bom estilo que cada módulo defina sua própria exceção de base.

6.5.2 Criando as suas próprias exceções

- Você pode e deve derivar suas próprias exceções a partir de uma exceção predefinida.
- Para ver quais exceções predefenidas existem, procure no módulo de exceções (tente `help('exceptions')`) ou acesse <http://docs.python.org/library/ Exceptions.html # bltin-exceptions>.

6.5.3 LBYL vs EAFP

- LBYL (Look Before You Leap = Pense duas vezes antes de agir) *versus*
- EAFP (Easer to ask forgiveness than permission = Facilidade de pedir perdão do que permissão)

```
In [42]: numerador = 7
         denominador = 0
```

Exemplo para LBYL:

```
In [43]: if denominador == 0:
             print("Oops...")
         else:
             print(numerador/denominador)
```

```
Oops...
```

EAFP:

```
In [44]: try:
             print(numerador/denominador)
         except ZeroDivisionError:
             print("Oops...")
```

```
Oops...
```

O que a documentação do Python diz sobre EAFP:
Veja http://docs.python.org/glossary.html#term-eafp
O que a documentação do Python diz sobre LBYL:
Veja http://docs.python.org/glossary.html#term-lbyl
EAFP é a maneira "Pythônica".

7 Funções e módulos

7.1 Introdução

As funções nos permitem agrupar várias declarações em um bloco lógico. Comunicamo-nos com uma função através de uma interface claramente definida, fornecendo certos parâmetros para a função e recebendo algumas informações de volta. Além dessa interface, geralmente não sabemos como, exatamente, uma função faz o trabalho dela para obter o valor que retorna.

Por exemplo, a função `math.sqrt`: não sabemos, na totalidade, como ela calcula a raiz quadrada, mas sabemos sobre a interface: se passarmos `x` para a função, ela retornará (uma aproximação de) $\sqrt{x}$.

Esta abstração é uma coisa útil: é uma técnica comum na engenharia para quebrar um sistema em componentes menores (caixa preta) que trabalham juntos através de interfaces bem definidas, mas que não precisam saber sobre as realizações internas da funcionalidade uns dos outros. Na verdade, não ter que se preocupar com esses detalhes de implementação pode nos ajudar a ter uma visão mais clara do sistema composto por muitas dessas componentes.

As funções fornecem os blocos básicos de funcionalidade em programas maiores (e simulações computacionais) e ajudam a controlar a complexidade inerente ao processo.

Podemos agrupar funções em um módulo Python e, assim, criar nossas próprias bibliotecas de funcionalidades.

7.2 Usando funções

A palavra "função"tem diferentes significados em matemática e programação. Na programação, ela se refere a uma sequência de operações que executam uma computação. Por exemplo, a função `sqrt()`, definida no módulo de matemática `math`, calcula a raiz quadrada de um determinado valor:

```
In [1]: from math import sqrt
        sqrt(4)

Out[1]: 2.0
```

O valor que passamos para a função `sqrt` é 4 neste exemplo. Esse valor é chamado de *argumento* da função. Uma função pode ter mais de um argumento.

A função retorna o valor 2.0 (o resultado de sua computação) para o "contexto de chamada". Esse valor é chamado de *valor de retorno* da função.

É comum dizer que uma função "leva"um argumento e "retorna um resultado, o chamado *valor de retorno*.

Confusão comum sobre a impressão e os valores de retorno É um erro comum do principiante confundir a *impressão* de valores com *valores de retorno*. No exemplo a seguir, é difícil ver se a função `math.sin` retorna um valor ou se ela imprime o valor:

```
In [2]: import math
        math.sin(2)

Out[2]: 0.9092974268256817
```

Nós importamos o módulo `math` e chamamos a função `math.sin` com o argumento `2`. A chamada `math.sin(2)` realmente *retornará* o valor `0.909...`, não o imprimirá. No entanto, como não foi atribuído o valor de retorno a uma variável, o prompt do Python imprimirá o objeto retornado.

A seguinte sequência alternativa funciona apenas se o valor for retornado:

```
In [3]: x = math.sin(2)
        print(x)

0.9092974268256817
```

O valor de retorno da chamada da função `math.sin(2)` é atribuído à variável `x` e `x` é impresso na linha seguinte.

Geralmente, as funções devem executar "silenciosamente"(ou seja, não imprimir nada) e reportar o resultado de sua computação através do valor de retorno.

Parte da confusão sobre valores impressos versus valores de retorno no prompt do Python vem da impressão rápida do Python (uma representação) dos objetos retornados *se* os objetos retornados não são atribuídos. Geralmente, ver os objetos retornados é exatamente o que queremos (como normalmente nos preocupamos com o objeto retornado), apenas ao aprender Python. Isso pode causar confusão leve sobre funções que retornam valores ou valores de impressão.

Informação adicional

- Pense em Python apresenta uma introdução gentil às funções (em que se baseia o parágrafo anterior) nos Capítulo 3 e 6.

7.3 Definindo funções

Formato genérico de definição de uma função:

```
def minha_funcao(arg1, arg2, ..., argn):
    """Docstring opcional"""

    # Implementação da função

    return resultado  # opcional

# isto nao é parte da função
algum_comando
```

A terminologia de Allen Downey (em seu livro Pense em Python) de funções frutíferas e infrutíferas distingue funções que retornam um valor daquelas que não retornam valor algum. A distinção refere-se à característica de uma função fornecer um valor de retorno (=frutífera) ou não retornar explicitamente um valor (=infrutífera). Se as funções não usam a instrução `return`, tendemos a dizer que a função não retorna nada (enquanto que, na realidade, sempre retornará o objeto `None` quando ele termina - mesmo que nela falte a instrução `return`).

Por exemplo, a função `cumprimento` imprimirá "Olá, mundo!"quando chamada (e é infrutífero, pois não retorna um valor).

```
In [4]: def cumprimento():
            print("Olá, mundo!")
```

Quando chamamos essa função:

```
In [6]: cumprimento()

Olá, mundo!
```

ela imprime “Olá, Mundo!” para `stdout`, como esperado. Se atribuírmos o valor de retorno da função para a variável `x`, poderemos inspecioná-lo:

```
In [7]: x = cumprimento()

Olá, mundo!


In [8]: print(x)

None
```

e encontrarmos que a função `cumprimento`, de fato, retornou, o objeto `None`.

Outro exemplo para uma função que não retorna valor algum (isso significa que não há palavra-chave `return` na função) seria:

```
In [9]: def printpluses(n):
            print(n * "+")
```

Geralmente, funções que retornam valores são mais úteis, pois podem ser usadas para montar código (talvez como outra função) combinando-as de maneira "esperta". Vejamos alguns exemplos de funções que retornam um valor.

Suponhamos que precisemos definir uma função que calcule o quadrado de uma determinada variável. O código-fonte da função poderia ser:

```
In [10]: def quadrado(x):
             return x * x
```

A palavra-chave `def` diz ao Python que estamos *definindo* uma função naquele ponto. A função leva um argumento, a saber `x`. A função retorna `x*x`, que é, claramente, x^2. Aqui está um exemplo que mostra como a função pode ser definida e usada:

```
In [11]: def quadrado(x):
             return x * x

         for i in range(5):
             i_quadrado = quadrado(i)
             print(i, '*', i, '=', i_quadrado)
```

```
0 * 0 = 0
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
```

Vale mencionar que as linhas 1 e 2 definem a função `quadrado`, ao passo que as de linhas 4 a 6 são o programa principal.

Podemos definir funções que levam mais do que um argumento:

```
In [12]: import math

         def hipotenusa(x, y):
             return math.sqrt(x * x + y * y)
```

Também é possível retornar mais de um argumento. Aqui está um exemplo de uma função que converte uma determinada *string* retornando-a em duas versões: com todos os caracteres em letras maiúsculas e todos os caracteres em letras minúsculas. Incluímos o programa principal para mostrar como esta função pode ser chamada:

```
In [15]: def maiusculasEMinusculas(string):
             return string.upper(), string.lower()

         palavra = 'Banana'

         uppercase, lowercase = maiusculasEMinusculas(palavra)

         print(palavra, 'em minúsculas:', lowercase,
               'e em maiúsculas', uppercase)
```

```
Banana em minúsculas: banana e em maiúsculas BANANA
```

Podemos definir múltiplas funções em Python em um arquivo. Aqui está um exemplo com duas funções:

```
In [16]: def retorna_estrelas( n ):
             return n * '*'

         def imprime_centrado_estrelas( string ):
             linelength = 46
             starstring = retorna_estrelas((linelength - len(string)) // 2)

             print(starstring + string + starstring)

         imprime_centrado_estrelas('Olá, Mundo!')
```

```
**********Olá, Mundo!**********
```

Leitura complementar

- Python Tutorial: Seção 4.6 Definindo Funções

7.4 Módulos

- Agrupam funcionalidade;
- Fornecem espaço de nomes;
- A biblioteca padrão do Python contém uma vasta coleção de módulos (tente `help('modules')`);
- Extensão da linguagem Python.

7.4.1 Importando módulos

```
In [1]: import math
```

Isto introduzirá o nome `math` no espaço de nomes no qual o comando de importação foi lançado. Os nomes dentro do módulo `math` não aparecerão diretamente: eles devem ser acessados através do nome `math`. Por exemplo: `math.sin`.

```
In [17]: import math, cmath
```

Mais de um módulo pode ser importado na mesma declaração, embora o Python Style Guide recomende não fazer isso. Em vez disso, devemos escrever

```
In [18]: import math
         import cmath

         import math as matematica
```

O nome pelo qual o módulo é conhecido localmente pode ser diferente do seu nome "oficial". Os usos típicos deste são:

- evitar conflitos de nomes com nomes existentes, e
- mudar o nome para algo mais gerenciável. Por exemplo, `import SimpleHTTPServer as shs`. Isto é desencorajado para código de produção (visto que nomes longos e mais significativos tornam os programas muito mais compreensíveis do que os curtos e crípticos), mas para testar experiências de forma interativa, ser capaz de usar um sinônimo curto pode tornar sua vida muito mais fácil. Dado que os módulos (importados) são objetos de primeira classe, você pode, claro, simplesmente fazer `shs = SimpleHTTPServer` a fim de obter o identificador mais facilmente identificável no módulo.

```
In [19]: from math import sin
```

Esta declaração importará a função `sin` do módulo `math`, mas não irá introduzir o nome `math` no espaço de nomes corrente, senãp apenas o nome `sin`. É possível extrair mais de um nome do módulo de uma só vez:

```
In [20]: from math import sin, cos
```

Finalmente, vejamos esta notação:

```
In [7]: from math import *
```

Mais uma vez, isso não introduz o nome `math` no espaço de nomes corrente, mas *todos os nomes públicos* do módulo `math`. Em termos gerais, é uma má idéia fazer isso:

- Muitos nomes novos serão despejados no espaço de nomes corrente.
- Você tem certeza de que eles não sobrescreverão nenhum nome já presente?
- Será muito difícil rastrear de onde esses nomes vieram
- Dito isto, alguns módulos (incluindo os da biblioteca padrão, recomendam que sejam importados dessa maneira). Use com cuidado!
- Isso é bom para testes rápidos interativos ou pequenos cálculos.

7.4.2 Criando módulos

Um módulo é, em princípio, nada mais do que um arquivo Python. Aqui está um exemplo de um arquivo de módulo que é salvo em `modulo1.py`:

```
def alguma_funcao_util():
    pass

print("Meu nome é", __name__)
```

Podemos executar este arquivo (módulo) como um programa em Python normal (por exemplo: `python modulo1.py`):

```
In [21]: cd static/data
```

```
/Users/gustavo/courses/calculo-numerico/lecture-ipynb/static/data
```

```
In [22]: !python modulo1.py

('Meu nome eh', '__main__')
```

Observamos que a variável mágica `__name__` leva o valor `__main__` se o arquivo de programa `modulo1.py` for executado.

Por outro lado, podemos *importar* `modulo1.py` em outro arquivo (que poderia ter o nome `prog.py`), por exemplo, como este:

```
In [23]: import modulo1            # em um certo arquivo prog.py

Meu nome eh modulo1
```

Quando o Python depara-se com a instrução `import modulo1` em `prog.py`, ele procura o arquivo `modulo1.py` no diretório de trabalho atual (e se ele não conseguir encontrá-lo em todos os diretórios em `sys.path`) e abre o arquivo `modulo1.py`. Ao fazer o *parsing* do arquivo `modulo1.py` de cima para baixo, ele adicionará quaisquer definições de função neste arquivo no espaço de nome `modulo1` no contexto de chamada (esse é o programa principal em `prog.py`). Neste exemplo, existe apenas a função `alguma_funcao_util`. Uma vez que o processo de importação esteja completo, podemos fazer uso de `modulo1.algum_funcao_util` em `prog.py`. Se o Python encontrar instruções diferentes das definições de função (e classe) durante a importação de `modulo1.py`, ele executará isso imediatamente. Neste caso, ele se deparará com a declaração `print(Meu nome eh, __name __)`.

Observe a diferença na saída se importarmos `modulo1.py` em vez de executá-lo por conta própria: `__name__` dentro de um módulo leva o valor do nome do módulo se o arquivo for importado.

7.4.3 Uso de ___name___

Em suma,

- `__name__` é `__main__` se o arquivo do módulo for executado sozinho;
- `__name__` é o nome do módulo (ou seja, o nome do arquivo do módulo sem o sufixo `.py`) se o arquivo do módulo for importado.

Podemos, portanto, usar a seguinte instrução `if` em `modulo1.py` para escrever o código que é *apenas executado* quando o módulo é executado por conta própria: isto é útil para manter programas de teste ou demonstrações das habilidades de um módulo neste programa principal "condicional". É prática comum para qualquer arquivo de módulo ter um programa principal condicional que demonstre suas capacidades.

7.4.4 Exemplo 1

O próximo exemplo mostra um programa principal para outro arquivo (`vetorial.py`) que é usado para demonstrar as habilidades das funções definidas naquele arquivo:

```
from __future__ import division
import math
import numpy as N

def norma(x):
```

```
    """retorna a magnitude de um vetor x"""
    return math.sqrt(sum(x ** 2))


def vetor_unitario(x):
    """retorna um vetor unitario x/|x|. x requer um tipo numpy array."""
    xnorm = norma(x)
    if xnorm == 0:
        raise ValueError("Vetor nulo nao pode ser normalizado")
    return x / norma(x)


if __name__ == "__main__":
    # uma pequena demonstracao de como as funcoes neste modulo podem ser usadas:
    x1 = N.array([0, 1, 2])
    print("A norma de " + str(x1) + " eh " + str(norma(x1)) + ".")
    print("O vetor unitario na direcao de " + str(x1) + " eh " \
        + str(vetor_unitario(x1)) + ".")
```

Se este arquivo for executado usando `python vetorial.py`, entao `__name__==__main__` é verdadeiro, e a saída será

```
In [14]: !python vetorial.py

A norma de [0 1 2] eh 2.2360679775.
O vetor unitario na direcao de [0 1 2] eh [ 0.          0.4472136   0.89442719].
```

Se este arquivo for importado (ou seja, usado como um módulo) em outro arquivo Python, então `__name __ == __ main__` é falso, e esse bloco de declaração não será executado (e nenhuma saída será produzida).

Esta é uma forma bastante comum de executar o código condicionalmente em arquivos que fornecem funções semelhantes às da biblioteca. O código que é executado se o arquivo for executado por conta própria geralmente consiste em uma série de testes (para verificar se as funções do arquivo realizam as operações certas - *testes de regressão* ou *testes de unidade*), ou alguns exemplos de como as funções da biblioteca no arquivo podem ser usadas.

7.4.5 Exemplo 2

Mesmo que um programa Python não se destine a ser usado como um arquivo de módulo, é uma boa prática usar sempre um programa principal condicional:

- muitas vezes, verifica-se, mais tarde, que as funções no arquivo podem ser reutilizadas (assim, economiza-se trabalho)
- isso é conveniente para testes de regressão.

Suponha que um determinado exercício seja escrever uma função que retorne os primeiros 5 números primos e além disso, os imprima. (Há, claro, uma solução trivial para isso, como sabemos, os números primos, e devemos imaginar que o cálculo necessário é mais complexo). Podemos ser tentados a escrever:

```
In [19]: def primos5():
             return (2, 3, 5, 7, 11)

         for p in primos5():
             print("%d" % p),
```

```
2 3 5 7 11
```

É um estilo melhor usar uma função principal condicional, i.e.:

```
In [20]: def primos5():
             return (2, 3, 5, 7, 11)

         if __name__=="__main__":
             for p in primos5():
                 print("%d" % p),
```

```
2 3 5 7 11
```

Um purista pode argumentar que o seguinte ainda é mais "limpo":

```
In [21]: def primos5():
             return (2, 3, 5, 7, 11)

         def main():
             for p in primos5():
                 print("%d" % p),

         if __name__=="__main__":
             main()
```

```
2 3 5 7 11
```

mas qualquer uma das duas opções é boa.

Leitura complementar

- Python Tutorial Seção 6

8 Ferramentas funcionais

A linguagem Python fornece alguns comandos predefinidos, tais como `map`, `filter`, `reduce`, bem como `lambda` (para criar funções anônimas) e compreensões de lista. Tais comandos são típicos de linguagens funcionais, das quais LISP provavelmente é a mais conhecida.

A programação funcional pode ser extremamente poderosa e um dos pontos fortes da linguagem Python é que ela permite programar usando (i) o estilo de programação procedural (ou imperativa), (ii) o estilo orientado a objetos e (iii) o estilo funcional. São os programadores quem escolhem quais ferramentas selecionar, qual estilo e como misturá-los para resolver melhor um determinado problema.

Neste capítulo, fornecemos alguns exemplos para o uso dos comandos listados acima.

8.1 Funções anônimas

Todas as funções que vimos em Python até agora foram definidas através da palavra-chave `def`. Por exemplo:

```
In [1]: def f(x):
            return x ** 2
```

Esta função tem o nome `f`. Uma vez que a função está definida (ou seja, o interpretador Python encontrou a linha `def`), podemos chamar a função usando seu nome. Por exemplo:

```
In [2]: y = f(6)
```

Às vezes, precisamos definir uma função que só é usada uma vez, ou queremos criar uma função, mas não precisamos de um nome para ela (como para criar fechamentos). Neste caso, isso é chamado de *função anônima*, pois não possui um nome. Em Python, a palavra-chave `lambda` pode criar uma função anônima.

Criamos uma função (nomeada) primeiro, verificamos seu tipo e comportamento:

```
In [3]: def f(x):
            return x ** 2

        f

Out[3]: <function __main__.f>

In [4]: type(f)

Out[4]: function

In [5]: f(10)

Out[5]: 100
```

Agora fazemos o mesmo com uma função anônima:

```
In [6]: lambda x: x ** 2

Out[6]: <function __main__.<lambda>>

In [7]: type(lambda x: x ** 2)

Out[7]: function

In [8]: (lambda x: x ** 2)(10)

Out[8]: 100
```

Isto funciona exatamente da mesma maneira, mas - como a função anônima não tem um nome - precisamos definir a função (através da expressão `lambda`) - toda vez que precisamos dela.

Funções anônimas podem levar mais de um argumento:

```
In [9]: (lambda x, y: x + y)(10, 20)

Out[9]: 30

In [10]: (lambda x, y, z: (x + y) * z )(10, 20, 2)

Out[10]: 60
```

Veremos alguns exemplos usando `lambda` que esclarecerão os casos típicos de uso.

8.2 `map`

A função `lst2 = map(f, s)` aplica uma função `f` a todos os elementos em uma seqüência `s`. O resultado de `map` pode ser convertido em uma lista com o mesmo comprimento que `s`:

```
In [11]: def f(x):
             return x ** 2
         lst2 = list(map(f, range(10)))
         lst2

Out[11]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [12]: list(map(str.capitalize, ['banana', 'maçã', 'laranja']))

Out[12]: ['Banana', 'Maçã', 'Laranja']
```

Frequentemente, isto é combinado com a função anônima`lambda`:

```
In [13]: list(map(lambda x: x ** 2, range(10) ))

Out[13]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [14]: list(map(lambda s: s.capitalize(), ['banana', 'maçã', 'laranja']))

Out[14]: ['Banana', 'Maçã', 'Laranja']
```

8.3 `filter`

A função `lst2 = filter(f, lst)` aplica a função `f` a todos os elementos em uma sequencia `s`. A função `f` deve retornar `True` ou `False`. Isto faz com que uma lista contenha somente aqueles elementos *si* da sequencia `s` para os quais `f`(*si*) tem valor de retorno `True`.

```
In [15]: def maior_que_5(x):
             if x > 5:
                     return True
             else:
                     return False

         list(filter(maior_que_5, range(11)))

Out[15]: [6, 7, 8, 9, 10]
```

O uso de `lambda` pode simplificar isto significativamente:

```
In [16]: list(filter(lambda x: x > 5, range(11)))

Out[16]: [6, 7, 8, 9, 10]

In [17]: nomes_conhecidos = ['smith', 'miller', 'bob']
         list(filter( lambda nome : nome in nomes_conhecidos, \
              ['ago', 'smith', 'bob', 'carl']))

Out[17]: ['smith', 'bob']
```

8.4 Compreensões de lista

Compreensões de lista fornecem uma maneira concisa de criar e modificar listas sem recorrer ao uso de `map()`, `filter()` e/ou `lambda`. A definição da lista resultante tende a ser mais clara do que as listas criadas usando essas construções. Cada compreensão de lista consiste de uma expressão seguida por uma cláusula `for`, zero ou mais cláusulas `for` ou `if`. O resultado será uma lista resultante da avaliação da expressão no contexto das cláusulas `for` e `if` que a seguem. Se a expressão for avaliada para uma tupla, ela deverá estar entre parênteses.

Alguns exemplos tornarão isso mais claro:

```
In [18]: frutas_frescas = ['  banana', '  framboesa ', 'maracujá  ']
         [weapon.strip() for weapon in frutas_frescas]

Out[18]: ['banana', 'framboesa', 'maracujá']

In [19]: vec = [2, 4, 6]
         [3 * x for x in vec]

Out[19]: [6, 12, 18]

In [20]: [3 * x for x in vec if x > 3]

Out[20]: [12, 18]

In [21]: [3 * x for x in vec if x < 2]

Out[21]: []

In [22]: [[x, x ** 2] for x in vec]

Out[22]: [[2, 4], [4, 16], [6, 36]]
```

Podemos também usar compreensões de lista para modificar a lista de inteiros retornada pelo comando `range`, de modo que nossos elementos subsequentes na lista aumentem por frações não-inteiras.

```
In [23]: [x*0.5 for x in range(10)]

Out[23]: [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

Vamos agora revisitar os exemplos da seção `filter`.

```
In [24]: [x for x in range(11) if x>5 ]

Out[24]: [6, 7, 8, 9, 10]

In [25]: [nome for nome in ['ago','smith','bob','carl'] \
               if nome in nomes_conhecidos]

Out[25]: ['smith', 'bob']
```

e os exemplos da seção `map`

```
In [26]: [x ** 2 for x in range(10) ]
```

```
Out[26]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [27]: [fruta.capitalize() for fruta in ['banana', 'maçã', 'laranja'] ]

Out[27]: ['Banana', 'Maçã', 'Laranja']
```

todas podem ser expressas através de compreensões de lista.
Mais detalhes

- Tutorial Python 5.1.4 Compreensões de lista

8.5 `reduce`

A função `reduce` assume uma função binária `f(x, y)`, uma sequência `s` e um valor de início `a0`. Em seguida, aplica a função `f` ao valor de inícioa0 e o primeiro elemento na seqüência: `a1 = f(a, s[0])`. O segundo elemento (`s [1]`) da sequência é processado da seguinte forma: a função `f` é chamada com argumentos `a1` e `s[1]`, ou seja, `a2 = f (a1, s[1])`. Desta forma, toda a sequência é processada. `reduce` retorna um único número.

Isso pode ser usado, por exemplo, para calcular uma soma de números em uma seqüência se a função `f(x, y)` retornar `x + y`:

```
In [29]: from functools import reduce

In [30]: def add(x,y):
             return x+y

         reduce(add, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 0)

Out[30]: 55

In [31]: reduce(add, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 100)

Out[31]: 155
```

Podemos modificar a função `add` para fornecer algum detalhe a mais sobre o processo:

```
In [32]: def add_verbose(x, y):
             print("add(x=%s, y=%s) -> %s" % (x, y, x+y))
             return x+y

         reduce(add_verbose, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 0)

add(x=0, y=1) -> 1
add(x=1, y=2) -> 3
add(x=3, y=3) -> 6
add(x=6, y=4) -> 10
add(x=10, y=5) -> 15
add(x=15, y=6) -> 21
add(x=21, y=7) -> 28
add(x=28, y=8) -> 36
add(x=36, y=9) -> 45
add(x=45, y=10) -> 55
```

```
Out[32]: 55
```

Pode ser instrutivo usar uma função assimétrica `f`, tal como `add_len(n,s)`, onde `s` é uma sequencia e a função retorna `n+len(s)` (sugestão de Thomas Fischbacher):

```
In [34]: def add_len(n,s):
             return n+len(s)

         reduce(add_len, ["Este","é","um","teste."],0)
```

```
Out[34]: 13
```

Como antes, usaremos uma versão mais prolixa (verbosa) da função binária para vermos o que está acontecendo:

```
In [35]: def add_len_verbose(n,s):
             print("add_len(n=%d, s=%s) -> %d" % (n, s, n+len(s)))
             return n+len(s)

         reduce(add_len_verbose, ["Este","é","um","teste."],0)
```

```
add_len(n=0, s=Este) -> 4
add_len(n=4, s=é) -> 5
add_len(n=5, s=um) -> 7
add_len(n=7, s=teste.) -> 13


Out[35]: 13
```

Outra maneira de entender o que a função `reduce` faz é examinar a seguinte função (gentilmente fornecida por Thomas Fischbacher), que se comporta como `reduce`, mas explica o que ela faz:

Aqui está um exemplo usando a função `explain_reduce`.

```
In [34]: cd code/
```

```
/Users/fangohr/hg/teaching-python/book/text/code


In [35]: from explain_reduce import explain_reduce
         def f(a,b):
             return a+b

         reduce(f, [1,2,3,4,5], 0)
```

```
Out[35]: 15
```

```
In [36]: explain_reduce(f, [1,2,3,4,5], 0)
```

```
Step 0: value-so-far=0 next-list-element=1
Step 1: value-so-far=1 next-list-element=2
Step 2: value-so-far=3 next-list-element=3
Step 3: value-so-far=6 next-list-element=4
Step 4: value-so-far=10 next-list-element=5
Done. Final result=15
```

```
Out[36]: 15
```

`reduce` é frequentemente combinado com `lambda`:

```
In [37]: reduce(lambda x,y:x+y, [1,2,3,4,5], 0)

Out[37]: 15
```

Existe também o módulo `operator` que fornece operadores Python padrão como funções. Por exemplo, a função `operator.__add__(a, b)` é executada quando o Python avalia o código como `a + b`. Estes geralmente são mais rápidos do que as expressões `lambda`. Poderíamos escrever o exemplo acima como

```
In [22]: import operator
         reduce(operator.__add__, [1,2,3,4,5], 0)

Out[22]: 15
```

Use `help(operator)` para ver a lista completa de funções `operator`.

8.6 Por que não usar apenas laços `for`?

Vamos comparar o exemplo apresentado no início do capítulo escrito (i) usando um laço `for` e (ii) compreensão de lista. Mais uma vez, queremos calcular os números 02, 12, 22 , 32,... até (n - 1)2 para um dado n.

Implementação (i) usando um laço `for` com n = 10:

```
In [23]: y = []
         for i in range(10):
             y.append(i ** 2)
```

Implementação (ii) usando compreensão de lista:

```
In [24]: y = [x ** 2 for x in range(10)]
```

ou usando `map`:

```
In [25]: y = map(lambda x: x ** 2, range(10))
```

As versões que utilizam compreensão de lista e `map` encaixam-se em uma linha de código, enquanto um laço `for` precisa de três linhas. Este exemplo mostra que o código funcional resulta em expressões *concisas*. Normalmente, o número de erros que um programador comete é por linha de código escrita, de modo que quanto menos linhas de código tivermos, menos *bugs* teremos de encontrar.

Muitas vezes, os programadores descobrem que, inicialmente, as ferramentas de processamento de lista apresentadas neste capítulo parecem menos intuitivas do que o uso de laços `for` para processar cada elemento em uma lista individualmente, mas isso - ao longo do tempo - tende a ser valorizado por eles como um estilo de programação mais funcional.

8.7 Rapidez

As ferramentas funcionais descritas neste capítulo também podem ser mais rápidas do que laços explícitos (`for` ou `while`) sobre os elementos da lista.

O programa `rapidez_compreensao_lista.py` abaixo calcula $\sum_{i=0}^{N-1} i^2$ para um grande valor de N usando 4 métodos diferentes e registra o tempo de execução:

- Método 1: laço `for` (com lista pré-alocada, armazenamento de $i2$ na lista, em seguida, usando a função predefinida `sum`)
- Método 2: laço `for` sem lista (atualizando `sum` à medida que o laço `for` progride)
- Método 3: usando compreensão de lista
- Método 4: usando `numpy`.

O programa produz a seguinte saída:

```
In [36]: !python static/data/rapidez_compreensao_lista.py
```

```
('N =', 10000000)
('for-loop1', 5.317609071731567)
('for-loop2', 4.719789981842041)
('listcomp', 3.4225449562072754)
('numpy', 0.04654884338378906)
O metodo mais lento eh 114.2 vezes mais lento do que o metodo mais rapido.
```

O desempenho de execução real depende do computador. O desempenho relativo pode depender de versões do Python e suas bibliotecas de suporte (como `numpy`) que usamos.

Com a versão atual (python 3.4, numpy 1.10, em uma máquina x64 que executa o OSX), vemos que os métodos 1 e 2 (para laço sem lista e com lista pré-alocada) são mais lentos, seguidos um tanto mais próximos de uma compreensão de lista levemente mais rápida. O método mais rápido é o número 4 (usando numpy).

Para referência, aqui está o código-fonte do programa:

```
In [37]: !cat static/data/rapidez_compreensao_lista.py
```

```
# -*- coding: utf-8 -*-
"""Compara cálculos de \sum_i x_i^2 for i = 0,...N-1

Usamos (i) laços for e list, (ii) laço for, (iii) compreensão de lista
e (iv) numpy.

Usamos números em ponto flutuante para evitar usar os longos inteiros do
Python (que provavelmente fariam a medição do tempo menos representativa)
"""

import time
import numpy
N = 10000000
```

```
def timeit(f, args):
    """ Dada uma função f e uma tupla contendo argumentos para f,
    esta funcao chama f(*args), e mede e retorna o tempo de
    execucao em segundos.

    O valor de retorno é uma tupla: a entrada 0 é o tempo,
    a entrada 1 é o valor de retorno de f."""

    starttime = time.time()
    y = f(*args)    # usa tupla de argumentos como entrada
    endtime = time.time()
    return endtime - starttime, y


def forloop1(N):
    s = 0
    for i in range(N):
        s += float(i) * float(i)
    return s


def forloop2(N):
    y = [0] * N
    for i in range(N):
        y[i] = float(i) ** 2
    return sum(y)


def listcomp(N):
    return sum([float(x) * x for x in range(N)])


def numpy_(N):
    return numpy.sum(numpy.arange(0, N, dtype='d') ** 2)

# inicio do programa principal
timings = []
print("N =", N)
forloop1_time, f1_res = timeit(forloop1, (N,))
timings.append(forloop1_time)
print("for-loop1", forloop1_time)
forloop2_time, f2_res = timeit(forloop2, (N,))
timings.append(forloop2_time)
print("for-loop2", forloop2_time)
listcomp_time, lc_res = timeit(listcomp, (N,))
timings.append(listcomp_time)
print("listcomp", listcomp_time)
numpy_time, n_res = timeit(numpy_, (N,))
timings.append(numpy_time)
print("numpy", numpy_time)
```

```
#assegura que metodos diferentes forneçam resultados idênticos
assert f1_res == f2_res
assert f1_res == lc_res

# Permite um pouco de diferença para o cálculo via numpy
numpy.testing.assert_approx_equal(f1_res, n_res)

print("O metodo mais lento eh {:.1f} vezes mais lento do que o metodo mais rapido.".format(
    max(timings)/min(timings)))
```

9 Tarefas comuns

Aqui fornecemos uma seleção de pequenos exemplos de programas que abordam algumas tarefas e um pouco mais de códigos em Python aos quais podemos recorrer enquanto buscamos inspiração para atacar um determinado problema.

9.1 Muitas maneiras de computar uma série

As an example, we compute the sum of odd numbers in different ways.

Como um exemplo, computamos a soma de números ímpares de diferentes maneiras.

```
In [30]: def calcula_soma1(n):
             """computa e retorna a soma de 2,4,6,...,m, onde m
             é o maior número par menor do que n.

             Por exemplo, com n = 7, computamos 0+2+4+6 = 12.

             Esta implementação usa a variável 'soma' que é
             incrementada a cada iteração do laço `for`.
             """
             soma = 0
             for i in range(0, n, 2):
                 soma = soma + i
             return soma


         def calcula_soma2(n):
             """Idem...

             Esta implementação usa um loop `while`:
             """

             contador = 0
             soma = 0
             while contador < n:
                 soma = soma + contador
                 contador = contador + 2

             return soma
```

```
def calcula_soma3(n, inicio=0):
    """Idem... Esta é uma implementação recursiva:"""

    if n <= inicio:
        return 0
    else:
        return inicio + calcula_soma3(n, inicio + 2)


def calcula_soma4a(n):
    """Uma abordagem funcional... Este parece ser
    o código mais curto e conciso.
    """
    return sum(range(0, n, 2))

from functools import reduce
def calcula_soma4b(n):
    """Uma abordagem funcional... não faz uso de
    'sum' mas conveniente aqui.
    """
    return reduce(lambda a, b: a + b, range(0, n, 2))


def calcula_soma4c(n):
    """Uma abordagem funcional... um pouco mais rápida do
    que `calcula_soma4b` uma vez que evitamos o uso de `lambda`
    """
    import operator
    return reduce(operator.__add__, range(0, n, 2))


def calcula_soma4d(n):
    """Usando compreensão de lista."""
    return sum([k for k in range(0, n, 2)])


def calcula_soma4e(n):
    """Usando outra variação de compreensão de lista."""
    return sum([k for k in range(0, n) if k % 2 == 0])


def calcula_soma5(n):
    """Usando python numérico (numpy). Esta é muito rápida
    (mas não tem o mesmo efeito para n muito maior do que 10)."""
    import numpy
    return numpy.sum(2 * numpy.arange(0, (n + 1) // 2))
```

```
        def teste_consistencia():
            """Verifica todas as funções `calcula_soma`. As funções neste arquivo
            produzem a mesma resposta para todo n >= 2 and < N.
            """
            def verificacao_1_n(n):
                """Compara a saida de `compute_sum1` com todas as outras funções
                para um dado n>=2. Lança exceção do tipo `AssertionError` se houver discor
                funcs = [calcula_soma1, calcula_soma2, calcula_soma3,
                         calcula_soma4a, calcula_soma4b, calcula_soma4c,
                         calcula_soma4d, calcula_soma4e, calcula_soma5]
                resp1 = calcula_soma1(n)
                for f in funcs[1:]:
                    assert resp1 == f(n), "%s(n)=%d diferente de %s(n)=%d " \
                                          % (funcs[0], funcs[0](n), f, f(n))

            # laço principal de teste na funcao teste_consistencia
            for n in range(2, 1000):
                verificacao_1_n(n)

        if __name__ == "__main__":
            m = 7
            resultado_correto = 12
            resultado_atual = calcula_soma1(m)
            print("este resultado eh {}, esperado {}".format(
                resultado_atual, resultado_correto))
            # compara com resultado correto
            assert resultado_atual == resultado_correto
            # verifica todos os outros metodos
            assert calcula_soma2(m) == resultado_correto
            assert calcula_soma3(m) == resultado_correto
            assert calcula_soma4a(m) == resultado_correto
            assert calcula_soma4b(m) == resultado_correto
            assert calcula_soma4c(m) == resultado_correto
            assert calcula_soma4d(m) == resultado_correto
            assert calcula_soma4e(m) == resultado_correto
            assert calcula_soma5(m) == resultado_correto

            # verificação mais sistemática para muitos valores
            teste_consistencia()
```

```
este resultado eh 12, esperado 12
```

Todas as implementações diferentes mostradas acima calculam o mesmo resultado. Há algumas coisas a serem aprendidas com isso:

- Há um número grande (provavelmente infinito) de soluções para um determinado problema. (Isso significa que escrever programas é uma tarefa que exige criatividade!)
- Estes podem atingir o mesmo "resultado"(neste caso, o cálculo de um número).
- Diferentes soluções podem ter características diferentes. Elas podem:

- ser mais rápidas ou mais lentas
- usar menos ou mais memória
- ser mais fáceis ou mais difíceis de entender (quando se lê o código-fonte)
- podem ser consideradas mais ou menos elegantes.

9.2 Classificação (*Sorting*)

Suponha que precisamos classificar uma lista de tuplas com pares de identificadores de usuário e nomes, i.e.

```
In [31]: minha_lista = [("tarso", "Paulo de Tarso",),
                        ("admin", "O Administrador"),
                        ("visitante", "O Visitante")]
```

a qual queremos classificar em ordem crescente de identificadores de usuário. Se houverem dois ou mais identificadores de usuários idênticos, eles devem ser classificados pela ordem dos nomes associados com esses identificadores. Este comportamento é apenas o comportamento padrão de `sort` (que volta ao assunto de como sequencias são comparadas).

```
In [32]: coisas = minha_lista # colete seus dados
         coisas.sort()        # classifique
         print(coisas)        # inspecione os dados classificados

[('admin', 'O Administrador'), ('tarso', 'Paulo de Tarso'), ('visitante', 'O Visitante')]
```

As sequências são inicialmente comparadas pelos primeiros elementos apenas. Se eles diferirem, então uma decisão é tomada com base apenas nesses elementos. Se os elementos forem iguais, somente então os próximos elementos na seqüência são comparados... e assim por diante, até que uma diferença for encontrada, ou ficarmos sem elementos. Por exemplo:

```
In [33]: (2,0) > (1,0)

Out[33]: True

In [34]: (2,1) > (1,3)

Out[34]: True

In [35]: (2,1) > (2,1)

Out[35]: False

In [36]: (2,2) > (2,1)

Out[36]: True
```

É também possível fazer:

```
In [37]: coisas = sorted(coisas)
```

Quando a lista não é particularmente grande, geralmente é aconselhável usar a função `sorted` (que *retorna uma cópia ordenada* da lista) sobre o método `sort` de uma lista (que altera a lista para uma classificação ordenada de elementos, e retorna `None`).

No entanto, e se os dados que temos forem armazenados de tal forma que, em cada uma das tuplas da lista, o nome venha primeiro, seguido do identificador, isto é,

```
In [38]: minha_lista2 = [("Paulo de Tarso", "tarso"),
                         ("O Administrador", "admin"),
                         ("O Visitante", "visitante")]
```

We want to sort with the id as the primary key. The first approach to do this is to change the order of `mylist2` to that of `mylist`, and use `sort` as shown above.

The second, neater approach relies on being able to decypher the cryptic help for the sorted function. `list.sort()` has the same options, but its help is less helpful.

Queremos ordenar tomando o identificador como chave primária. A primeira abordagem para fazer isto é mudar a ordem de `minha_lista2` para aquela de `minha_lista`, e usar `sort` como mostrado acima.

```
In [39]: help(sorted)

Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

Você deve notar que `sorted` e `list.sort` têm dois parâmetros. O primeiro deles é chamado `key` (palavra-chave, ou simplesmente chave). Você pode usar isso para fornecer uma *função-chave*, que será usada para transformar os itens para ser comparados via `sort`.

Vamos ilustrar isso no contexto do nosso exercício, assumindo que nós armazenamos uma lista de pares como este

```
par = nome, id
```

(i.e. como em `minha_lista_2`) e que queiramos classificar de acordo com `id`, ignorando `nome`. Podemos atingir isto escrevendo uma função que recupere apenas o segundo elemento do par que ela recebe:

```
In [40]: def minha_chave(par):
             return par[1]

In [41]: minha_lista2.sort(key=minha_chave)
```

Isto também funciona com uma função anônima:

```
In [42]: minha_lista2.sort(key=lambda p: p[1])
```

9.2.1 Eficiência

A função `key` será chamada exatamente uma vez para cada elemento da lista. Isso é muito mais eficiente do que chamar uma função em cada *comparação* (que era como você personalizava a classificação em versões antigas do Python). Mas se você tiver uma grande lista para ordenar, a sobrecarga de chamada de uma função Python (que é relativamente grande em comparação com a sobrecarga de uma função em C) pode ser notável.

Se a eficiência for realmente importante (e você provou que uma proporção significativa de tempo é gasta nessas funções), você tem a opção de recodificá-las em C (ou em outra linguagem de baixo nível).

10 Do Matlab para Python

10.1 Comandos importantes

10.1.1 O laço `for`

Matlab:

```
for i = 1:10
    disp(i)
end
```

No Matlab, a palavra-chave `end` é necessária para finalizar o bloco do laço `for`.

No Python:

```
In [1]: for i in range(1,11):
            print(i)

1
2
3
4
5
6
7
8
9
10
```

Em Python, é necessário adicionar dois-pontos (":") no final da linha do `for`. (Isto é importante e frequentemente esquecido, principalmente, para quem já programou em Matlab antes.) Além disso, os comandos a serem executados dentro do laço `for` devem ser indentados.

10.1.2 A declaração condicional `if-then`

Matlab:

```
if a==0
    disp('a é zero')
elseif a<0
```

```
    disp('a é negativo')
elseif a==42
    disp('a é 42')
else
    disp('a é positivo')
end
```

Em Matlab, a palavra-chave `end` é necessária na parte final do bloco.
Python:

```
In [2]: a = -5

        if a==0:
            print('a é zero')
        elif a<0:
            print('a é negativo')
        elif a==42:
            print('a é 42')
        else:
            print('a é positivo')
```

```
a é negativo
```

Em Python, é necessário adicionar dois-pontos (":") após cada condição (i.e., nas linhas começando com `if`, `elif`, `else`). Além disso, os comandos a serem executados dentro de cada parte do escopo `if-then-else` devem ser indentados.

10.1.3 Indexação

No Matlab, a indexação de matrizes e vetores começa em 1 (similarmente ao Fortran), ao passo que no Python, ela começa em 0 (similarmente ao C).

10.1.4 Matrizes

Em Matlab, todo objeto é uma matriz. Em Python, há uma biblioteca especializada chamada NumPy que fornece o objeto `array`, o qual, por sua vez, possui todas as funcionalidades correspondentes. Similarmente ao Matlab, o NumPy é baseado em bibliotecas cuja execução é bastante rápida.

Há um documento introdutório dedicado ao NumPy para usuários do Matlab disponível em https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html.

11 *Shells* para Python

11.1 IDLE

O **IDLE** vem com toda distribuição Python e é uma ferramenta útil para a programação diária. Seu editor fornece destaque de sintaxe.

Existem duas razões pelas quais você poderia optar por outro *shell* para Python, por exemplo:

- Ao trabalhar com o *prompt* do Python, se você desejar preenchimento automático de nomes de variáveis, nomes de arquivos e comandos. Nesse caso, *IPython* é a sua ferramenta de escolha (veja abaixo). O IPython não fornece um editor, mas você pode continuar usando o editor IDLE ou qualquer outro que você quiser para editar seus arquivos.

O IPython fornece uma série de recursos agradáveis para o programador Python mais experiente, incluindo um perfil conveniente de código (veja http://ipython.scipy.org).

Recentemente, algumas características interessantes foram adicionadas ao IDLE também (e.g. pressionar a tecla `tab` depois de ter digitado as primeiras letras de nomes de objetos e palavras-chave para preenchimento automático).

11.2 Python (linha de comando)

Esta é a face mais básica de um *shell* para Python. É muito semelhante ao *prompt* do Python no IDLE, mas não há menus para clicar e nenhuma instalação para editar arquivos.

11.3 Python Interativo (IPython)

11.3.1 Console IPython

O IPython é uma versão melhorada da linha de comando do Python. É uma ferramenta valiosa e vale a pena explorar suas capacidades (veja http://ipython.org/ipython-doc/stable/interactive/qtconsole.html)

Você encontrará os seguintes recursos muito úteis:

- Preenchimento automático: Suponha que você digite `a = range(10)`. Em vez de digitar todas as letras, basta digitar `a = ra` e pressionar a tecla `tab`. O IPython mostrará todos os comandos possíveis (e nomes de variáveis) que começam com `ra`. Se você digitar uma terceira letra, aqui `n` e pressionar `Tab` novamente, o IPython completará e adicionará `ge` automaticamente. Isso também funciona para nomes de variáveis e de módulos.

- Para obter ajuda sobre um comando, podemos usar o comando de ajuda do Python. Por exemplo: `help(range)`. O IPython fornece um atalho. Para alcançar o mesmo, basta digitar o comando seguido de um ponto de interrogação: `range?`

- Você pode navegar facilmente por diretórios no seu computador. Por exemplo,

  - `!dir` lista o conteúdo do diretório atual (mesmo que `ls`)
  - `pwd` mostra o diretório de trabalho atual
  - `cd` permite alterar diretórios

- Em geral, ao usar um ponto de exclamação antes do comando, você passa o comando para o `shell` (e não para o interpretador do Python).

- Você pode executar programas Python a partir do IPython usando `%run`. Suponha que você tenha um arquivo `hello.py` no diretório atual. Você pode então executá-lo digitando: `%run hello`. Observe que isto difere da execução de um programa Python no IDLE: o IDLE reinicia a sessão do interpretador do Python e, assim, deleta todos os objetos existentes antes de a execução começar. Este não é o caso com o comando `run` no IPython (nem ao executar partes de código Python no Emacs usando o modo Python). Em particular, isso pode ser muito útil se for necessário configurar alguns objetos para testar o código em que se está trabalhando. Usar o comando `run` do IPython, ou o Emacs em vez do IDLE permite-nos manter esses objetos na sessão do interpretador e atualizar apenas a função, classe, etc. que está sendo desenvolvida.

- permite a edição de várias linhas do histórico de comandos

- fornece destaque de sintaxe *on-the-fly*

- exibe *docstrings on-the-fly*

- pode incluir figuras via Matplotlib (ativar modo `inline` com o comando `%matplotlib inline`)
- `% load` carrega o arquivo do disco ou de alguma URL para edição
- `% timeit` mede o tempo de execução para uma determinada declaração
- ...e muito mais.
- Leia mais em http://ipython.org/ipython-doc/dev/interactive/qtconsole.html

Se você tiver acesso a esse *shell*, você talvez o considerará como seu *prompt* padrão do Python.

11.3.2 Jupyter Notebook

O Jupyter Notebook (anteriormente IPython Notebook) permite que você execute, armazene, carregue e reexecute uma sequencia de comandos Python, além de incluir texto explicativo, imagens e outras mídias. É um desenvolvimento recente e excitante que tem o potencial de se desdobrar em uma ferramenta de grande significado, por exemplo, para

- documentar cálculos e processamento de dados
- dar suporte ao ensino e aprendizagem
  - ▷ do Python em si
  - ▷ de métodos estatísticos
  - ▷ do pós-processamento geral de dados
  - ▷ ...
- documentação de novos códigos
- testes de regressão automática pela reexecucação de *IPython notebooks* para fins de comparação entre dados de saída e dados armazenados.

Leitura complementar

- Jupyter Notebook (http://jupyter-notebook.readthedocs.io/en/latest/).
- IPython (http://ipython.org).

11.4 Spyder

Spyder é o *Scientific PYthon Development EnviRonment*: um ambiente de desenvolvimento interativo poderoso para a linguagem Python com recursos avançados de edição, teste interativo, depuração, introspecção e computação numérica, graças ao suporte do IPython e bibliotecas populares, tais como NumPy (linear algebra), SciPy (signal and image processing) ou Matplotlib (plotagem interativa 2D/3D). Veja mais em http://pythonhosted.org/spyder/.

Algumas características do Spyder:

- Dentro do Spyder, o console IPython é o interpretador Python padrão, e
- o código no editor pode ser executado total ou parcialmente neste buffer.
- O editor suporta a verificação automática de erros usando `pyflakes` e

- o editor avisa (se desejado) se a formatação do código se desviar do guia de estilo PEP8.
- O Depurador Ipython pode ser ativado e
- um analisador de desempenho (*profiler*) é fornecido.
- Um explorador de objetos mostra documentação para funções, métodos, etc, *on-the-fly* e um
- explorador de variáveis exibe nomes, tamanho e valores para variáveis numéricas.

O Spyder está atualmente (a partir de 2014) no caminho para se tornar um ambiente integrado multi-plataforma poderoso e robusto para desenvolvimento em Python, com especial ênfase em computação científica e engenharia.

11.5 Editores

Todos os principais editores que são usados para programação fornecem suporte a Python (e.g. Emacs, Vim, Sublime Text), alguns *Integrated Development Enviroments* (IDEs) vêm com seu próprio editor (Spyder, Eclipse). Qual é o melhor é, em parte, uma questão de escolha.

Para iniciantes, o Spyder parece uma escolha sensata, pois fornece um IDE, permite a execução de porções de código em uma sessão de interpretação e é fácil de aprender.

12 Computação simbólica

12.1 SymPy

Nesta seção, apresentamos algumas funcionalidades básicas da biblioteca SymPy (SYMbolic Python). Em contraste com a computação numérica (envolvendo números), no cálculo simbólico estamos processando e transformando variáveis genéricas.

A página inicial do SymPy (http://sympy.org/) fornece a documentação completa (e atualizada) para esta biblioteca.

A computação simbólica é muito lento em comparação com as operações em ponto flutuante e, assim, geralmente não muito adequada para simulação direta. No entanto, é uma ferramenta poderosa no suporte à preparação do código e trabalho simbólico. Ocasionalmente, usamos operações simbólicas em simulações para elaborar o código numérico mais eficiente, antes que seja executado.

12.1.1 Saída

Antes de começarmos a usar o SymPy, chamaremos `init_printing`. Isto diz ao SymPy para mostrar as expressões em um formato mais conveniente.

```
In [1]: import sympy
        sympy.init_printing()
```

12.1.2 Símbolos

Antes de começarmos a executar qualquer operação simbólica, precisamos criar variáveis simbólicas usando a função `Symbol` do SymPy:

```
In [2]: from sympy import Symbol
        x = Symbol('x')
        type(x)
```

```
Out[2]: sympy.core.symbol.Symbol

In [3]: y = Symbol('y')
        2 * x - x

   Out[3]:
```

$$x$$

```
In [4]: x + y + x + 10*y

   Out[4]:
```

$$2x + 11y$$

```
In [5]: y + x - y + 10

   Out[5]:
```

$$x + 10$$

Podemos abreviar a criação de múltiplas variáveis simbólicas usando a função `symbols`. Por exemplo, para criar as variáveis simbólicas `x`, `y` e `z`, podemos usar

```
In [6]: import sympy
        x, y, z = sympy.symbols('x,y,z')
        x + 2*y + 3*z - x

   Out[6]:
```

$$2y + 3z$$

Uma vez que terminarmos a manipulação dos termos, às vezes desejamos inserir números para as variáveis. Isso pode ser feito usando o método `subs`.

```
In [7]: from sympy import symbols
        x, y = symbols('x,y')
        x + 2*y

   Out[7]:
```

$$x + 2y$$

```
In [8]: x + 2*y.subs(x, 10)

   Out[8]:
```

$$x + 2y$$

```
In [9]: (x + 2*y).subs(x, 10).subs(y, 3)

   Out[9]:
```

```
In [10]: (x + 2*y).subs({x:10, y:3})
```

Out[10]:

$$16$$

Também podemos substituir uma variável simbólica por outra, como neste exemplo, em que `y` é substituído por `x` antes de substituirmos `x` pelo número 2.

```
In [11]: termo = 3*x + y**2
         termo
```

Out[11]:

$$3x + y^2$$

```
In [12]: termo.subs(x, y)
```

Out[12]:

$$y^2 + 3y$$

```
In [13]: termo.subs(x, y).subs(y, 2)
```

Out[13]:

$$10$$

A partir deste ponto, alguns fragmentos de código e exemplos que apresentamos assumirão que os símbolos necessários já foram definidos. Se você tentar um exemplo e o SymPy der uma mensagem como `NameError: name 'x' is not defined`, é porque você precisa definir o símbolo usando um dos métodos acima.

12.1.3 isympy

O executável `isympy` é um *wrapper* em torno do ipython que cria as variáveis simbólicas (reais) `x`, `y` e `z`, as variáveis inteiras simbólicas `k`, `m` e `n`, e as variáveis de função simbólica `f`, `g` e `h`, e importa todos os objetos do SymPy.

Isto é conveniente para descobrir novos recursos ou fazer experimentos de forma interativa
$> isympy Python 2.6.5 console for SymPy 0.6.7

```
These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z = symbols('xyz')
>>> k, m, n = symbols('kmn', integer=True)
>>> f, g, h = map(Function, 'fgh')

Documentation can be found at http://sympy.org/

In [1]:
```

12.1.4 Tipos numéricos

O SymPy tem os tipos numéricos `Rational` e `RealNumber`. A classe `Rational` representa um número racional como um par de dois inteiros: o numerador e o denominador, então `Rational(1,2)` representa `1/2`, `Rational(5,2)` representa `5/2`, e assim por diante.

```
In [14]: from sympy import Rational

In [15]: a = Rational(1, 10)
         a

   Out[15]:
```

$$\frac{1}{10}$$

```
In [16]: b = Rational(45, 67)
         b

   Out[16]:
```

$$\frac{45}{67}$$

```
In [17]: a * b

   Out[17]:
```

$$\frac{9}{134}$$

```
In [18]: a - b

   Out[18]:
```

$$-\frac{383}{670}$$

```
In [19]: a + b

   Out[19]:
```

$$\frac{517}{670}$$

Note que a classe `Rational` funciona com expressões racionais *exatas*. Isto contrasta com o tipo de dado `float`, padrão do Python, que usa a representação em ponto flutuante para *aproximar* números racionais.

Podemos converter o tipo `sympy.Rational` em uma variável de ponto flutuante no Python usando `float` ou o método `evalf` do objeto `Rational`. O método `evalf` pode levar um argumento que especifica quantos dígitos devem ser calculados para a aproximação de ponto flutuante (nem todos podem ser usados pelo tipo de ponto flutuante do Python, evidentemente).

```
In [20]: c = Rational(2, 3)
         c
```

Out[20]:

$$\frac{2}{3}$$

In [21]: `float(c)`

Out[21]:

$$0.6666666666666666$$

In [22]: `c.evalf()`

Out[22]:

$$0.666666666666667$$

In [23]: `c.evalf(50)`

Out[23]:

$$0.66666666666666666666666666666666666666666666666667$$

12.1.5 Diferenciação e Integração

O SymPy é capaz de executar a diferenciação e integração de muitas funções:

In [24]:
```
from sympy import Symbol, exp, sin, sqrt, diff
x = Symbol('x')
y = Symbol('y')
diff(sin(x), x)
```

Out[24]:

$$\cos(x)$$

In [25]: `diff(sin(x), y)`

Out[25]:

$$0$$

In [26]: `diff(10 + 3*x + 4*y + 10*x**2 + x**9, x)`

Out[26]:

$$9x^8 + 20x + 3$$

In [27]: `diff(10 + 3*x + 4*y + 10*x**2 + x**9, y)`

Out[27]:

$$4$$

```
In [28]: diff(10 + 3*x + 4*y + 10*x**2 + x**9, x).subs(x,1)
```

Out[28]:

$$32$$

```
In [29]: diff(10 + 3*x + 4*y + 10*x**2 + x**9, x).subs(x,1.5)
```

Out[29]:

$$263.66015625$$

```
In [30]: diff(exp(x), x)
```

Out[30]:

$$e^x$$

```
In [31]: diff(exp(-x ** 2 / 2), x)
```

Out[31]:

$$-xe^{-\frac{x^2}{2}}$$

A função SymPy `diff()` usa no mínimo dois argumentos: a função a ser diferenciada e a variável em relação à qual a diferenciação é realizada. Derivadas de ordem superior podem ser calculadas pela especificação de variáveis adicionais, ou pela adição de um argumento inteiro opcional:

```
In [32]: diff(3*x**4, x)
```

Out[32]:

$$12x^3$$

```
In [33]: diff(3*x**4, x, x, x)
```

Out[33]:

$$72x$$

```
In [34]: diff(3*x**4, x, 3)
```

Out[34]:

$$72x$$

```
In [35]: diff(3*x**4*y**7, x, 2, y, 2)
```

Out[35]:

$$1512x^2y^5$$

```
In [36]: diff(diff(3*x**4*y**7, x, x), y, y)
```

$$98$$

Out[36]:

$$1512x^2y^5$$

Às vezes, o SymPy pode retornar um resultado de uma forma pouco familiar. Se, por exemplo, você desejar usar o SymPy para verificar se você diferenciou algo corretamente, uma técnica que pode ser útil é subtrair o resultado do SymPy do seu resultado e verificar se a resposta é zero.

Tomando o exemplo simples de uma função de base radial multiquádrica, $\phi(r) = \sqrt{r^2 + \sigma^2}$ com $r = \sqrt{x^2 + y^2}$ e σ uma constante, podemos verificar se a primeira derivada em x é $\frac{\partial \phi}{\partial x} = \frac{x}{\sqrt{r^2+\sigma^2}}$.

Neste exemplo, primeiro pedimos que o SymPy imprima a derivada. Veja que ela é impressa de forma diferente da nossa derivada de teste, mas a subtração verifica que elas são idênticas:

```
In [37]: r = sqrt(x**2 + y**2)
         sigma = Symbol('')
         def phi(x,y,sigma):
             return sqrt(x**2 + y**2 + sigma**2)

         minha_dfdx= x/sqrt(r**2 + sigma**2)
         print(diff(phi(x, y, sigma), x))
```

```
x/sqrt(x**2 + y**2 + **2)
```

```
In [38]: print(minha_dfdx - diff(phi(x, y, sigma), x))
```

```
0
```

Aqui é trivial dizer que as expressões são idênticas sem a ajuda do SymPy, mas, em exemplos mais complicados, pode haver muitos outros termos, tornando-se cada vez mais difícil, demorado e propenso a erros tentar reorganizar nossa derivada de teste e a resposta do SymPy na mesma forma. São nesses casos que esta técnica de subtração é de maior utilidade. A integração usa uma sintaxe semelhante. Para o caso indefinido, especifique a função e a variável em relação à qual a integração é realizada:

```
In [39]: from sympy import integrate
         integrate(x**2, x)
```

Out[39]:

$$\frac{x^3}{3}$$

```
In [40]: integrate(x**2, y)
```

Out[40]:

$$x^2y$$

```
In [41]: integrate(sin(x), y)
```

Out[41]:

$$y \sin(x)$$

```
In [42]: integrate(sin(x), x)
```

Out[42]:

$$-\cos(x)$$

```
In [43]: integrate(-x*exp(-x**2/2), x)
```

Out[43]:

$$e^{-\frac{x^2}{2}}$$

Podemos calcular integrais definidas fornecendo `integrate()` com uma tupla contendo a variável de interesse, os limites inferior e superior. Se várias variáveis forem especificadas, a integração múltipla é realizada. Quando o SymPy retorna um resultado na classe `Rational`, é possível avaliá-lo em uma representação de ponto flutuante com qualquer precisão desejada.

```
In [44]: integrate(x*2, (x, 0, 1))
```

Out[44]:

$$1$$

```
In [45]: integrate(x**2, x)
```

Out[45]:

$$\frac{x^3}{3}$$

```
In [46]: integrate(x**2, x, x)
```

Out[46]:

$$\frac{x^4}{12}$$

```
In [47]: integrate(x**2, x, x, y)
```

Out[47]:

$$\frac{x^4 y}{12}$$

```
In [48]: integrate(x**2, (x, 0, 2))
```

Out[48]:

$$\frac{8}{3}$$

```
In [49]: integrate(x**2, (x, 0, 2), (x, 0, 2), (y, 0, 1))
```

Out[49]:

$$\frac{16}{3}$$

```
In [50]: float(integrate(x**2, (x, 0, 2)))
```

Out[50]:

$$2.6666666666666665$$

```
In [51]: type(integrate(x**2, (x, 0, 2)))

Out[51]: sympy.core.numbers.Rational

In [52]: resultado_racional=integrate(x**2, (x, 0, 2))
         resultado_racional.evalf()
```

Out[52]:

$$2.66666666666667$$

```
In [53]: resultado_racional.evalf(50)
```

Out[53]:

$$2.6666666666666666666666666666666666666666666666667$$

12.1.6 Equações diferenciais ordinárias

O SymPy tem suporte incorporado para resolver vários tipos de equações diferenciais ordinárias através do comando `dsolve`. Precisamos configurar a EDO e passá-la como o primeiro argumento, `eq`. O segundo argumento é a função `f(x)` a resolver. Um terceiro argumento opcional, `hint`, influencia o método que `dsolve` usa: alguns métodos são mais adequados a certas classes de EDOs ou expressarão a solução de forma mais simples do que outros.

Para configurar o solucionador de EDO, precisamos de alguma maneira, fazer referencia à função incógnita que estamos resolvendo, bem como às suas derivadas. As classes `Function` e `Derivative` facilitam isso:

```
In [54]: from sympy import Symbol, dsolve, Function, Derivative, Eq
         y = Function("y")
         x = Symbol('x')
         y_ = Derivative(y(x), x)
         dsolve(y_ + 5*y(x), y(x))
```

Out[54]:

$$y(x) = C_1 e^{-5x}$$

Observe como `dsolve` introduziu uma constante de integração, `C1`. Esta função introduzirá tantas constantes quantas forem necessárias, e todas serão chamadas `Cn`, onde `n` é um número inteiro. Observe também que o primeiro argumento para `dsolve` é considerado igual a zero, a menos que usemos a função `Eq()` para especificar o contrário:

```
In [55]: dsolve(y_ + 5*y(x), y(x))
```

Out[55]:

$$y(x) = C_1 e^{-5x}$$

```
In [56]: dsolve(Eq(y_ + 5*y(x), 0), y(x))
```

Out[56]:

$$y(x) = C_1 e^{-5x}$$

```
In [57]: dsolve(Eq(y_ + 5*y(x), 12), y(x))
```

Out[57]:

$$y(x) = \frac{C_1}{5} e^{-5x} + \frac{12}{5}$$

Os resultados de `dsolve` são uma instância da classe `Equality`. Isto tem consequências quando desejamos avaliar numericamente a função e usar o resultado em outro lugar (e.g. se quisermos plotar $y(x)$ por x), porque mesmo depois de usar `subs()` e `evalf()`, ainda temos uma `Equality`, e não um escalar. A maneira de avaliar a função para um número é através do atributo `rhs` da `Equality`. Note que, aqui, usamos `z` para armazenar a `Equality` retornada por `dsolve`, ainda que seja uma expressão para uma função chamada `y(x)`, enfatizando a distinção entre `Equality` e os dados que ela contém.

```
In [58]: z = dsolve(y_ + 5*y(x), y(x))
         z
```

Out[58]:

$$y(x) = C_1 e^{-5x}$$

```
In [59]: type(z)

Out[59]: sympy.core.relational.Equality

In [60]: z.rhs
```

Out[60]:

$$C_1 e^{-5x}$$

```
In [61]: C1=Symbol('C1')
         y3 = z.subs({C1:2, x:3})
         y3
```

Out[61]:

$$y(3) = \frac{2}{e^{15}}$$

```
In [62]: y3.evalf(10)
```

Out[62]:

$$y(3) = 6.11804641 \cdot 10^{-7}$$

```
In [63]: y3.rhs
```

Out[63]:

$$\frac{2}{e^{15}}$$

In [64]:
```
y3.evalf(10)
```

Out[64]:

$$y(3) = 6.11804641 \cdot 10^{-7}$$

In [65]:
```
y3.rhs
```

Out[65]:

$$\frac{2}{e^{15}}$$

In [66]:
```
y3.rhs.evalf(10)
```

Out[66]:

$$6.11804641 \cdot 10^{-7}$$

In [67]:
```
z.rhs.subs({C1:2, x:4}).evalf(10)
```

Out[67]:

$$4.122307245 \cdot 10^{-9}$$

In [68]:
```
z.rhs.subs({C1:2, x:5}).evalf(10)
```

Out[68]:

$$2.777588773 \cdot 10^{-11}$$

In [69]:
```
type(z.rhs.subs({C1:2, x:5}).evalf(10))
```

Out[69]:
```
sympy.core.numbers.Float
```

Às vezes, `dsolve` pode retornar uma solução muito geral. Um exemplo é quando existe a possibilidade de que alguns coeficientes sejam complexos. Se soubermos que, por exemplo, eles são sempre reais e positivos, podemos passar esta informação para `dsolve` e evitar que a solução se torne desnecessariamente complicada:

In [70]:
```
from sympy import *
a, x = symbols('a,x')
f = Function('f')
dsolve(Derivative(f(x), x, 2) + a**4*f(x), f(x))
```

Out[70]:

$$f(x) = C_1 e^{-ia^2 x} + C_2 e^{ia^2 x}$$

In [71]:
```
a=Symbol('a',real=True,positive=True)
dsolve(Derivative(f(x), x, 2)+a**4*f(x), f(x))
```

Out[71]:

$$f(x) = C_1 \sin\left(a^2 x\right) + C_2 \cos\left(a^2 x\right)$$

12.1.7 Expansões em série e plotagens

É possível expandir muitas expressões do SymPy em série de Taylor. O método `series` torna isso direto. No mínimo, devemos especificar a expressão e a variável a ser expandida. Opcionalmente, também podemos especificar o ponto em torno do qual expandir, o número máximo de termos e a direção da expansão (tente `help(Basic.series)` para mais informações).

```
In [72]: from sympy import *
         x = Symbol('x')
         sin(x).series(x, 0)
```

Out[72]:

$$x - \frac{x^3}{6} + \frac{x^5}{120} + \mathcal{O}\left(x^6\right)$$

```
In [73]: series(sin(x), x, 0)
```

Out[73]:

$$x - \frac{x^3}{6} + \frac{x^5}{120} + \mathcal{O}\left(x^6\right)$$

```
In [74]: cos(x).series(x, 0.5, 10)
```

Out[74]:

$$1.11729533119247 - 0.438791280945186\left(x - 0.5\right)^2 + 0.0799042564340338\left(x - 0.5\right)^3 + 0.0365659400787655\left(x -$$

Em alguns casos, especialmente para avaliação numérica e plotagem dos resultados, é necessário remover o termo final `O (n)`:

```
In [75]: cos(x).series(x, 0.5, 10).removeO()
```

Out[75]:

$$-0.479425538604203x - 1.32116826114474 \cdot 10^{-6}\left(x - 0.5\right)^9 + 2.17654405230747 \cdot 10^{-5}\left(x - 0.5\right)^8 + 9.512411480$$

O SymPy fornece duas funções de plotagem, `plot()`, do módulo `sympy.plotting`, e `plot` do módulo `sympy.mpmath.visualization`. No momento da escrita, essas funções careciam de algumas capacidades de plotagem, o que significa que não são adequadas para a maioria de nossas necessidades. Ainda assim, caso você deseje utilizá-las, a função `help()` delas é útil.

Para a maioria dos nossos propósitos, Matplotlib deve ser a ferramenta de plotagem a ser escolhida. Aqui, fornecemos apenas um exemplo de como plotar os resultados de uma computação SymPy.

```
In [76]: %matplotlib inline
```

```
In [77]: from sympy import sin,series,Symbol
         import pylab
         x = Symbol('x')
         s10 = sin(x).series(x,0,10).removeO()
```

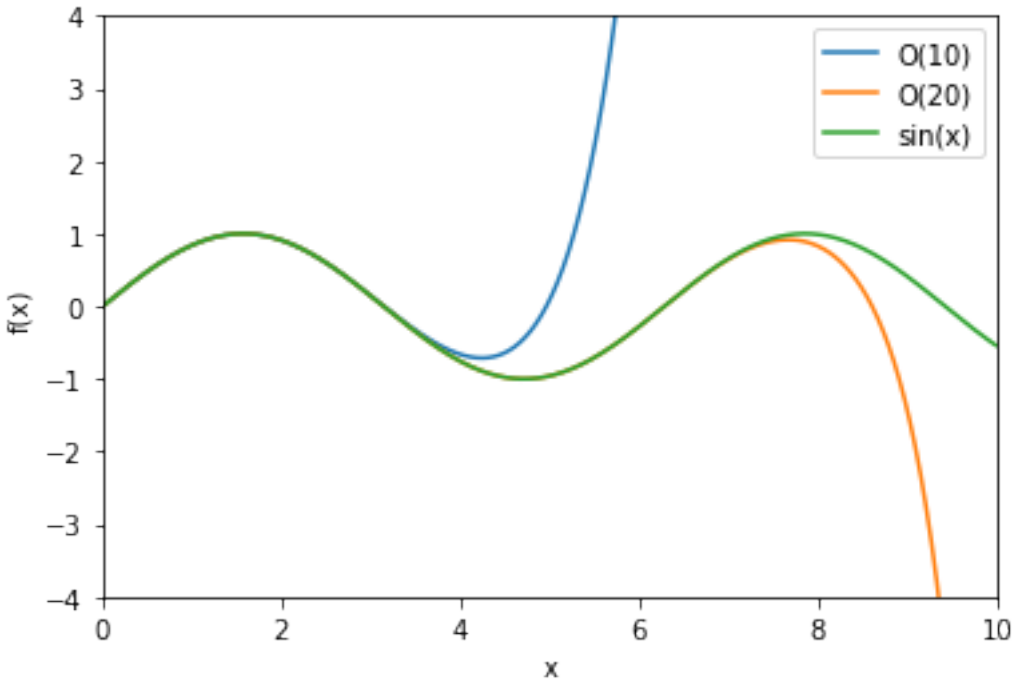
```
s20 = sin(x).series(x,0,20).removeO()
s = sin(x)
xx = []
y10 = []
y20 = []
y = []
for i in range(1000):
  xx.append(i / 100.0)
  y10.append(float(s10.subs({x:i/100.0})))
  y20.append(float(s20.subs({x:i/100.0})))
  y.append(float(s.subs({x:i/100.0})))

pylab.figure()
```

```
Out[77]: <matplotlib.figure.Figure at 0x1140d6c50>
```

```
<matplotlib.figure.Figure at 0x1140d6c50>
```

```
In [78]: pylab.plot(xx, y10, label='O(10)')
         pylab.plot(xx, y20, label='O(20)')
         pylab.plot(xx, y, label='sin(x)')

         pylab.axis([0, 10, -4, 4])
         pylab.xlabel('x')
         pylab.ylabel('f(x)')

         pylab.legend()
```

```
Out[78]: <matplotlib.legend.Legend at 0x1143f4a58>
```



12.1.8 Equações lineares e inversão de matrizes

O SymPy possui uma classe `Matrix` e funções associadas que permitem a solução simbólica de sistemas de equações lineares (e, claro, podemos obter respostas numéricas com `subs()` e `evalf()`). Considerararemos o exemplo do seguinte par de equações lineares simples:

$$3x + 7y = 12z$$
$$4x - 2y = 5z$$

Podemos escrever este sistema na forma $\mathbf{Ax} = \mathbf{b}$ (multiplique $\mathbf{A}$ por $\mathbf{x}$ caso queira verificar se as equações originais são recuperadas), onde

$$\mathbf{A} = \begin{pmatrix} 3 & 7 \\ 4 & -2 \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 12z \\ 5z \end{pmatrix}.$$

Aqui, incluímos um símbolo, z, no lado direito para demonstrar que os símbolos serão propagados na solução. Em muitos casos, teríamos $z = 1$, mas ainda pode haver benefício ao se usar SymPy como um solucionador numérico, mesmo quando a solução não contenha símbolos, devido à sua capacidade de retornar frações exatas em vez de `floats` aproximados.

Uma estratégia para resolver o sistema para $\mathbf{x}$ é inverter a matriz $\mathbf{A}$ e pré-multiplicar a equação pela matriz inversa, i.e. $\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. A classe `Matrix` do SymPy possui um método `inv()` que nos permite encontrar a inversa e `*` realiza a multiplicação da matriz para nós, quando apropriado:

```
In [79]: from sympy import symbols,Matrix
         x, y, z = symbols('x,y,z')
         A = Matrix(([3, 7], [4, -2]))
         A
```

Out[79]:

$$\begin{bmatrix} 3 & 7 \\ 4 & -2 \end{bmatrix}$$

```
In [80]: A.inv()
```

Out[80]:

$$\begin{bmatrix} \frac{1}{17} & \frac{7}{34} \\ \frac{2}{17} & -\frac{3}{34} \end{bmatrix}$$

```
In [81]: b = Matrix(( 12*z,5*z  ))
         b
```

Out[81]:

$$\begin{bmatrix} 12z \\ 5z \end{bmatrix}$$

```
In [82]: x = A.inv()*b
         x
```

Out[82]:

$$\begin{bmatrix} \frac{59z}{34} \\ \frac{33z}{34} \end{bmatrix}$$

```
In [83]: x.subs({z:3.3}).evalf(4)
```

Out[83]:

$$\begin{bmatrix} 5.726 \\ 3.203 \end{bmatrix}$$

```
In [84]: type(x)

Out[84]: sympy.matrices.dense.MutableDenseMatrix
```

Um método alternativo para resolver o mesmo problema é construir o sistema como uma matriz aumentada. Essa é a forma obtida dispondo juntas as colunas (no nosso exemplo) de **A** e o vetor **b**. A matriz aumentada é [1]:

$$(\mathbf{A} \,|\, \mathbf{b}) = \left(\begin{array}{cc|c} 3 & 7 & 12z \\ 4 & -2 & 5z \end{array} \right),$$

e, como antes, construímos isto como um objeto SymPy `Matrix`, mas, neste caso, passamos para a função `solve_linear_system()`:

```
In [85]: from sympy import Matrix, symbols, solve_linear_system
         x, y, z = symbols('x,y,z')
         system = Matrix(([3, 7, 12*z],[4, -2, 5*z]))
         system
```

Out[85]:

$$\begin{bmatrix} 3 & 7 & 12z \\ 4 & -2 & 5z \end{bmatrix}$$

```
In [86]: sol = solve_linear_system(system,x,y)
         sol
```

Out[86]:

$$\left\{ x : \frac{59z}{34}, \quad y : \frac{33z}{34} \right\}$$

```
In [87]: type(sol)

Out[87]: dict

In [88]: for k in sol.keys():
             print(k,'=',sol[k].subs({z:3.3}).evalf(4))

x = 5.726
y = 3.203
```

Uma terceira opção é o método `solve()`, cujos argumentos incluem as equações simbólicas individuais, em vez de matrizes. Como `dsolve()`, `solve()` espera ou expressões que ela assumirá como iguais a zero, ou objetos `Equality`, que podemos criar convenientemente com `Eq()`:

```
In [89]: from sympy import symbols,solve,Eq
         x, y, z = symbols('x,y,z')
         solve((Eq(3*x+7*y,12*z), Eq(4*x-2*y,5*z)), x, y)
```

Out[89]:

$$\left\{x : \frac{59z}{34}, \quad y : \frac{33z}{34}\right\}$$

```
In [90]: solve((3*x+7*y-12*z, 4*x-2*y-5*z), x, y)
```

Out[90]:

$$\left\{x : \frac{59z}{34}, \quad y : \frac{33z}{34}\right\}$$

Para obter mais informações, consulte `help(solve)` e `help(solve_linear_system)`.

12.1.9 Equações não-lineares

Vamos resolver uma equação simples como $x = x^2$. Existem duas soluções óbvias: $x = 0$ e $x = 1$. Como podemos pedir ao SymPy para calcular isso para nós?

```
In [91]: import sympy
         x, y, z = sympy.symbols('x, y, z')        # cria alguns símbolos
         eq = x - x ** 2                           # define a equação

In [92]: sympy.solve(eq, x)                        # resolve eq = 0
```

Out[92]:

$$[0, \quad 1]$$

A função `solve()` espera uma expressão que deve ser resolvida com o zero isolado. Para o nosso exemplo, reescrevemos $x = x^2$ como $x - x^2 = 0$ e, em seguida, a passamos para a função `solve()`.

Vamos repetir o mesmo para a equação $x = x^3$ e resolver

```
In [93]: eq = x - x ** 3                           # define a equação
         sympy.solve(eq, x)                        # resolve eq = 0
```

Out[93]:

$$[-1, \quad 0, \quad 1]$$

12.1.10 Saída: interface com LaTeX e impressão elegante

Como é o caso de muitos sistemas algébricos computacionais, o SymPy tem a capacidade de formatar sua saída como código LaTeX para facilitar sua inclusão em documentos.

No início deste capítulo, chamamos:

```
sympy.init_printing()
```

O SymPy detectou que estava em Jupyter e habilitou a saída em LaTeX. O Jupyter Notebook suporta (um pouco de) LaTeX e isso é o que nos proporciona a saída elegantemente formatada vista acima. Também podemos ver a saída em texto simples a partir do SymPy, e o código LaTeX puro que pode ser gerado:

```
In [94]: print(series(1/(x+y), y, 0, 3))

y**2/x**3 - y/x**2 + 1/x + O(y**3)


In [95]: print(latex(series(1/(x+y), y, 0, 3)))

\frac{y^{2}}{x^{3}} - \frac{y}{x^{2}} + \frac{1}{x} + \mathcal{O}\left(y^{3}\right)


In [96]: print(latex(series(1/(x+y), y, 0, 3), mode='inline'))

$\frac{y^{2}}{x^{3}} - \frac{y}{x^{2}} + 1 / x + \mathcal{O}\left(y^{3}\right)$
```

Esteja ciente de que em seu modo padrão, o código de saída produzido por `latex()` requer o pacote `amsmath`, a ser carregado através do comando `\usepackage amsmath}` no preâmbulo do documento.

O SymPy também suporta uma rotina de "impressão elegante"com a função `pprint()` (abreviatura de "pretty print"), que produz saída de texto com melhor formatação do que a rotina de impressão padrão. Observe os recursos de notação, tais como subíndices para elementos de um `array` cujos nomes são da forma `T_n`, a constante italicizada e, pontos centralizados verticalmente para a multiplicação, bordas de matrizes e frações.

Finalmente, o SymPy oferece a função `preview()`, que exibe a saída renderizada na tela (verifique `help(preview)` para obter detalhes).

12.1.11 Geração automática de código em C

Um ponto forte de muitas bibliotecas simbólicas é que elas podem converter as expressões simbólicas em código C (ou outro código), o qual, posteriormente, pode ser compilado para execução em alta velocidade. Aqui está um exemplo que demonstra isso:

```
In [97]: from sympy import *
         from sympy.utilities.codegen import codegen
         x = Symbol('x')
         sin(x).series(x, 0, 6)
```

Out[97]:

$$x - \frac{x^3}{6} + \frac{x^5}{120} + \mathcal{O}\left(x^6\right)$$

```
In [98]: print(codegen(("taylor_sine",sin(x).series(x,0,6)), language='C')[0][1])

/******************************************************************************
 *                      Code generated with sympy 1.0                         *
 *                                                                            *
 *              See http://www.sympy.org/ for more information.               *
 *                                                                            *
 *                      This file is part of 'project'                        *
 ******************************************************************************/
#include "taylor_sine.h"
#include <math.h>

double taylor_sine(double x) {

   double taylor_sine_result;
   taylor_sine_result = x - 1.0L/6.0L*pow(x, 3) + (1.0L/120.0L)*pow(x, 5) + O(x**6);
   return taylor_sine_result;

}
```

12.2 Ferramentas relacionadas

Vale a pena conferir a iniciativa SAGE (http://www.sagemath.org/), que está tentando "criar uma alternativa de fonte aberta livre e viável para Magma, Maple, Mathematica e Matlab"e inclui a biblioteca SymPy e muitas outras. Suas capacidades simbólicas são mais poderosas do que as do SymPy e o SAGE, à exceção dos recursos do SymPy, já cobrirá muitas das necessidades emergentes nos campos das ciências e engenharias. O SAGE inclui o sistema algébrico computacional Maxima, que também está disponível de forma autônoma em http://maxima.sourceforge.net/.

13 Computação numérica

13.1 Números e números

Já vimos que o Python reconhece diferentes *tipos* de números:

- números em ponto flutuante (`float`), como 3.14
- inteiros (`int`) como 42
- números complexos (`complex`), como 3.14 + 1*j

13.1.1 Limitações dos tipos de números

Limitações de `ints` A matemática fornece o conjunto infinito dos números naturais $= \{1, 2, 3, \dots\}$. Como o computador opera de modo *finito*, é impossível que ele represente todos esses números. Em vez disso, apenas um pequeno subconjunto de números é representado.

O tipo `int` (geralmente [3]) representa números entre -2147483648 e +2147483647, e corresponde a 4 *bytes* (isto é 4*8 *bits* e 232 = 4294967296, que é o intervalo de -2147483648 a +2147483647).

Você pode imaginar que o hardware usa uma tabela como esta para codificar inteiros usando bits (suponha, por simplicidade, que usemos apenas 8 *bits*):

número natural	representação binária
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
254	11111110
255	11111111

Usando 8 *bits*, podemos representar 256 números naturais (por exemplo, de 0 a 255), pois temos 28 = 256 formas diferentes de combinar oito 0s e 1s.

Poderíamos também usar uma tabela ligeiramente diferente para descrever 256 números inteiros que variem, por exemplo, de -127 a +128.

Assim é, *em princípio*, como os números inteiros são representados no computador. Dependendo do número de *bytes* utilizados, apenas números inteiros entre um valor mínimo e um máximo podem ser representados. Nos hardwares de hoje, é comum usar 4 ou 8 *bytes* para representar um número inteiro, o que leva exatamente aos valores mínimo e máximo de -2147483648 e +2147483647, como mostrado acima para 4 *bytes*, e +9223372036854775807 como o inteiro máximo para 8 *bytes* (isto é 9.21018).

Limitações de `floats` Os números em ponto flutuante em um computador não são os mesmos que os números em ponto flutuante da matemática. (Isto é exatamente o mesmo que dize que os números inteiros (matemáticos) não são os mesmos em um computador: apenas um *subconjunto* do conjunto infinito dos números inteiros pode ser representado pelo tipo `int`, como mostrado anteriormente). Então, como os números em ponto flutuante são representados no computador?

- Qualquer número real x pode ser escrito como

$$x = a \cdot 10^b,$$

onde a é a mantissa e b o expoente.

- Exemplos:

x	a	b
123.45 = 1.23456 102	1.23456	2
1000000 = 1.0 106	1.00000	6
0.0000024 = 2.4 10-6	2.40000	-6

- Portanto, podemos usar 2 inteiros para codificar um número em ponto flutuante!
- Seguindo (aproximadamente) o padrão IEEE-754, usa-se 8 *bytes* para um `float` x: estes 64 bits são divididos entre

  - 10 bits para o expoente b e
  - 54 bits para a mantissa a.

Isto resulta em

- maior número em ponto flutuante possível 10 308 (medida de qualidade para b)
- menor número em ponto flutuante possível (positivo) 10 -308 (medida de qualidade para b)
- distância entre 1.0 e o número imediatamente superior 10-16 (medida de qualidade para a)

Observe que, *a priori*, é assim como os números em ponto flutuante são armazenados (na realidade, é um pouco mais complicado).

Limitações dos números complexos O tipo `complex` tem essencialmente as mesmas limitações que o tipo `float` porque um número complexo consiste de dois números `float`: um representa a parte real, o outro a parte imaginária.

... são esses tipos de números de valor prático? Na prática cotidiana, geralmente não encontramos números que excedam 10300 (este é um número com 300 zeros!). Portanto, os números em ponto flutuante cobrem o intervalo de números de que geralmente precisamos.

No entanto, você precisa estar ciente de que, em computação científica, números pequenos e grandes são usados, os quais podem (geralmente em resultados intermediários) exceder o intervalo de números em ponto flutuante.

- Imagine, por exemplo, que tenhamos de calcular a quarta potência da constante = 1.0545716 10 -34 *kg. m2/s*:
- 4 = 1.2368136958909421 10 -136 *kg4.m8/s4*, que está a "meio caminho"de nosso menor `float` positivo representável da ordem de 10-308.

Se houver algum perigo de que possamos exceder os limites dos números em ponto flutuante, teremos que *escalonar* nossas equações de modo que (idealmente) todos os números sejam da ordem de 1. Escalonar nossas equações para que todos os números relevantes sejam próximos de 1 (normalização) também é útil para a depuração de código: se números muito maiores ou menores do que 1 surgirem, isso pode ser uma indicação de erro.

13.1.2 Usando números de ponto flutuante (sem o devido cuidado)

Já sabemos que precisamos ter cuidado para que nossos valores em ponto flutuante não excedam os limites que podem ser representados no computador (*underflow* e *overflow*). Há outra complicação devido à forma como os números em ponto flutuante têm de ser representados internamente: nem todos eles podem ser representados exatamente no computador. O número 1.0 pode ser representado exatamente, mas os números 0.1, 0.2 e 0.3 não:

```
In [23]: '%.20f' % 1.0

Out[23]: '1.00000000000000000000'

In [24]: '%.20f' % 0.1

Out[24]: '0.10000000000000000555'

In [25]: '%.20f' % 0.2

Out[25]: '0.20000000000000001110'

In [26]: '%.20f' % 0.3
```

```
Out[26]: '0.29999999999999998890'
```

Em vez disso, o número em ponto flutuante "mais próximo"do número real é escolhido.

Isso pode causar problemas. Suponha que precisemos de um laço onde x tome os valores 0,1, 0,2, 0,3, ..., 0,9, 1,0. Podemos ser tentados a escrever algo como:

```
x = 0.0
while not x == 1.0:
    x = x + 0.1
    print (" x = %19.17f" % ( x ))
```

Entretanto, este laço nunca terminará. Aqui estão as primeiras 11 linhas de saída do programa:

```
x=0.10000000000000001
x=0.20000000000000001
x=0.30000000000000004
x=0.40000000000000002
x=                  0.5
x=0.59999999999999998
x=0.69999999999999996
x=0.79999999999999993
x=0.89999999999999991
x=0.99999999999999989
x=1.09999999999999987
```

Como a variável x nunca assume exatamente o valor 1.0, o laço `while` continuará infinitamente.

Portanto: **Nunca compare dois números em ponto flutuante por igualdade!**

13.1.3 Usando números de ponto flutuante sem o devido cuidado - 1

Há várias alternativas para resolver este problema. Por exemplo, podemos comparar a distância entre dois números em ponto flutuante:

```
In [27]: x = 0.0
         while abs(x - 1.0) > 1e-8:
             x = x + 0.1
             print ( " x =%19.17f" % ( x ))

 x =0.10000000000000001
 x =0.20000000000000001
 x =0.30000000000000004
 x =0.40000000000000002
 x =0.50000000000000000
 x =0.59999999999999998
 x =0.69999999999999996
 x =0.79999999999999993
 x =0.89999999999999991
 x =0.99999999999999989
```

13.1.4 Usando números de ponto flutuante sem o devido cuidado - 2

Alternativamente, podemos iterar sobre uma sequência de inteiros e calcular o número em ponto flutuante a partir do inteiro:

```
In [28]: for i in range (1 , 11):
             x = i * 0.1
             print(" x =%19.17f" % ( x ))

 x =0.10000000000000001
 x =0.20000000000000001
 x =0.30000000000000004
 x =0.40000000000000002
 x =0.50000000000000000
 x =0.60000000000000009
 x =0.70000000000000007
 x =0.80000000000000004
 x =0.90000000000000002
 x =1.00000000000000000
```

```
In [29]: x=0.10000000000000001
         x=0.20000000000000001
         x=0.30000000000000004
         x=0.40000000000000002
         x=                 0.5
         x=0.60000000000000009
         x=0.70000000000000007
         x=0.80000000000000004
         x=0.90000000000000002
         x=                   1
```

Se compararmos esta saída com aquela do programa do caso 1, veremos que os números em ponto flutuante diferem. Isto significa que - em cálculo numérico - não é verdade que 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 1.0.

13.1.5 Cálculo simbólico

Usando o pacote SymPy, temos precisão arbitrária. Usando `sympy.Rational`, podemos definir a fração 1/10 exatamente com o cálculo simbólico. Adicionando a fração 10 vezes, seremos levados exatamente ao valor 1, conforme demonstrado por este script:

```
In [30]: from sympy import Rational
         dx = Rational (1 ,10)
         x = 0
         while x != 1.0:
             x = x + dx
             print("Atual x=%4s = %3.1f " % (x , x . evalf ()))
             print(" Atingido x=%s " % x)

Atual x=1/10 = 0.1
 Atingido x=1/10
```

```
Atual x= 1/5 = 0.2
 Atingido x=1/5
Atual x=3/10 = 0.3
 Atingido x=3/10
Atual x= 2/5 = 0.4
 Atingido x=2/5
Atual x= 1/2 = 0.5
 Atingido x=1/2
Atual x= 3/5 = 0.6
 Atingido x=3/5
Atual x=7/10 = 0.7
 Atingido x=7/10
Atual x= 4/5 = 0.8
 Atingido x=4/5
Atual x=9/10 = 0.9
 Atingido x=9/10
Atual x=   1 = 1.0
 Atingido x=1
```

No entanto, esse cálculo simbólico é muito mais lento, já que é feito através de software em vez de operações de ponto flutuante baseadas em CPU. O próximo programa aproxima os desempenhos relativos:

```
In [31]: from sympy import Rational
         dx_simbolico = Rational (1 ,10)
         dx = 0.1

         def loop_sympy (n):
             x = 0
             for i in range(n):
                 x = x + dx_simbolico
             return x

         def loop_float(n):
             x =0
             for i in range(n):
                 x = x + dx
             return x

         def medir_tempo(f, n):
             import time
             tempo_inicial = time.time()
             resultado = f(n)
             tempo_final = time.time()
             print(" desvio eh %16.15g" % ( n * dx_simbolico - resultado ))
             return tempo_final - tempo_inicial

         n = 100000
         print("laço usando float dx:")
         tempo_float = medir_tempo(loop_float, n)
```

```
        print("laço float n=%d leva %6.5f segundos" % (n, tempo_float))
        print("laço usando sympy simbólico dx:")
        tempo_sympy = medir_tempo(loop_sympy, n)
        print("laço sympy n =% d leva %6.5f segundos" % (n , tempo_sympy ))
        print("laço simbólico é um fator %.1f mais lento." % ( tempo_sympy / tempo_float ))
```

```
laco usando float dx:
 desvio eh -1.88483681995422e-08
laco float n=100000 leva 0.00944 segundos
laco usando sympy simbolico dx:
 desvio eh                 0
laco sympy n = 100000 leva 1.80431 segundos
laco simbolico eh um fator 191.1 mais lento.
```

Este é, naturalmente, um exemplo artificial: adicionamos o código simbólico para demonstrar que esses erros de arredondamento originam-se da representação aproximada de números em ponto flutuante no hardware (e, portanto, linguagens de programação). Podemos, em princípio, evitar essas complicações através de cálculo com expressões simbólicas, mas, na prática, isso é muito lento. [4]

13.1.6 Resumo

Em resumo, aprendemos que

- números em ponto flutuante e inteiros utilizados em computação numérica geralmente são bastante diferentes dos "números matemáticos"(os cálculos simbólicos são exatos e usam os "números matemáticos"):

  - há um número máximo e um número mínimo que podem ser representados (tanto para números inteiros como para ponto flutuante)
  - dentro desse intervalo, nem todo número em ponto flutuante pode ser representado no computador.

- lidamos com essa limitação:

  - nunca comparando dois números em ponto flutuante por igualdade (em vez disso, calculamos o valor absoluto da diferença)
  - uso de algoritmos que são *estáveis* (isto significa que pequenos desvios de números corretos podem ser corrigidos pelo algoritmo. Ainda não mostramos nenhum exemplo desse tipo neste documento.)

- Observe que há muito mais a dizer sobre métodos numéricos e técnicas algorítmicas para tornar a computação numérica tão precisa quanto possível, mas isto está fora do escopo desta seção.

13.1.7 Exercício: laço finito ou infinito

1. O que o seguinte fragmento de código computa? O laço terminará? Por quê?

```
eps = 1.0
while 1.0 + eps > 1.0:
    eps = eps / 2.0
print(eps)
```

14 Python Numérico (numpy): *arrays*

14.1 Introdução ao Numpy

O pacote NumPy (lido como NUMerical PYthon) dá acesso a

- uma nova estrutura de dados chamada `array` que permite
- operações vetoriais e matriciais eficientes. Ela também fornece
- uma série de operações de álgebra linear (como a resolução de sistemas de equações lineares e a computação de autovetores/autovalores).

14.1.1 Histórico

Alguns conhecimentos prévios: existem duas outras implementações que fornecem quase a mesma funcionalidade do NumPy: `Numeric` e `numarray`.

- `Numeric` foi a primeira provisão de um conjunto de métodos numéricos (semelhante ao Matlab) para Python. Ela evoluiu a partir de um projeto de doutorado.
- `Numarray` é uma implementação de `Numeric` com certas melhorias (mas, para nossos propósitos, `Numeric` e `numarray` comportam-se praticamente de modo idêntico).
- No início de 2006, decidiram combinar os melhores aspectos do `Numeric` e `numarray` no `Scipy` e fornecer o `array` ("esperançosamente", um produto final) como um tipo de dado incluído no módulo "NumPy".

No restante do material, usaremos o pacote "NumPy"como fornecido pelo (novo) SciPy. Se, por algum motivo, isso não funcionar para você, é porque, provavelmente, sua versão do SciPy seja muito antiga. Nesse caso, `Numeric` ou `numarray` podem estar instaladas e devem fornecer quase que as mesmas capacidades. [5]

14.1.2 *Arrays*

Apresentamos um novo tipo de dado (fornecido pelo NumPy) que é chamado de *array*. Uma matriz é "parecida"com uma lista, mas um *array* pode guardar apenas elementos do mesmo tipo (ao passo que uma lista pode misturar diferentes tipos de objetos). Isto significa que *arrays* são mais eficientes em armazenamento (porque não precisamos armazenar um tipo para cada elemento). *Arrays* também são a estrutura de dados de escolha para cálculos numéricos quando lidamos com vetores e matrizes. Vetores e matrizes (e matrizes com mais de dois índices) são todos chamados de *arrays* pela biblioteca NumPy.

Vetores (*Arrays* 1D) A estrutura de dados de que precisaremos mais frequentemente é um vetor. Aqui estão alguns exemplos de como podemos gerar um:

- Conversão de uma lista (ou tupla) em uma matriz usando `numpy.array`:

```
In [1]: import numpy as N
        x = N.array([0, 0.5, 1, 1.5])
        print(x)
```

```
[ 0.   0.5  1.   1.5]
```

- Criação de um vetor usando "ARRAYRange":

```
In [2]: x = N.arange(0, 2, 0.5)
        print(x)

[ 0.   0.5  1.   1.5]
```

- Criação de vetor de zeros

```
In [3]: x = N.zeros(4)
        print(x)

[ 0.  0.  0.  0.]
```

Uma vez que a matriz for estabelecida, podemos definir e recuperar valores individuais. Por exemplo:

```
In [4]: x = N.zeros(4)
        x[0] = 3.4
        x[2] = 4
        print(x)
        print(x[0])
        print(x[0:-1])

[ 3.4  0.   4.   0. ]
3.4
[ 3.4  0.   4. ]
```

Observe que, tendo o vetor, podemos executar cálculos em cada elemento no vetor com uma única declaração:

```
In [5]: x = N.arange(0, 2, 0.5)
        print(x)
        print(x + 10)
        print(x ** 2)
        print(N.sin(x))

[ 0.   0.5  1.   1.5]
[ 10.   10.5  11.   11.5]
[ 0.    0.25  1.    2.25]
[ 0.          0.47942554  0.84147098  0.99749499]
```

Matrizes (*Arrays* 2D) Aqui estão duas maneiras de criar um *array* bidimensional:

- Ao converter uma lista de listas (ou tuplas) em uma matriz:

```
In [6]: x = N.array([[1, 2, 3], [4, 5, 6]])
        x
```

```
Out[6]: array([[1, 2, 3],
               [4, 5, 6]])
```

- Usando o método zeros para criar uma matriz com 5 linhas e 4 colunas

```
In [7]: x = N.zeros((5, 4))
        x

Out[7]: array([[ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.]])
```

O "formato"de uma matriz pode ser consultado assim (aqui temos 2 linhas e 3 colunas):

```
In [8]: x=N.array([[1, 2, 3], [4, 5, 6]])
        print(x)
        x.shape

[[1 2 3]
 [4 5 6]]


Out[8]: (2, 3)
```

Elementos individuais podem ser acessados e configurados usando esta sintaxe:

```
In [9]: x=N.array([[1, 2, 3], [4, 5, 6]])
        x[0, 0]

Out[9]: 1

In [10]: x[0, 1]

Out[10]: 2

In [11]: x[0, 2]

Out[11]: 3

In [12]: x[1, 0]

Out[12]: 4

In [13]: x[:, 0]

Out[13]: array([1, 4])

In [14]: x[0,:]

Out[14]: array([1, 2, 3])
```

14.1.3 Conversão de *array* para lista ou tupla

Para converter um *array* de volta para uma lista ou tupla, podemos usar as funções padrão `list(s)` e `tuple(s)` do Python, que levam uma sequência `s` como argumento de entrada e retornam uma lista e uma tupla, respectivamente:

```
In [15]: a = N.array([1, 4, 10])
         a

Out[15]: array([ 1,  4, 10])

In [16]: list(a)

Out[16]: [1, 4, 10]

In [17]: tuple(a)

Out[17]: (1, 4, 10)
```

14.1.4 Operações de Álgebra Linear padrão

Multiplicação de matrizes Dois *arrays* podem ser multiplicados na forma usual da álgebra linear usando `numpy.matrixmultiply`. Aqui está um exemplo:

```
In [18]: import numpy as N
         import numpy.random
         A = numpy.random.rand(5, 5)     # gera uma matriz aleatória 5x5
         x = numpy.random.rand(5)        # gera um vetor de 5 elementos
         b=N.dot(A, x)                   # multiplica a matriz A pelo vetor x
```

Solução de sistemas de equações lineares Para resolver um sistema de equações $\mathbf{Ax} = \mathbf{b}$ que é dado na forma matricial (i.e. $\mathbf{A}$ é uma matriz e $\mathbf{x}$ e $\mathbf{b}$ são vetores, onde $\mathbf{A}$ e $\mathbf{b}$ são conhecidos e queremos encontrar o vetor desconhecido $\mathbf{x}$), podemos usar o pacote de álgebra linear (`linalg`) do `numpy`:

```
In [19]: import numpy.linalg as LA
         x = LA.solve(A, b)
```

Calculando Autovetores e Autovalores Aqui está um pequeno exemplo que calcula os autovetores e autovalores [triviais] (`eig`) da matriz identidade (`eye`)):

```
In [20]: import numpy
         import numpy.linalg as LA
         A = numpy.eye(3)      #'eye'->I->1 (matriz identidade)
         print(A)

[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]


In [21]: auto_valores, auto_vetores = LA.eig(A)
         print(auto_valores)
```

```
[ 1.  1.  1.]
```

```
In [22]: print(auto_vetores)
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

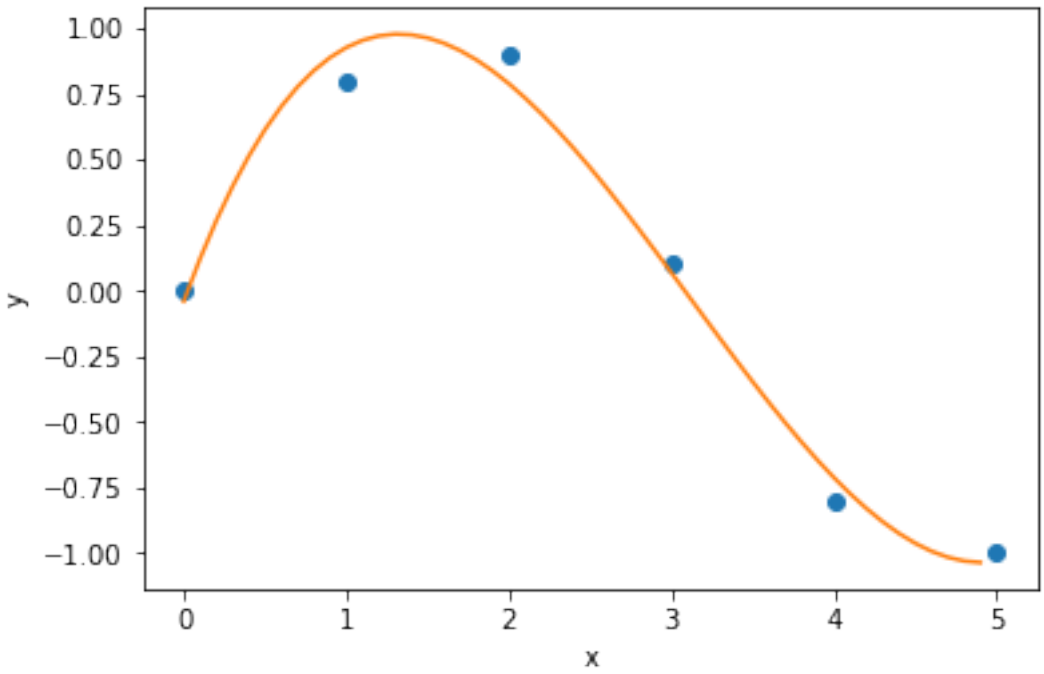
Observe que cada um desses comandos fornece sua própria documentação. Por exemplo, `help(LA.eig)` irá lhe dizer tudo sobre a função de autovetor e autovalor (uma vez que você tenha importado `numpy.linalg` como `LA`).

Ajuste polinomial de curvas Vamos supor que temos dados x-y para os quais queremos ajustar um polinômio (no sentido de quadrados mínimos).

O `NumPy` fornece a rotina `polyfit(x, y, n)` (que é semelhante à função `polyfit` do Matlab, que toma por argumentos uma lista `x</ span>` de valores `x` para os pontos dos dados, uma lista `y` de valores y para os mesmos pontos de dados e `n</ span>`, a ordem desejada do polinômio que será determinado para ajustar os dados.

```
In [23]: %matplotlib inline
         import numpy

         # demonstração - ajuste de curva: `xdata` e `ydata` são dados de entrada
         xdata = numpy.array([0.0 , 1.0 , 2.0 , 3.0 , 4.0 , 5.0])
         ydata = numpy.array([0.0 , 0.8 , 0.9 , 0.1 , -0.8 , -1.0])

         # ajuste por um polinômio cúbico (ordem = 3)
         z = numpy.polyfit(xdata, ydata, 3)

         # z é um array de coeficientes, maior grau primeiro, i . e .
         #                X^3           X^2          X            0
         # z = array ([ 0.08703704 , -0.81349206 , 1.69312169 , -0.03968254])

         # É conveniente usar objetos poly1d para lidar com polinômios:
         p = numpy.poly1d(z) # cria uma função polinomial p a partir dos coeficientes
                             # e p pode ser avaliado para todo x então.

         # cria plot
         xs = [0.1 * i for i in range (50)]
         ys = [p(x) for x in xs]   # avalia p(x) para todo `x` na lista `xs`

         import pylab
         pylab.plot(xdata, ydata, 'o', label='dados')
         pylab.plot(xs, ys, label='curva ajustada')
         pylab.ylabel('y')
         pylab.xlabel('x')
```

```
Out[23]: <matplotlib.text.Text at 0x10b7098d0>
```



A figura mostra a curva ajustada (linha contínua) juntamente com os pontos de dados calculados precisos.

14.1.5 Mais exemplos NumPy

Podem ser encontrados aqui: http://www.scipy.org/Numpy_Example_List

14.1.6 NumPy para usuários do Matlab

Há uma página dedicada que explica o módulo NumPy a partir da perspectiva de um usuário (experiente) do Matlab em http://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html.

15 Visualização de dados

O objetivo da computação científica é promover a compreensão do significado dos muitos números que calculamos. Precisamos de pós-processamento, análise estatística e visualização gráfica dos nossos dados. As seções seguintes descrevem

- Matplotlib/Pylab - que permitem gerar gráficos de alta qualidade do tipo $y = f(x)$ (e um pouco mais).

15.1 Matplotlib (Pylab): plotando $y = f(x)$ (e um pouco mais)

Matplotlib é uma biblioteca Python para plotagem 2D que produz figuras com qualidade de publicação em uma variedade de formatos impressos e interativos. O Matplotlib tenta fazer as coisas fáceis ainda mais fáceis, e as difíceis, possíveis. Você pode gerar gráficos, histogramas, espectros de energia, gráficos de barras, de erros, de dispersão, etc, com apenas algumas linhas de código.

Para informações mais detalhadas, verifique estes links

- Uma introdução muito boa sobre Matplotlib orientado a objetos e um resumo de todos os modos relevantes para mudança de estilo, tamanho de figura, largura de linhas, etc. pode ser encontrada nesta útil referência:http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-4-Matplotlib.ipynb
- Tutoriais do Matplotlib
- Homepage do Matplotlib
- Lista de exemplos simples (capturas de tela) http://matplotlib.sourceforge.net/users/screenshots.html
- Galeria de exemplos: http://matplotlib.sourceforge.net/gallery.html

15.1.1 Matplotlib e Pylab

Podemos ver o Matplotlib como uma *biblioteca de plotagem orientada a objetos*. Pylab é uma interface para o mesmo conjunto de funções que imita a interface de plotagem do Matlab.

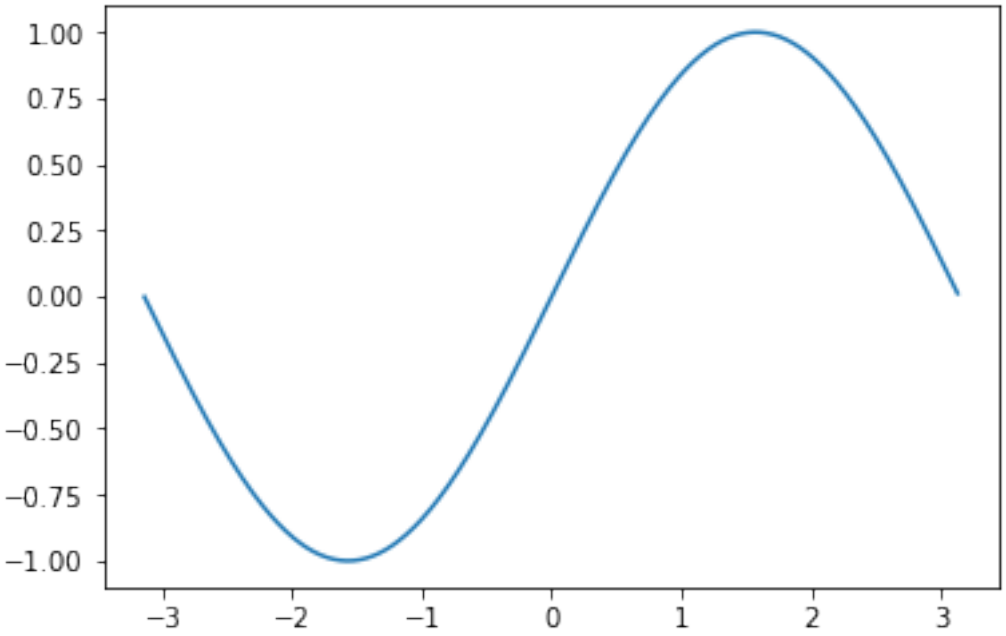
Pylab é um pouco mais conveniente de usar para plotagens fáceis e o Matplotlib dá um controle muito mais detalhado sobre a forma como os plots são criados. Se você usa o Matplotlib rotineiramente para produzir figuras, aconselhamos você a aprender sobre a interface orientada a objetos do Matplotlib (em vez da interface Pylab).

Este capítulo concentra-se na interface Pylab.

15.1.2 Primeiro exemplo

A forma recomendada de usar Matplotlib em um exemplo simples é mostrada aqui (vamos chamá-lo de exemplo 1a):

```
In [1]: %matplotlib inline

In [2]: # exemplo 1a
        import numpy as np                  # obtém acesso aos arrays do numpy
        import matplotlib.pyplot as plt     # funções de plotagem

        x = np.arange(-3.14, 3.14, 0.01)    # cria eixo x de dados
        y = np.sin(x)                       # calcula eixo y de dados
        plt.plot(x, y)                      # cria plot

Out[2]: [<matplotlib.lines.Line2D at 0x105c68dd8>]
```



15.1.3 Como importar tudo isso: matplotlib, pylab, pyplot, numpy

O sub-módulo `matplotlib.pyplot` fornece uma interface orientada a objetos para a biblioteca de plotagem. Muitos dos exemplos na documentação do Matplotlib seguem a convenção de importação do `matplotlib.pyplot` como `plt` e o `numpy` como `np`. Evidentemente, a decisão sobre importar a biblioteca `numpy` com o mome `np` (como é frequentemente feito nos exemplos da documentação), ou com o nome `N` como feito neste texto (e nos primórdios do `Numeric`, predecessor do NumPy), ou ainda com qualquer outro nome, é uma decisão plena do usuário. Da mesma forma, é uma questão de gosto pessoal se o submódulo de plotagem `matplotlib.pyplot` deve ser importado como `plt`, como é feito na documentação do Matplotlib, ou como `plot`, se a argumentação for por uma escrita mais clara, etc.

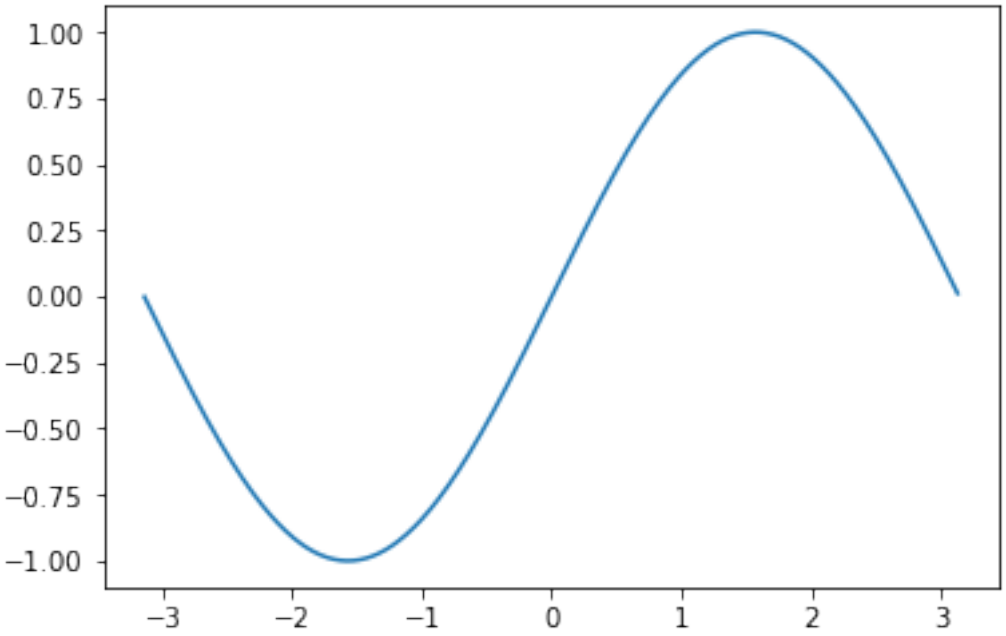
Como sempre, um equilíbrio deve ser encontrado entre preferências pessoais e consistência com a prática comum ao escolher esses nomes. A consistência com o uso comum é, certamente, mais importante se o código estiver propenso a ser usado por outros ou publicado.

Uma plotagem quase sempre precisa de *arrays* de dados numéricos e é por esta razão que o módulo NumPy é bastante usado: ele fornece manipulação rápida e eficiente de memória em Python.

Poderíamos, portanto, ter escrito o exemplo 1a acima como o exemplo 1b (que é idêntico em funcionalidade ao anterior na criação da mesma plotagem):

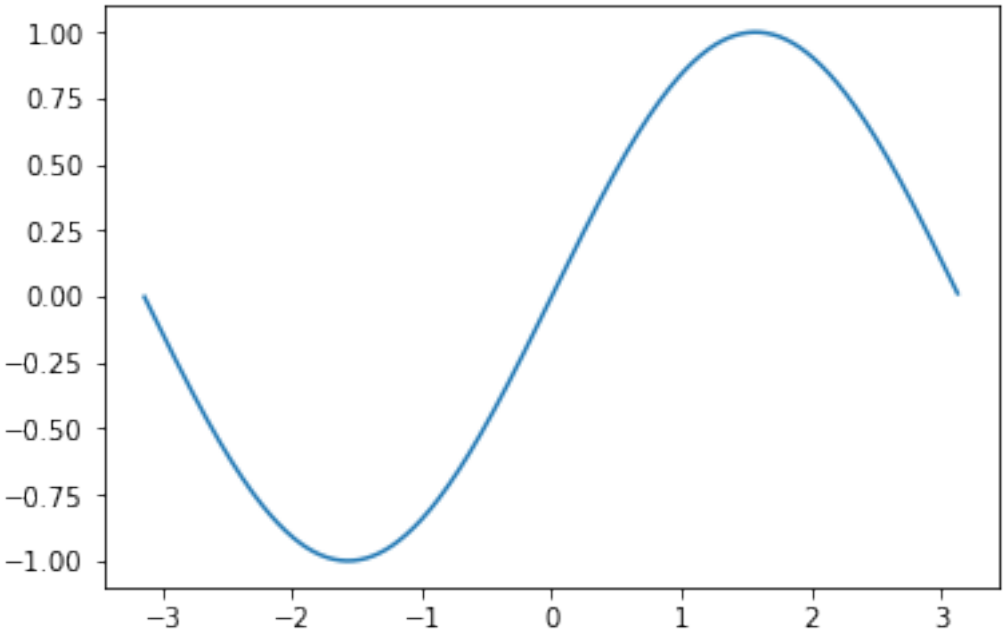
```
In [3]: # exmplo 1b
        import pylab
        import numpy as N

        x = N.arange (-3.14, 3.14, 0.01)
        y = N.sin(x)

        pylab.plot(x, y)
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x105d63828>]
```



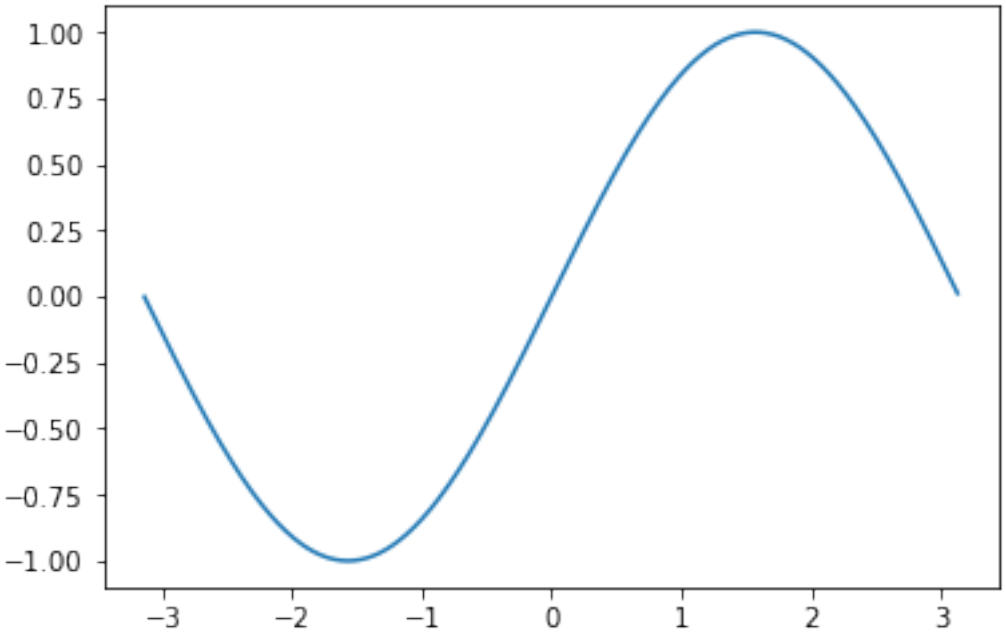
Pelo fato de `numpy.arange` e `numpy.sin` serem objetos já importados para o espaço de nomes (conveniente) `pylab`, também poderíamos ter escrito o exemplo 1c:

```
In [4]: # exemplo 1c
        import pylab as p

        x = p.arange(-3.14, 3.14, 0.01)
        y = p.sin(x)

        p.plot(x, y)

Out[4]: [<matplotlib.lines.Line2D at 0x105da3828>]
```



Se realmente quisermos reduzir a digitação de caracteres, também poderíamos importar toda a funcionalidade do módulo conveniente `pylab` e reescrever o código como no exemplo 1d:

```
In [5]: # exemplo 1d
        from pylab import *  # geralmente não recomendado
                             # OK para testes interativos

        x = arange(-3.14, 3.14, 0.01)
        y = sin(x)
        plot(x, y)
        show()
```



Isto pode ser extremamente conveniente, mas vem com um grande e "salutar"aviso:

- Enquanto usar `from pylab import *` é aceitável no prompt de comando para criar gráficos interativamente e analisar dados, isto nunca deve ser usado como regra de uso nos scripts de plotagem.
- O Pylab fornece mais de 800 objetos diferentes que são todos importados para o espaço de nomes global quando executamos o comando `from pylab import *`. Isto não é uma boa prática, e pode gerar conflitos com outros objetos que já existam ou com os que serão criados mais tarde.
- Como uma regra de uso: nunca invoque `from somewhere import *` em programas que salvamos. Esta é uma opção razoável no *prompt* de comando.

Nos exemplos a seguir, usamos a interface do `pylab` para as rotinas de plotagem, mas isto é puramente uma questão de gosto e hábito. De nenhuma maneira, é a única opção a seguir (note que os autores do Matplotlib recomendam o estilo de importação do exemplo 1a. Veja também: Matplotlib FAQ e Matplotlib, pylab, and pyplot: how are they related?)

15.1.4 Modo `inline` do IPython

Dentro do Jupyter Notebook ou Qtconsole, podemos usar o comando mágico `%matplotlib inline` para fazer novas plotagens aparecerem dentro do próprio console ou notebook. Em vez disso, para forçar janelas *pop-up*, use `%matplotlib qt`.

Há também o nome mágico `%pylab`, o qual não só irá mudar para a plotagem `inline`, mas também executará automaticamente `from pylab import *`.

15.1.5 Salvando figuras como arquivos

Depois de ter criado uma figura (usando o comando `plot`) e adicionado legendas, rótulos, etc, você tem duas opções para salvar a plotagem.

1. Exibir a figura (usando `show`) e salvá-la de forma *interativa* clicando no botão de salvamento .

2. Sem exibir a figura, salvá-la do seu código Python diretamente para arquivo. O comando a usar é `savefig`. O formato é determinado pela extensão do nome do arquivo que você fornece. Aqui está um exemplo (`pylabsavefig.py`).

```
In [6]: # pylabsavefig.py

        import pylab
        import numpy as N

        x = N.arange(-3.14, 3.14, 0.01)
        y = N.sin(x)

        pylab.plot(x, y, label='sin(x)')
        pylab.savefig('static/data/15-meuplot.png')  # salva arquivo png
        pylab.savefig('static/data/15-meuplot.eps')  # salva arquivo eps
        pylab.savefig('static/data/15-meuplot.pdf')  # salva arquivo pdf
        pylab.close()
```

Uma nota sobre formatos de arquivo: escolha o formato de arquivo `png` se você pretende incluir o seu gráfico em um documento do Word ou em uma página web. Escolha os formatos `pdf` ou `eps` se você pretende incluir a figura em um documento LaTeX - dependendo se você for compilar usando `latex` (requer `eps`) ou `pdflatex` (pode usar `pdf` [melhor] ou `png`). Se a versão do Microsoft Word (ou outro software de processamento de texto que você usa) puder lidar com arquivos `pdf`, é melhor usar `pdf` do que `png`.

Tanto `pdf` quanto `eps` são formatos vetoriais de arquivo, o que significa que podemos ampliar a imagem sem que ela perca a qualidade (as linhas ainda serão nítidas). Formatos de arquivos como `png`, `jpg`, `gif`, `tif` e `bmp` salvam a imagem em formato de mapa de bits (*bitmap*), ou seja, uma matriz de valores de cor, que aparecerá desfocada ou pixelizada ao ser ampliada ou impressa em alta resolução.

15.1.6 Modo interativo

O Matplotlib pode ser executado de dois modos:

- não interativo (padrão)
- interativo.

No modo não interativo, nenhum plot será exibido até que o comando `show()` tenha sido chamado. Neste modo, o comando `show()` deve ser a última declaração do seu programa.

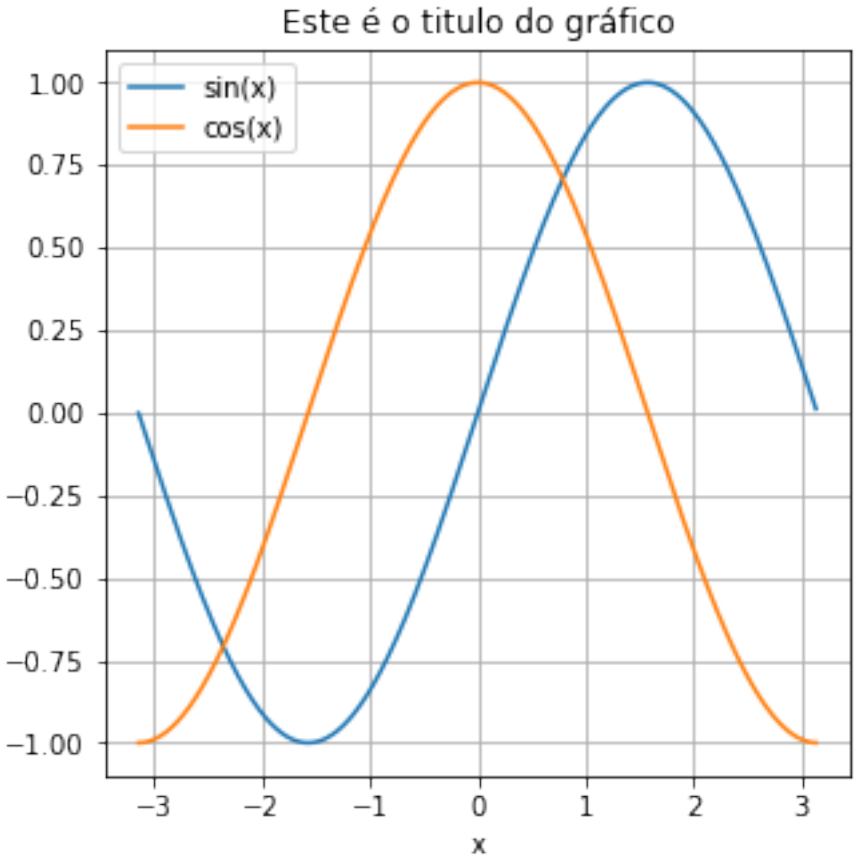
No modo interativo, os plots serão imediatamente mostrados depois que o comando de plotagem tiver sido chamado.

Pode-se ativar o modo interativo com o comando `pylab.ion()` e desativá-lo com o comando `pylab.ioff()`. O comando mágico `%matplotlib` do IPython também habilita o modo interativo.

15.1.7 Lidando com os detalhes de sua plotagem

O Matplotlib permite-lhe fazer o "ajuste fino"de suas plotagens nos mínimos detalhes. Aqui está um exemplo:

```
In [7]: import pylab
        import numpy as N

        x = N.arange(-3.14, 3.14, 0.01)
        y1 = N.sin(x)
        y2 = N.cos(x)
        pylab.figure(figsize =(5 , 5))
        pylab.plot(x, y1, label='sin(x)')
        pylab.plot(x, y2, label='cos(x)')
        pylab.legend()
        pylab.grid()
        pylab.xlabel('x')
        pylab.title('Este é o titulo do gráfico')

Out[7]: <matplotlib.text.Text at 0x105e7bd30>
```



Segue uma lista de outros comandos úteis:

- `figure(figsize=(5,5))` configura o tamanho da figura para 5x5 (polegadas)
- `plot(x,y1,label=sin(x))` a palavra-chave "label" define o nome desta linha. O rótulo da linha será mostrado na legenda se o comando `legend()` for usado posteriormente.
- Note que chamando o comando `plot` repetidamente, você poderá sobrepor uma série de curvas.

- `axis([-2,2,-1,1])` restringe a área de plotagem nos limites de `xmin=-2` a `xmax=2` na direção x e de `ymin=-1` a `ymax=1` na direção y.
- `legend()` este comando mostrará uma legenda com os rótulos definidos no comando plot. Experimente `help(pylab.legend)` para aprender mais sobre o posicionamento das legendas.
- `grid()` este comando mostrará as linhas de grade em segundo plano.
- `xlabel(...)` e `ylabel(...)` adicionam legendas aos eixos.

Além disso, você pode escolher diferentes estilos e espessuras de linha, cores e símbolos para os dados a serem plotados. A sintaxe é muito semelhante à do Matlab. Por exemplo:

- `plot(x,y,og)` plotará círculos (`o`) na cor verde (`g`)
- `plot(x,y,-r)` plotará uma linha (`-`) na cor vermelha (`r`)
- `plot(x,y,-b,linewidth=2)` plotará uma linha azul (`b`) com dois pixels de espessura `linewidth=2`, que é duas vezes mais espessa do que o padrão.

A lista completa de opções pode ser encontrada ao digitar `help(pylab.plot)` no *prompt* do Python. Como esta documentação é tão útil, repetimos partes dela aqui:

plot(*args, **kwargs) Plot lines and/or markers to the :class:`~matplotlib.axes.Axes`. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal::

```
    plot(x, y)         # plot x and y using default line style and color
    plot(x, y, 'bo')   # plot x and y using blue circle markers
    plot(y)            # plot y using x as index array 0..N-1
    plot(y, 'r+')      # ditto, but with red plusses

If *x* and/or *y* is 2-dimensional, then the corresponding columns
will be plotted.

An arbitrary number of *x*, *y*, *fmt* groups can be
specified, as in::

    a.plot(x1, y1, 'g^', x2, y2, 'g-')

Return value is a list of lines that were added.

The following format string characters are accepted to control
the line style or marker:

================    ===============================
character           description
================    ===============================
'-'                 solid line style
'--'                dashed line style
'-.'                dash-dot line style
':'                 dotted line style
'.'                 point marker
','                 pixel marker
```

```
    'o'                 circle marker
    'v'                 triangle_down marker
    '^'                 triangle_up marker
    '<'                 triangle_left marker
    '>'                 triangle_right marker
    '1'                 tri_down marker
    '2'                 tri_up marker
    '3'                 tri_left marker
    '4'                 tri_right marker
    's'                 square marker
    'p'                 pentagon marker
    '*'                 star marker
    'h'                 hexagon1 marker
    'H'                 hexagon2 marker
    '+'                 plus marker
    'x'                 x marker
    'D'                 diamond marker
    'd'                 thin_diamond marker
    '|'                 vline marker
    '_'                 hline marker
    ================    ================================

The following color abbreviations are supported:

==========  ========
character   color
==========  ========
'b'         blue
'g'         green
'r'         red
'c'         cyan
'm'         magenta
'y'         yellow
'k'         black
'w'         white
==========  ========

In addition, you can specify colors in many weird and
wonderful ways, including full names (``'green'``), hex
strings (``'#008000'``), RGB or RGBA tuples (``(0,1,0,1)``) or
grayscale intensities as a string (``'0.8'``).  Of these, the
string specifications can be used in place of a ``fmt`` group,
but the tuple forms can be used only as ``kwargs``.

Line styles and colors are combined in a single format string, as in
``'bo'`` for blue circles.

The *kwargs* can be used to set line properties (any property that has
a ``set_*`` method).  You can use this to set a line label (for auto
legends), linewidth, anitialising, marker face color, etc.  Here is an
```

```
example::

    plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
    plot([1,2,3], [1,4,9], 'rs',  label='line 2')
    axis([0, 4, 0, 10])
    legend()

If you make multiple lines with one plot command, the kwargs
apply to all those lines, e.g.::

    plot(x1, y1, x2, y2, antialised=False)

Neither line will be antialiased.

You do not need to use format strings, which are just
abbreviations.  All of the line properties can be controlled
by keyword arguments.  For example, you can set the color,
marker, linestyle, and markercolor with::

    plot(x, y, color='green', linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=12).  See
         :class:`~matplotlib.lines.Line2D` for details.
```

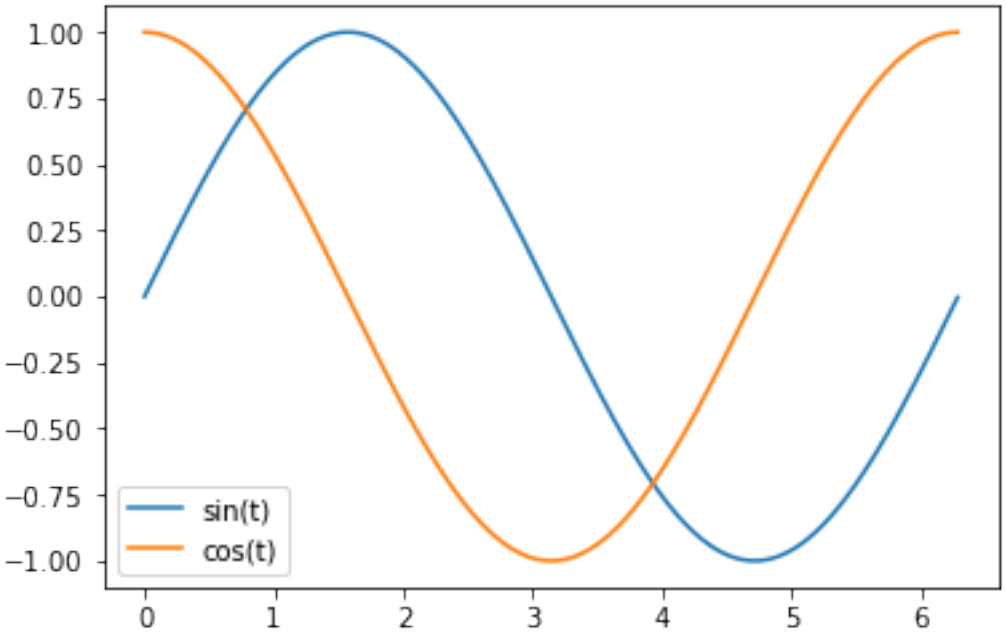
O uso de diferentes estilos de linha e espessuras é particularmente útil quando a cor não pode ser usada para distinguir linhas (por exemplo, quando o gráfico for utilizado em um documento que será impresso apenas em preto e branco).

15.1.8 Plotando mais de uma curva

Existem três métodos diferentes para plotar mais do que uma curva.

Duas (ou mais) curvas em um gráfico Chamando o comando `plot` repetidamente, mais do que uma curva pode ser desenhada no mesmo gráfico. Exemplo:

```
In [8]: import numpy as N
        t = N.arange(0,2*N.pi,0.01)

        import pylab
        pylab.plot(t,N.sin(t),label='sin(t)')
        pylab.plot(t,N.cos(t),label='cos(t)')
        pylab.legend()

Out[8]: <matplotlib.legend.Legend at 0x1062bffd0>
```

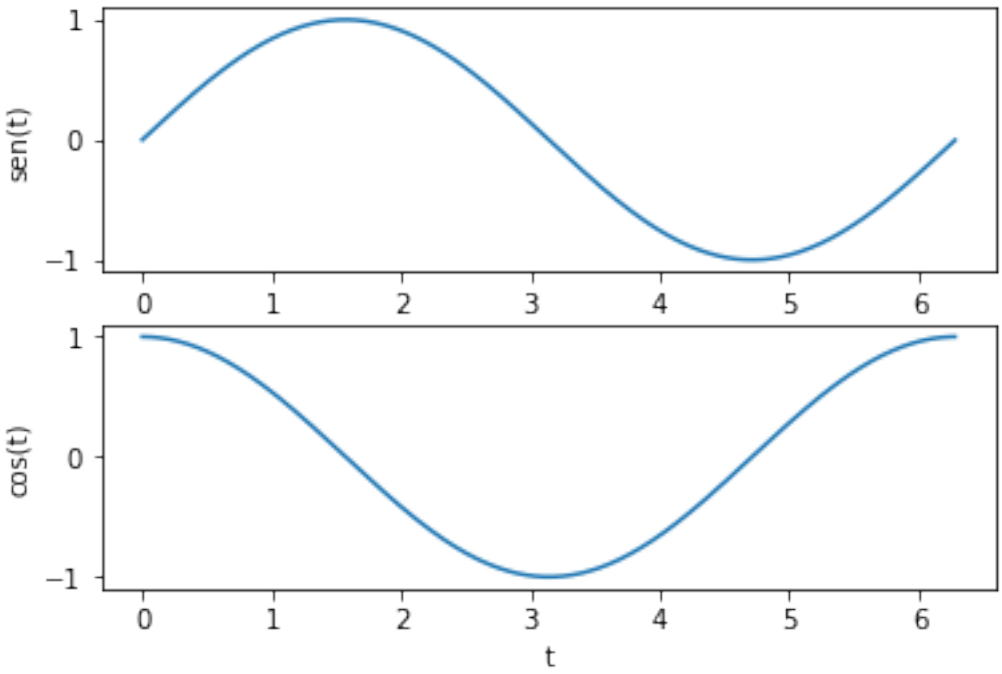


Dois (ou mais gráficos) em uma janela de figura O comando `pylab.subplot` permite-lhe organizar vários gráficos dentro da mesma janela. A sintaxe geral é

```
subplot(numRows, numCols, plotNum)
```

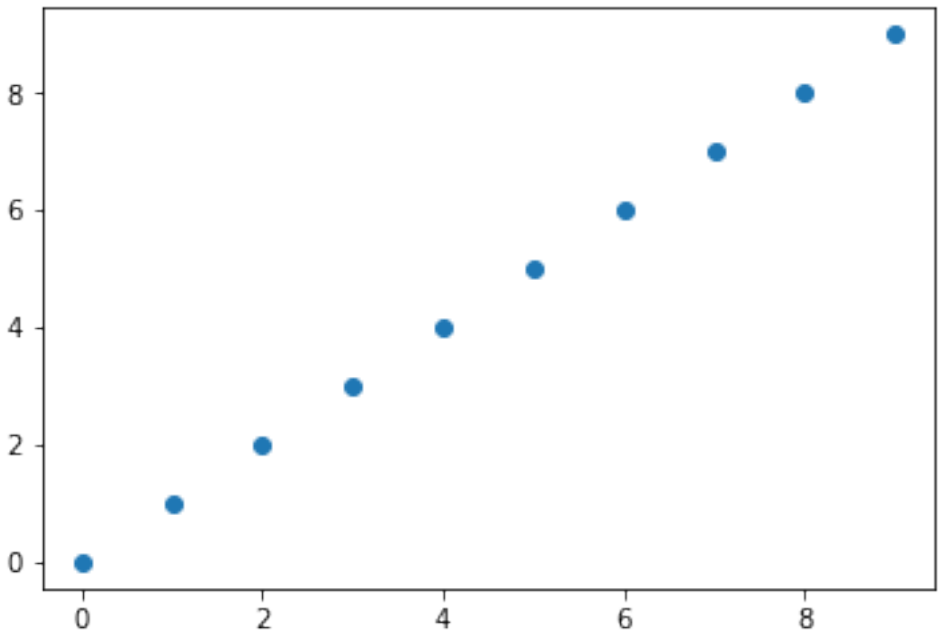
Aqui está um exemplo completo de plotagem das curvas seno e cosseno em dois gráficos alinhados um embaixo do outro na mesma janela:

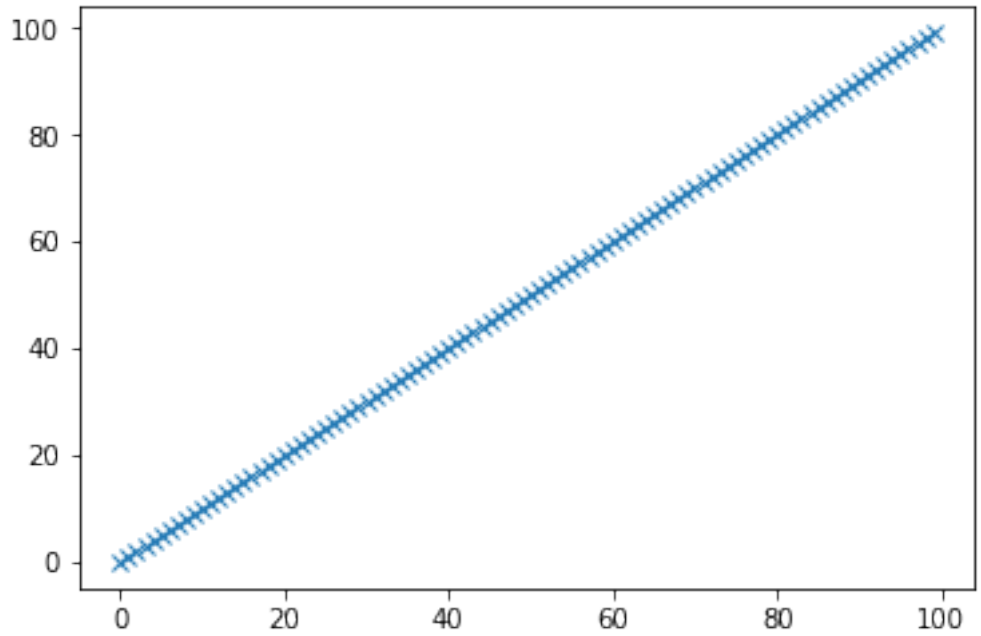
```
In [9]: import numpy as N
        t = N.arange (0 , 2 * N . pi , 0.01)

        import pylab

        pylab.subplot(2, 1, 1)
        pylab.plot(t, N.sin(t))
        pylab.xlabel('t')
        pylab.ylabel('sen(t)')

        pylab.subplot(2, 1, 2)
        pylab.plot(t, N.cos(t))
        pylab.xlabel('t')
        pylab.ylabel('cos(t)')

Out[9]: <matplotlib.text.Text at 0x1063a79e8>
```




```
In [10]: import pylab
         pylab.figure(1)
         pylab.plot(range(10),'o')

         pylab.figure(2)
         pylab.plot(range(100),'x')
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x1065622b0>]
```





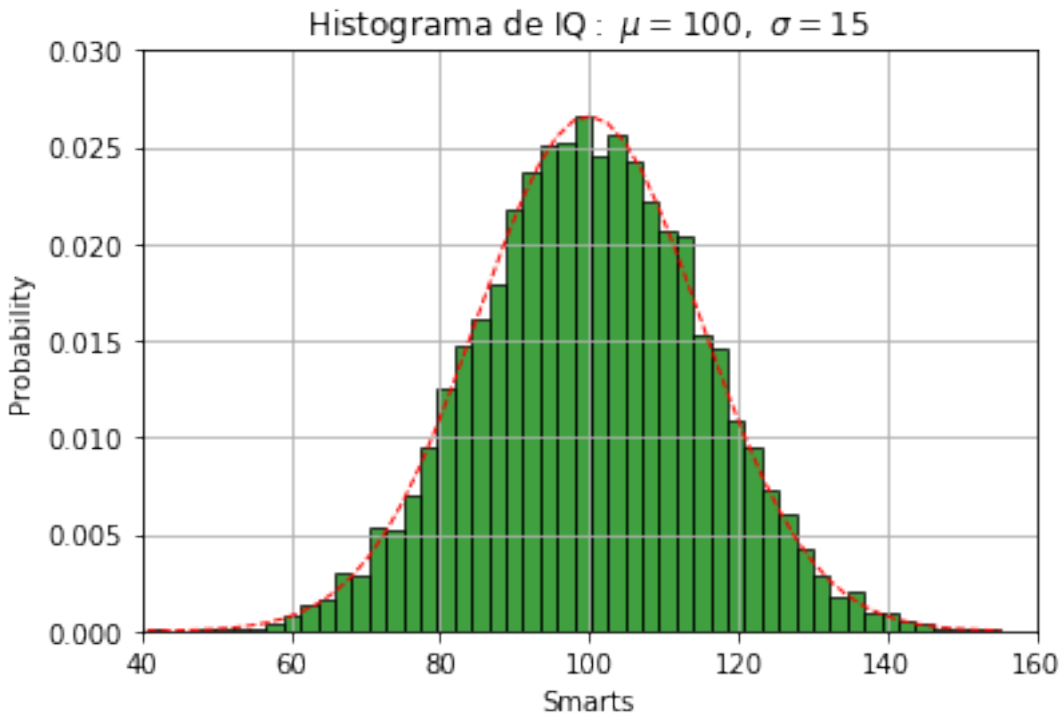
Note que você pode usar `pylab.close()` para fechar uma, algumas ou todas as janelas de figura (use `help(pylab.close)` para aprender mais).

15.1.9 Histogramas

O programa abaixo demonstra como criar histogramas a partir de dados estatísticos no Matplotlib. O plot resultante é mostrado na figura

```
In [11]: # versão modificada de
         # http://matplotlib.sourceforge.net/plot_directive/mpl_examples/...
         #                                   /pylab_examples/histogram_demo.py
         import numpy as np
         import matplotlib.mlab as mlab
         import matplotlib.pyplot as plt

         # cria dados
         mu, sigma = 100, 15
         x = mu + sigma * np.random.randn(10000)

         # histograma dos dados
         n, bins, patches = plt.hist(x, 50, normed=1,
                                     facecolor='green', alpha=0.75,edgecolor='black')

         # alguns detalhes
         plt.xlabel('Smarts')
         plt.ylabel('Probability')
```

```
# strings Latex para legendagem e títulos
plt.title(r'$\mathrm{Histograma\ de\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)


# adiciona linha de 'melhor ajuste'
y = mlab.normpdf(bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)

# salva para arquivo
plt.savefig('static/data/15-pylabhistogram.pdf')
```



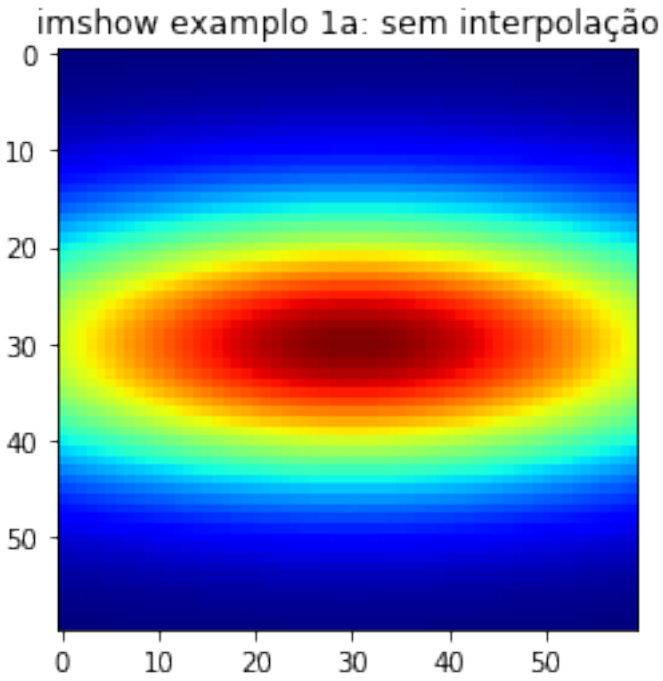
Não tente compreender cada comando neste arquivo: alguns são bastante especializados e não foram abordados neste texto. A intenção é fornecer alguns exemplos para mostrar o que pode - em princípio - ser feito com Matplotlib. Se você precisa de um plot como este, a expectativa é que você terá que experimentar e possivelmente aprender um pouco mais sobre Matplotlib.

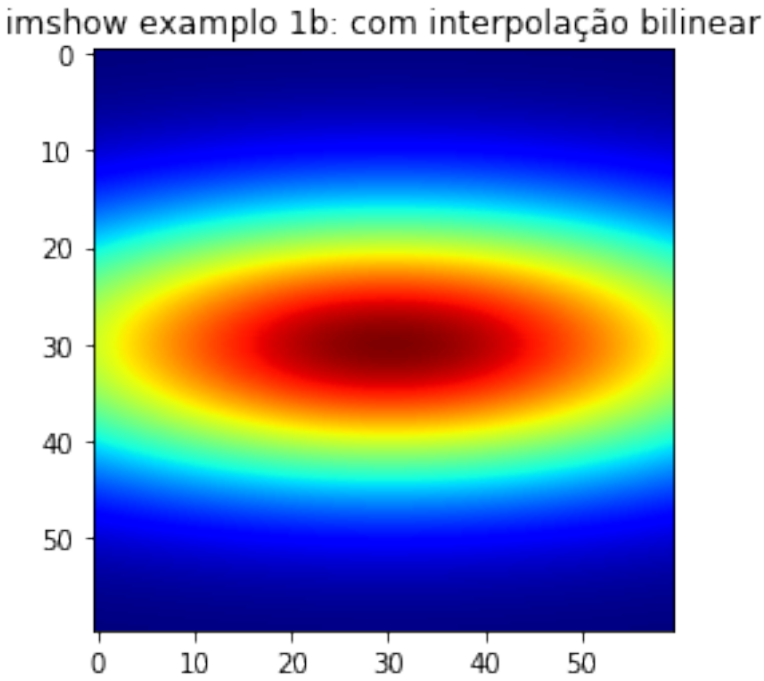
15.1.10 Visualizando dados de matrizes

O programa abaixo demonstra como criar uma plotagem em bitmap das entradas de uma matriz.

```
In [12]: import numpy as np
         import matplotlib.mlab as mlab    #  comandos compatíveis com Matlab
         import matplotlib.pyplot as plt
```

```
# cria matriz Z que contém alguns dados interessantes
delta = 0.1
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z = mlab.bivariate_normal(X, Y, 3.0, 1.0, 0.0, 0.0)

# mostra a matriz pura de dados de Z em uma figura
plt.figure(1)
plt.imshow(Z, interpolation='nearest', cmap=cm.jet)
plt.title("imshow examplo 1a: sem interpolação")
plt.savefig("static/data/15-pylabimshow1a.pdf")

# mostra os dados interpolados em outra figura
plt.figure(2)
im = plt.imshow(Z, interpolation='bilinear', cmap=cm.jet)
plt.title("imshow examplo 1b: com interpolação bilinear")
plt.savefig("static/data/15-pylabimshow1b.pdf")
```

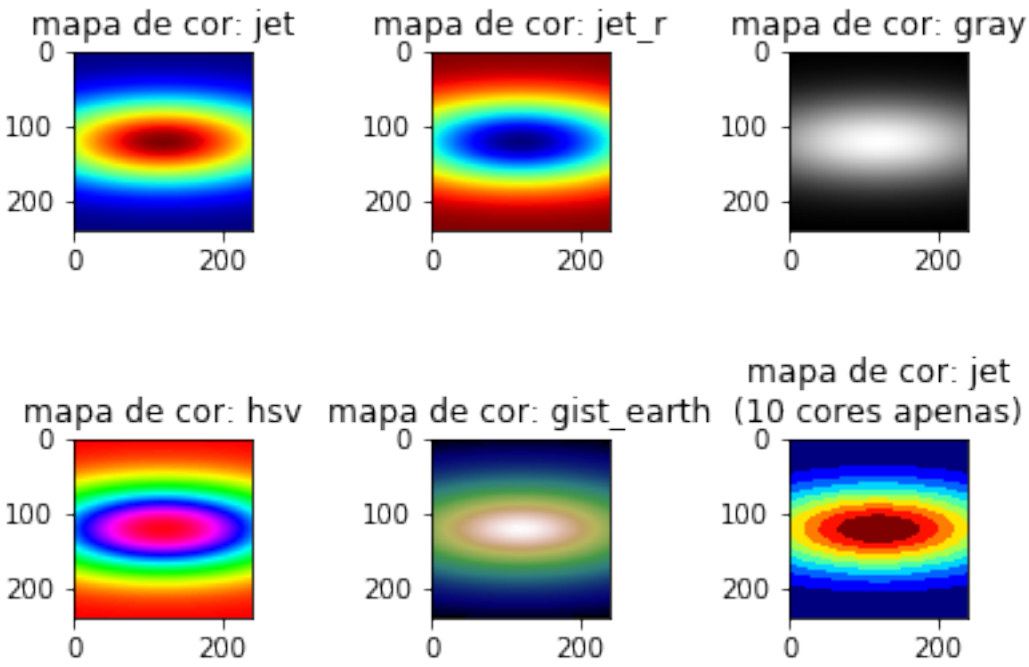




Para usar diferentes mapas de cor, fazemos uso do módulo `matplotlib.cm` (onde `cm` significa *colormap*). O código abaixo demonstra como podemos selecionar mapas de cor no conjunto de mapas já fornecidos e como podemos modificá-los (aqui, reduzindo o número de cores no mapa). O último exemplo imita o comportamento do comando mais sofisticado `contour` que também vem com o `matplotlib`.

```
In [13]: import numpy as np
         import matplotlib.mlab as mlab    #  comandos compativeis com Matlab
         import matplotlib.pyplot as plt
         import matplotlib.cm as cm        # submódulo de mapas de cor

         # cria matriz Z que contém alguns dados interessantes
         delta = 0.025
         x = y = np.arange(-3.0, 3.0, delta)
         X, Y = np.meshgrid(x, y)
         Z = mlab.bivariate_normal(X, Y, 3.0, 1.0, 0.0, 0.0)

         # ajuste de espaçamento dos subplots para evitar sobreposição
         aux = 1.0
         subplots_adjust(wspace=aux, hspace=aux)

         Nx, Ny = 2, 3
         plt.subplot(Nx, Ny, 1)  # próximo plot será mostrado no
                                 # primeiro subplot como uma
                                 # matriz Nx x Ny de subplots

         plt.imshow(Z, cmap=cm.jet)  # mapa de cor padrão 'jet'
         plt.title("mapa de cor: jet")
```

```
plt.subplot(Nx, Ny, 2)   # próximo plot para segundosubplot
plt.imshow(Z, cmap=cm.jet_r)  # mapa de cor jet reverso
plt.title("mapa de cor: jet_r")

plt.subplot(Nx, Ny, 3)
plt.imshow(Z, cmap=cm.gray)
plt.title("mapa de cor: gray")

plt.subplot(Nx, Ny, 4)
plt.imshow(Z, cmap=cm.hsv)
plt.title("mapa de cor: hsv")

plt.subplot(Nx, Ny, 5)
plt.imshow(Z, cmap=cm.gist_earth)
plt.title("mapa de cor: gist_earth")

plt.subplot(Nx, Ny, 6)

# produz isolinhas reduzindo o número de cores para 10
mycmap = cm.get_cmap('jet', 10)    # 10 cores discretas
plt.imshow(Z, cmap=mycmap)
plt.title("mapa de cor: jet\n(10 cores apenas)")

plt.savefig("static/data/15-pylabimshowcm.pdf")
```



15.1.11 Plots de $z = f(x,y)$ e outros recursos do Matplotlib

O Matplotlib tem um grande número de recursos e pode criar todo padrão de plotagens (1D e 2D): tais como histogramas, gráficos circulares, gráficos de dispersão, plotagens de intensidade 2D (i.e $z = f(x,y)$ e linhas de contorno) e muito mais. O programa `contour_demo.py` é um exemplo padrão do pacote `pylab` (vide figura abaixo). O seguinte link fornece o código-fonte para produzir este tipo de plotagem: contour_demo.py.

Outros exemplos são:

- http://matplotlib.org/users/screenshots.html
- http://matplotlib.org/gallery.html
- Recentemente, a criação de plotagens 3D foi adicionada ao pylab: http://matplotlib.org/examples/mplot3d/index.html#mplot3d-examples

15.2 Visualizando dados em dimensões superiores

Muitas vezes, precisamos entender os dados definidos em posições no espaço 3D. Os dados em si são muitas vezes um campo escalar (como uma temperatura) ou um vetor 3D (como velocidade ou campo magnético), ou ocasionalmente um tensor. Por exemplo, para um campo vetorial 3D (definido como $\vec{f}(\vec{x})$, onde $\vec{x} \in \mathbb{R}^3$ e $\vec{f}(\vec{x}) \in \mathbb{R}^3$), poderíamos desenhar uma seta 3D em cada ponto (grade) no espaço. É comum que esses conjuntos de dados dependam do tempo.

A biblioteca provavelmente mais comumente utilizada nas ciências e engenharia para visualizar tais conjuntos de dados é provavelmente o VTK, *Visualization ToolKit* (http://vtk.org). Esta é uma importante biblioteca C++ com interfaces para linguagens de alto nível, incluindo o Python.

Pode-se chamar essas rotinas diretamente do código Python ou escrever os dados no disco em um formato que a biblioteca VTK pode ler (chamados arquivos *vtk*) e, em seguida, usar um programa como Mayavi, ParaView ou VisIt para ler esses arquivos de dados e manipulá-los (com uma interface gráfica). Todos estes três estão usando a biblioteca VTK internamente e podem ler arquivos *vtk*.

Este pacote é muito adequado para visualizar campos estáticos e temporais 2D e 3D (campos escalares, vetoriais e tensoriais).

15.2.1 Mayavi, Paraview, Visit

- *Homepage* do Mayavi: http://code.enthought.com/projects/mayavi/
- *Homepage* do Paraview: http://paraview.org
- *Homepage* do VisIt: https://wci.llnl.gov/simulation/computer-codes/visit/

15.2.2 Escrevendo arquivos vtk a partir do Python (pyvtk)

Uma pequena, mas poderosa biblioteca Python está disponível em https://code.google.com/p/pyvtk/. Ela permite criar arquivos *vtk* a partir de estruturas de dados do Python muito facilmente.

Dada uma malha de elementos finitos ou um conjunto de dados de diferenças finitas em Python, pode-se usar o *pyvtk* para escrever esses dados em arquivos e, em seguida, usar uma das aplicações de visualização listadas acima para carregar os arquivos *vtk* para exibi-los e investigá-los.

16 Métodos numéricos usando Python (SciPy)

16.1 Visão geral

O núcleo da linguagem Python (incluindo as bibliotecas padrão) fornece funcionalidades suficiente para realizar as tarefas de pesquisa computacional. No entanto, existem bibliotecas Python dedicadas

(de terceiros) que oferecem funcionalidades estendidas:

- ferramentas numéricas para tarefas freqüentes
- que são convenientes
- e mais eficientes em termos de tempo de processamento e requisitos de memória do que as funcionalidades do Python operando sozinhas.

Nós enumeramos três desses módulos, em particular:

- o módulo NumPy fornece um tipo de dado especializado para operar numericamente com vetores e matrizes (este é o tipo `array` fornecido pelo NumPy) além de ferramentas de álgebra linear.
- o módulo Matplotlib (também conhecido como Pylab) fornece recursos de plotagem e visualização e
- o módulo `SciPy` (*SCIentific PYthon*), que fornece uma infinidade de algoritmos numéricos (introduzido neste capítulo).

Muitos dos algoritmos numéricos disponíveis através do SciPy e NumPy são fornecidos por bibliotecas compiladas estabelecidas que são muitas vezes escritas em Fortran ou C. Eles, por sua vez, serão executados muito mais rápido do que o código Python puro (interpretado). Como regra geral, esperamos que o código compilado seja duas ordens de magnitude mais rápido do que o código Python puro.

Você pode usar a função de ajuda para cada método numérico para descobrir mais sobre a origem da implementação.

16.2 SciPy

O SciPy é construído com base no NumPy. Toda a funcionalidade do NumPy parece estar disponível também no SciPy. Por exemplo, em vez de:

```
In [25]: import numpy
         x = numpy.arange(0, 10, 0.1)
         y = numpy.sin(x)
```

podemos também usar:

```
In [26]: import scipy as s
         x = s.arange(0, 10, 0.1)
         y = s.sin(x)
```

Primeiro, precisamos importar o SciPy

```
In [27]: import scipy as s
```

O pacote SciPy fornece informações sobre sua própria estrutura quando usamos o comando de ajuda:

```
help(scipy)
```

A saída é muito longa. Então, estamos mostrando apenas parte dela aqui:

```
stats       --- Statistical Functions [*]
sparse      --- Sparse matrix [*]
lib         --- Python wrappers to external libraries [*]
linalg      --- Linear algebra routines [*]
signal      --- Signal Processing Tools [*]
misc        --- Various utilities that don't have another home.
interpolate --- Interpolation Tools [*]
optimize    --- Optimization Tools [*]
cluster     --- Vector Quantization / Kmeans [*]
fftpack     --- Discrete Fourier Transform algorithms [*]
io          --- Data input and output [*]
integrate   --- Integration routines [*]
lib.lapack  --- Wrappers to LAPACK library [*]
special     --- Special Functions [*]
lib.blas    --- Wrappers to BLAS library [*]
  [*] - using a package requires explicit import (see pkgload)
```

Se estivermos procurando por um algoritmo para integrar uma função, podemos explorar o pacote `integrate`:

```
import scipy.integrate

scipy.integrate?
```

produz:

```
=============================================
Integration and ODEs (:mod:`scipy.integrate`)
=============================================

.. currentmodule:: scipy.integrate

Integrating functions, given function object
============================================

.. autosummary::
   :toctree: generated/

   quad          -- General purpose integration
   dblquad       -- General purpose double integration
   tplquad       -- General purpose triple integration
   nquad         -- General purpose n-dimensional integration
   fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n
   quadrature    -- Integrate with given tolerance using Gaussian quadrature
   romberg       -- Integrate func using Romberg integration
   quad_explain  -- Print information for use of quad
   newton_cotes  -- Weights and error coefficient for Newton-Cotes integration
   IntegrationWarning -- Warning on issues during integration

Integrating functions, given fixed samples
==========================================
```

```
.. autosummary::
   :toctree: generated/

   trapz          -- Use trapezoidal rule to compute integral.
   cumtrapz       -- Use trapezoidal rule to cumulatively compute integral.
   simps          -- Use Simpson's rule to compute integral from samples.
   romb           -- Use Romberg Integration to compute integral from
                  -- (2**k + 1) evenly-spaced samples.

.. seealso::

   :mod:`scipy.special` for orthogonal polynomials (special) for Gaussian
   quadrature roots and weights for other weighting factors and regions.

Integrators of ODE systems
==========================

.. autosummary::
   :toctree: generated/

   odeint         -- General integration of ordinary differential equations.
   ode            -- Integrate ODE using VODE and ZVODE routines.
   complex_ode    -- Convert a complex-valued ODE to real-valued and integrate.
```

As seções seguintes mostram exemplos que demonstram como empregar os algoritmos fornecidos pelo SciPy.

16.3 Integração numérica

O SciPy fornece uma série de rotinas de integração. Uma ferramenta com a finalidade de resolver integrais I do tipo

$$I = \int_a^b f(x)\,dx$$

é fornecida pela função `quad()` do módulo `scipy.integrate`.

Ela usa como argumento de entrada, a função a ser integrada $f(x)$ ("integrando") e os limites inferior a e superior b. Ela retorna dois valores (em uma tupla): o primeiro é o resultado calculado e o segundo é uma estimativa do erro numérico desse resultado.

Aqui está um exemplo que produz esta saída:

```
In [1]: from math import cos, exp, pi
        from scipy.integrate import quad

        # função a ser integrada
        def f(x):
            return exp(cos(-2 * x * pi)) + 3.2

        # chamamos quad para integrar f de -2 a 2
        res, err = quad(f, -2, 2)

        print("O resultado numérico é {:f} (+-{:g})".format(res, err))
```

```
O resultado numérico é 17.864264 (+-1.55113e-11)
```

Note que `quad()` usa os parâmetros opcionais `epsabs` e `epsrel` para aumentar ou diminuir a precisão de sua computação. (Use `help(quad)` para saber mais.) Os valores padrão são `epsabs = 1.5e-8` e `epsrel = 1.5e-8`. Para o próximo exercício, os valores padrão são suficientes.

16.3.1 Exercício: integrar uma função

1. Usando a função `quad` do SciPy, escreva um programa que resolva numericamente o seguinte: $I = \int_0^1 \cos(2\pi x)\, dx$.
2. Encontre a integral analítica e compare-a com a solução numérica.
3. Por que é importante ter uma estimativa da precisão (ou o erro) da integral numérica?

16.3.2 Exercício: plote antes de integrar

É uma boa prática plotar a função integrando para verificar se ela é "bem comportada"antes de tentar integrá-la. Singularidades (ou seja, valores de x para os quais $f(x)$ tende para menos ou mais infinito) ou outro comportamento irregular (como $f(x) = \sin(\frac{1}{x})$ perto de $x = 0$) são difíceis de manusear numericamente.

1. Escreva uma função com o nome `plotquad` que leva os mesmos argumentos que o comando quad (i.e. f, a e b) e que

- (i) crie um gráfico do integrando $f(x)$ e
- (ii) calcule a integral numérica usando a função `quad`. Os valores de retorno devem ser como aqueles para a função `quad`.

16.4 Resolvendo equações diferenciais ordinárias (EDOs)

Para resolver uma EDO do tipo

$$\frac{dy}{dt}(t) = f(y,t)$$

com uma dada condição inicial $y(t_0) = y_0$, podemos usar a função `odeint` do SciPy. Aqui está um programa de exemplo auto-explicativo (`useodeint.py`) para encontar

$$y(t) \quad \text{para} \quad t \in [0,2]$$

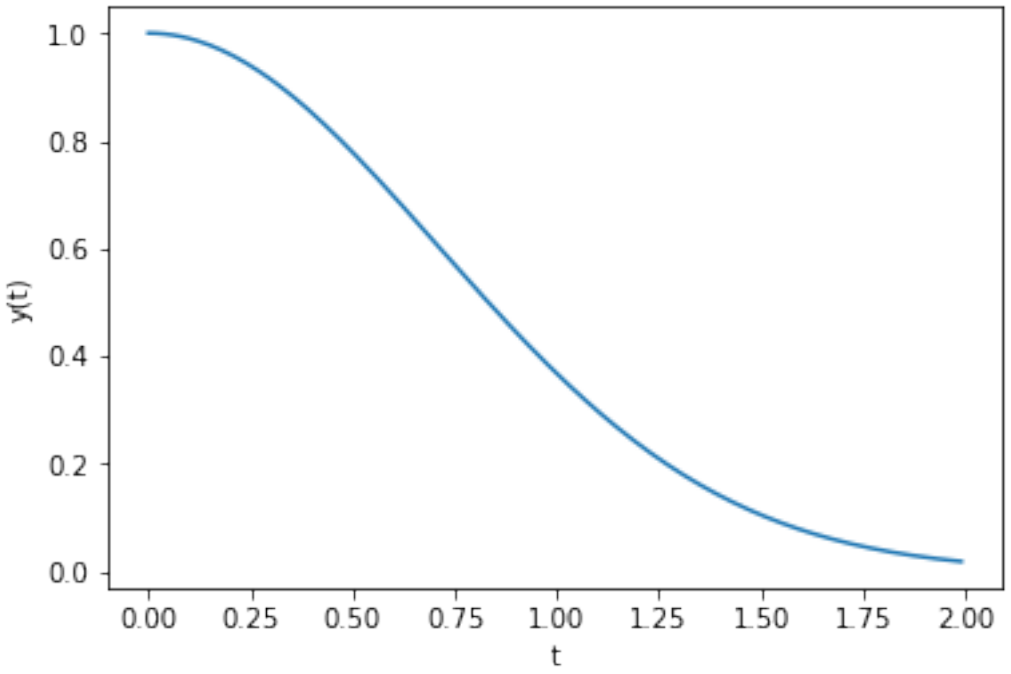
dada a EDO :

$$\frac{dy}{dt}(t) = -2yt \quad \text{com} \quad y(0) = 1.$$

```
In [2]: %matplotlib inline
        from scipy.integrate import odeint
        import numpy as N

        def f(y, t):
            """este é o lado direito da EDO a ser integrada, i.e. dy/dt = f(y,t)"""
            return -2 * y * t
```

```
y0 = 1              # valor inicial
a = 0               # limites de integração para t
b = 2

t = N.arange(a, b, 0.01)  # valores de t para
                          # os quais queremos
                          # a solução y(t)
y = odeint(f, y0, t)  # cálculo de y(t)

import pylab          # plotagem dos resultados
pylab.plot(t, y)
pylab.xlabel('t'); pylab.ylabel('y(t)')
```

```
Out[2]: <matplotlib.text.Text at 0x1128f51d0>
```



O comando `odeint` usa uma série de parâmetros opcionais para alterar a tolerância padrão de erro da integração (e para acionar a produção de saída de depuração). Use o comando de ajuda para explorá-los:

```
help(scipy.integrate.odeint)
```

irá mostrar:

```
Help on function odeint in module scipy.integrate.odepack:

odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=N
    Integrate a system of ordinary differential equations.
```

```
Solve a system of ordinary differential equations using lsoda from the
FORTRAN library odepack.

Solves the initial value problem for stiff or non-stiff systems
of first order ode-s::

    dy/dt = func(y, t0, ...)

where y can be a vector.

*Note*: The first two arguments of ``func(y, t0, ...)`` are in the
opposite order of the arguments in the system definition function used
by the `scipy.integrate.ode` class.

Parameters
----------
func : callable(y, t0, ...)
    Computes the derivative of y at t0.
y0 : array
    Initial condition on y (can be a vector).
t : array
    A sequence of time points for which to solve for y.  The initial
    value point should be the first element of this sequence.
args : tuple, optional
    Extra arguments to pass to function.
Dfun : callable(y, t0, ...)
    Gradient (Jacobian) of `func`.
col_deriv : bool, optional
    True if `Dfun` defines derivatives down columns (faster),
    otherwise `Dfun` should define derivatives across rows.
full_output : bool, optional
    True if to return a dictionary of optional outputs as the second output
printmessg : bool, optional
    Whether to print the convergence message

Returns
-------
y : array, shape (len(t), len(y0))
    Array containing the value of y for each desired time in t,
    with the initial value `y0` in the first row.
infodict : dict, only returned if full_output == True
    Dictionary containing additional output information

    =======  ============================================================
    key      meaning
    =======  ============================================================
    'hu'     vector of step sizes successfully used for each time step.
    'tcur'   vector with the value of t reached for each time step.
             (will always be at least as large as the input times).
    'tolsf'  vector of tolerance scale factors, greater than 1.0,
```

```
            computed when a request for too much accuracy was detected.
    'tsw'   value of t at the time of the last method switch
            (given for each time step)
    'nst'   cumulative number of time steps
    'nfe'   cumulative number of function evaluations for each time step
    'nje'   cumulative number of jacobian evaluations for each time step
    'nqu'   a vector of method orders for each successful step.
    'imxer' index of the component of largest magnitude in the
            weighted local error vector (e / ewt) on an error return, -1
            otherwise.
    'lenrw' the length of the double work array required.
    'leniw' the length of integer work array required.
    'mused' a vector of method indicators for each successful time step:
            1: adams (nonstiff), 2: bdf (stiff)
    =======  ============================================================

Other Parameters
----------------
ml, mu : int, optional
    If either of these are not None or non-negative, then the
    Jacobian is assumed to be banded.  These give the number of
    lower and upper non-zero diagonals in this banded matrix.
    For the banded case, `Dfun` should return a matrix whose
    rows contain the non-zero bands (starting with the lowest diagonal).
    Thus, the return matrix `jac` from `Dfun` should have shape
    ``(ml + mu + 1, len(y0))`` when ``ml >=0`` or ``mu >=0``.
    The data in `jac` must be stored such that ``jac[i - j + mu, j]``
    holds the derivative of the `i`th equation with respect to the `j`th
    state variable.  If `col_deriv` is True, the transpose of this
    `jac` must be returned.
rtol, atol : float, optional
    The input parameters `rtol` and `atol` determine the error
    control performed by the solver.  The solver will control the
    vector, e, of estimated local errors in y, according to an
    inequality of the form ``max-norm of (e / ewt) <= 1``,
    where ewt is a vector of positive error weights computed as
    ``ewt = rtol * abs(y) + atol``.
    rtol and atol can be either vectors the same length as y or scalars.
    Defaults to 1.49012e-8.
tcrit : ndarray, optional
    Vector of critical points (e.g. singularities) where integration
    care should be taken.
h0 : float, (0: solver-determined), optional
    The step size to be attempted on the first step.
hmax : float, (0: solver-determined), optional
    The maximum absolute step size allowed.
hmin : float, (0: solver-determined), optional
    The minimum absolute step size allowed.
ixpr : bool, optional
    Whether to generate extra printing at method switches.
```

```
mxstep : int, (0: solver-determined), optional
    Maximum number of (internally defined) steps allowed for each
    integration point in t.
mxhnil : int, (0: solver-determined), optional
    Maximum number of messages printed.
mxordn : int, (0: solver-determined), optional
    Maximum order to be allowed for the non-stiff (Adams) method.
mxords : int, (0: solver-determined), optional
    Maximum order to be allowed for the stiff (BDF) method.

See Also
--------
ode : a more object-oriented integrator based on VODE.
quad : for finding the area under a curve.

Examples
--------
The second order differential equation for the angle `theta` of a
pendulum acted on by gravity with friction can be written::

    theta''(t) + b*theta'(t) + c*sin(theta(t)) = 0

where `b` and `c` are positive constants, and a prime (') denotes a
derivative.  To solve this equation with `odeint`, we must first convert
it to a system of first order equations.  By defining the angular
velocity ``omega(t) = theta'(t)``, we obtain the system::

    theta'(t) = omega(t)
    omega'(t) = -b*omega(t) - c*sin(theta(t))

Let `y` be the vector [`theta`, `omega`].  We implement this system
in python as:

>>> def pend(y, t, b, c):
...     theta, omega = y
...     dydt = [omega, -b*omega - c*np.sin(theta)]
...     return dydt
...

We assume the constants are `b` = 0.25 and `c` = 5.0:

>>> b = 0.25
>>> c = 5.0

For initial conditions, we assume the pendulum is nearly vertical
with `theta(0)` = `pi` - 0.1, and it initially at rest, so
`omega(0)` = 0.  Then the vector of initial conditions is

>>> y0 = [np.pi - 0.1, 0.0]
```

```
We generate a solution 101 evenly spaced samples in the interval
0 <= `t` <= 10.  So our array of times is:

>>> t = np.linspace(0, 10, 101)

Call `odeint` to generate the solution.  To pass the parameters
`b` and `c` to `pend`, we give them to `odeint` using the `args`
argument.

>>> from scipy.integrate import odeint
>>> sol = odeint(pend, y0, t, args=(b, c))

The solution is an array with shape (101, 2).  The first column
is `theta(t)`, and the second is `omega(t)`.  The following code
plots both components.

>>> import matplotlib.pyplot as plt
>>> plt.plot(t, sol[:, 0], 'b', label='theta(t)')
>>> plt.plot(t, sol[:, 1], 'g', label='omega(t)')
>>> plt.legend(loc='best')
>>> plt.xlabel('t')
>>> plt.grid()
>>> plt.show()
```

16.4.1 Exercício: usando `odeint`

1. Abra um novo arquivo com o nome `testeEDOInt.py` em um editor de texto.
2. Escreva um programa que calcule a solução $y(t)$ da EDO seguinte usando o algoritmo `odeint`:
$$\frac{dy}{dt} = -\exp(-t)(10\sin(10t) + \cos(10t))$$
De $t = 0$ a $t = 10$. O valor inicial é $y(0) = 1$.
3. Você deve exibir a solução graficamente nos pontos $t = 0$, $t = 0.01$, $t = 0.02$, ..., $t = 9.99$, $t = 10$.

16.5 Localização de raízes

Se você tentar encontrar um x tal que
$$f(x) = 0,$$
então este é um problema chamado de *localização de raízes*. Observe que problemas como $g(x) = h(x)$ caem nesta categoria, pois você pode reescrevê-los como $f(x) = g(x) - h(x) = 0$.

Várias ferramentas para localização de raízes estão disponíveis no módulo `optimize` do SciPy.

16.5.1 Localização de raiz pelo método de bisecção

Primeiro, apresentamos o algoritmo `bisect`, que é (i) robusto e (ii) lento, mas conceitualmente muito simples.

Suponhamos que precisemos calcular as raízes de $f(x) = x^3 - 2x^2$. Esta função tem uma raiz dupla em $x = 0$ (isto é trivial de ver) e outra raiz localizada entre $x = 1.5$ (pois $f(1.5) = -1.125$) e $x = 3$ (pois $f(3) = 9$). É direto ver que esta outra raiz está localizada em $x = 2$. Aqui está um programa que determina esta raiz numericamente:

```
In [3]: from scipy.optimize import bisect

        def f(x):
            """retorna f(x) = x^3 - 2x^2. Tem raízes em
            x = 0 (raiz dupla) e x = 2"""
            return x ** 3 - 2 * x ** 2

        # o programa principal começa aqui
        x = bisect(f, 1.5, 3, xtol=1e-6)

        print("A raiz x é aproximadamente x=%14.12g,\n"
              "o erro é menor do que 1e-6." % (x))
        print("O erro exato é %g." % (2 - x))
```

```
A raiz x é aproximadamente x= 2.00000023842,
o erro é menor do que 1e-6.
O erro exato é -2.38419e-07.
```

O método `bisect()` requer três argumentos obrigatórios: (i) a função $f(x)$, (ii) um limite inferior a (para o qual escolhemos 1.5 no nosso exemplo) e (ii) um limite superior b (para o qual escolhemos 3). O parâmetro opcional `xtol` determina o erro máximo do método.

Um dos requisitos do método da bisecção é que o intervalo $[a,b]$ deve ser escolhido de tal forma que a função seja positiva em a e negativa em b, ou que a função seja negativa em a e positiva em b. Em outras palavras, a e b devem incluir uma raiz.

16.5.2 Exercício: localização de raízes usando o método `bisect`

1. Escreva um programa com o nome `sqrt2.py` para determinar uma aproximação para $\sqrt{2}$ encontrando uma raiz x da função $f(x) = 2 - x^2$ usando o algoritmo da bisecção. Escolha uma tolerância para a aproximação da raiz de 10^{-8}.

2. Documente a sua escolha do intervalo inicial $[a,b]$ para a busca da raiz: quais valores você escolheu para a e b. Por quê?

3. Estude os resultados:

   - Que valor o algoritmo de bisecção retorna para a raiz x?
   - Calcule o valor de $\sqrt{2}$ usando `math.sqrt(2)` e compare-o com a aproximação da raiz. Quão grande é o erro absoluto de x? Como isso se compara com `xtol`?

16.5.3 Localização de raízes usando a função `fsolve`

Um algoritmo (frequentemente) melhor (no sentido de "mais eficiente") do que o algoritmo da bisecção é implementado na função generalizada `fsolve()` para a localização de raízes de funções (multidimensionais). Este algoritmo precisa apenas de um ponto de partida próximo à provável localização da raiz (mas não possui convergência garantida).

Aqui está um exemplo:

```
In [4]: from scipy.optimize import fsolve

        def f(x):
```

```
        return x ** 3 - 2 * x ** 2

    x = fsolve(f, 3)           # one root is at x=2.0

    print("A raiz x é aproximadamente x=%21.19g" % x)
    print("O erro exato é %g." % (2 - x))
```

```
A raiz x é aproximadamente x= 2.000000000000006661
O erro exato é -6.66134e-15.
```

O valor de retorno [6] de `fsolve` é uma matriz numpy de comprimento n para um problema de localização de raízes com n variáveis. No exemplo acima, temos $n = 1$.

16.6 Interpolação

Dado um conjunto de N pontos (x_i, y_i) com $i = 1, 2, \ldots, N$, às vezes precisamos de uma função $\hat{f}(x)$ que retorne $y_i = f(x_i)$ e que, além disso, forneça alguma interpolação dos dados (x_i, y_i) para todo x.

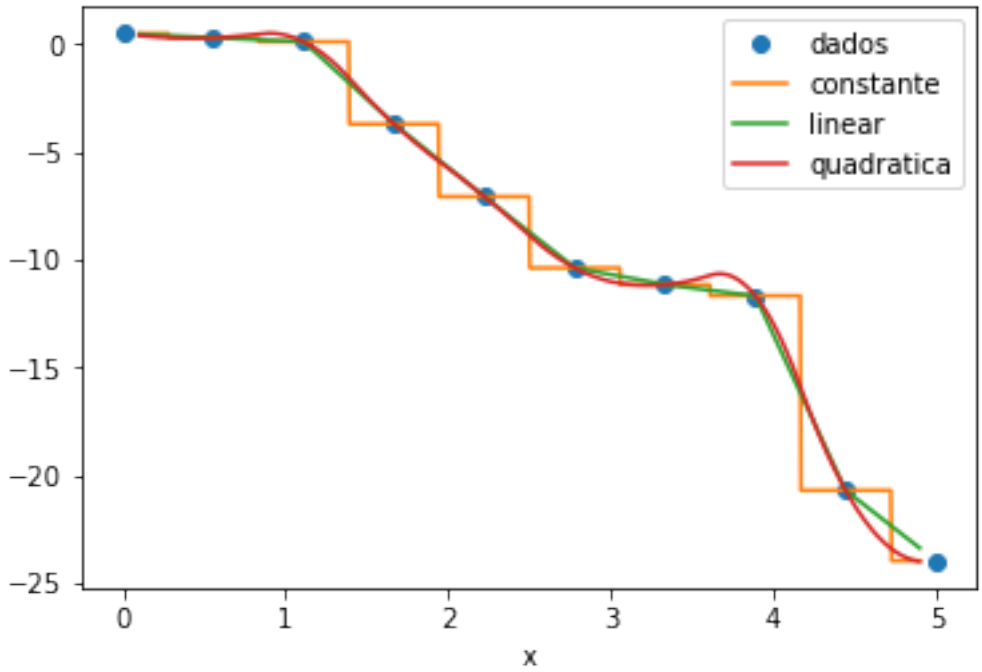
A função `y0 = scipy.interpolate.interp1d (x, y, kind = 'nearest')` realiza esta interpolação baseada em *splines* de ordem variável. Observe que a função `interp1d` retorna *uma função* `y0` que irá então interpolar os dados x-y para qualquer dado x quando chamado como `y0(x)`.

O código abaixo mostra isso, bem como os diferentes tipos de interpolação.

```
In [32]: import numpy as np
         import scipy.interpolate
         import pylab

         def cria_dados(n):
             """Dado um inteiro n, retorne n pontos
             x e valores y como um numpy.array."""
             xmax = 5.
             x = np.linspace(0, xmax, n)
             y = - x**2

             # faça os dados dos pontos x um tanto irregulares
             y += 1.5 * np.random.normal(size=len(x))
             return x, y

         # programa principal
         n = 10
         x, y = cria_dados(n)

         # usa uma malha mais fina e irregular para a plotagem
         xfine = np.linspace(0.1, 4.9, n * 100)

         # interpola com função constante por partes (p=0)
         y0 = scipy.interpolate.interp1d(x, y, kind='nearest')

         # interpola com função linear por partes (p=1)
         y1 = scipy.interpolate.interp1d(x, y, kind='linear')
```

```
# interpola com função quadrática por partes (p=2)
y2 = scipy.interpolate.interp1d(x, y, kind='quadratic')

pylab.plot(x, y, 'o', label='dados')
pylab.plot(xfine, y0(xfine), label='constante')
pylab.plot(xfine, y1(xfine), label='linear')
pylab.plot(xfine, y2(xfine), label='quadrática')
pylab.legend()
pylab.xlabel('x')
```

```
Out[32]: <matplotlib.text.Text at 0x11223be80>
```



16.7 Ajuste de curva

Já vimos que podemos ajustar funções polinomiais a um conjunto de dados usando a função `numpy.polyfit`. Aqui, apresentamos um algoritmo de ajuste de curva mais genérico.

O SciPy fornece uma função um pouco genérica (baseada no algoritmo de Levenberg-Marquardt) através do `scipy.optimize.curve_fit` para ajustar uma determinada função (Python) a um conjunto de dados. O pressuposto é que temos um conjunto de dados com pontos $x_1, x_2, ...x_N$ e com valores de função correspondentes y_i dependentes de x_i, tal que $y_i = f(x_i, \mathbf{p})$. Queremos determinar o vetor de parâmetros $\mathbf{p} = (p_1, p_2, \ldots, p_k)$ para que r, a soma dos resíduos, seja a menor possível:

$$r = \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{p}))^2$$

O ajuste da curva é de uso particular se os dados forem contaminados por ruídos: para um dado x_i e $y_i = f(x_i, \mathbf{p})$, temos um termo de erro (desconhecido) ϵ_i para que $y_i = f(x_i, \mathbf{p}) + \epsilon_i$.

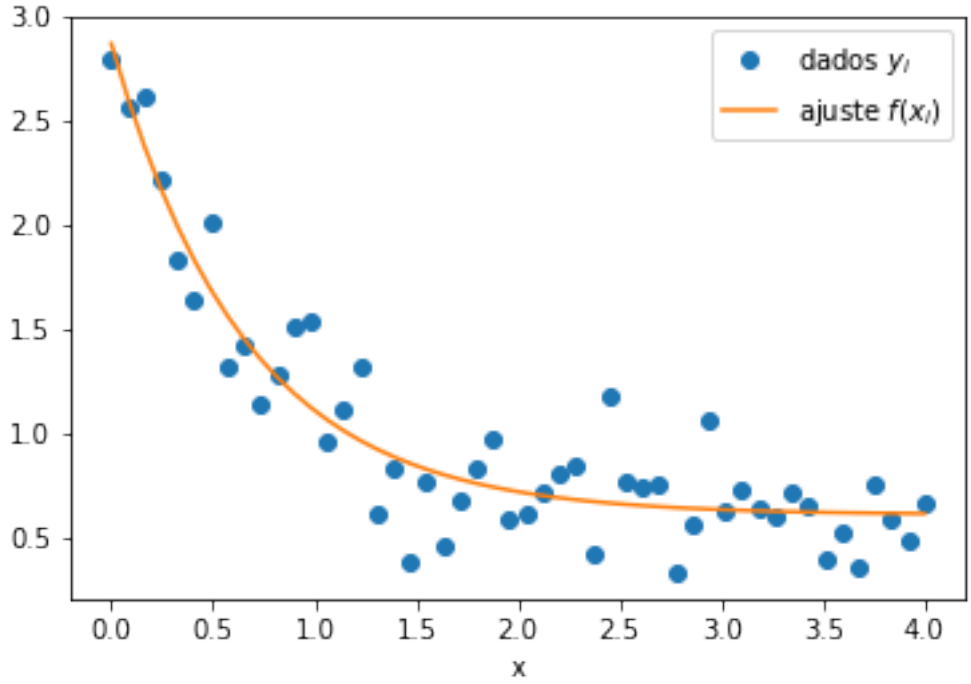
Utilizamos o seguinte exemplo para esclarecer isso:

$$f(x, \mathbf{p}) = a\exp(-bx) + c, \quad \text{i.e.} \quad \mathbf{p} = (a, b, c)$$

```
In [5]: import numpy as np
        from scipy.optimize import curve_fit


        def f(x, a, b, c):
            """Ajusta função  y=f(x,p) com parâmetros p=(a,b,c). """
            return a * np.exp(- b * x) + c

        # cria dados aleatórios
        x = np.linspace(0, 4, 50)
        y = f(x, a=2.5, b=1.3, c=0.5)

        # adiciona ruído
        yi = y + 0.2 * np.random.normal(size=len(x))

        # chama a função de ajuste
        popt, pcov = curve_fit(f, x, yi)
        a, b, c = popt
        print("Os parâmetros ótimos são a=%g, b=%g, and c=%g" % (a, b, c))

        # plotagem
        import pylab
        yfitted = f(x, *popt)   # equivalente a f(x, popt[0], popt[1], popt[2])
        pylab.plot(x, yi, 'o', label='dados $y_i$')
        pylab.plot(x, yfitted, '-', label='ajuste $f(x_i)$')
        pylab.xlabel('x')
        pylab.legend()
```

```
Os parâmetros ótimos são a=2.26288, b=1.50872, and c=0.605159
```

```
Out[5]: <matplotlib.legend.Legend at 0x112abb908>
```



Observe que no código-fonte acima, definimos a função de ajuste $y = f(x)$ através do código Python. Podemos, portanto, ajustar funções (quase) arbitrárias usando o método `curve_fit`.

A função `curve_fit` retorna uma tupla `popt, pcov`. A primeira entrada `popt` contém uma tupla dos parâmetros ótimos (no sentido de que estes minimizam o valor r). A segunda entrada contém a matriz de covariância para todos os parâmetros. As diagonais fornecem a variância dos parâmetro estimados.

Para que o processo de ajuste de curva funcione, o algoritmo de Levenberg-Marquardt precisa iniciar o processo de ajuste com palpites iniciais para os parâmetros finais. Se estes não forem especificados (como no exemplo acima), o valor "1.0"é usado para o palpite inicial.

Se o algoritmo falhar em obter uma função para os dados (mesmo que a função descreva os dados razoavelmente), precisamos dar ao algoritmo melhores estimativas para os parâmetros iniciais. Para o exemplo mostrado acima, podemos dar as estimativas para a função `curve_fit` mudando a linha

```
Popt, pcov = curve_fit (f, x, yi)
```

para

```
Popt, pcov = curve_fit (f, x, yi, p0 = (2,1,0.6))
```

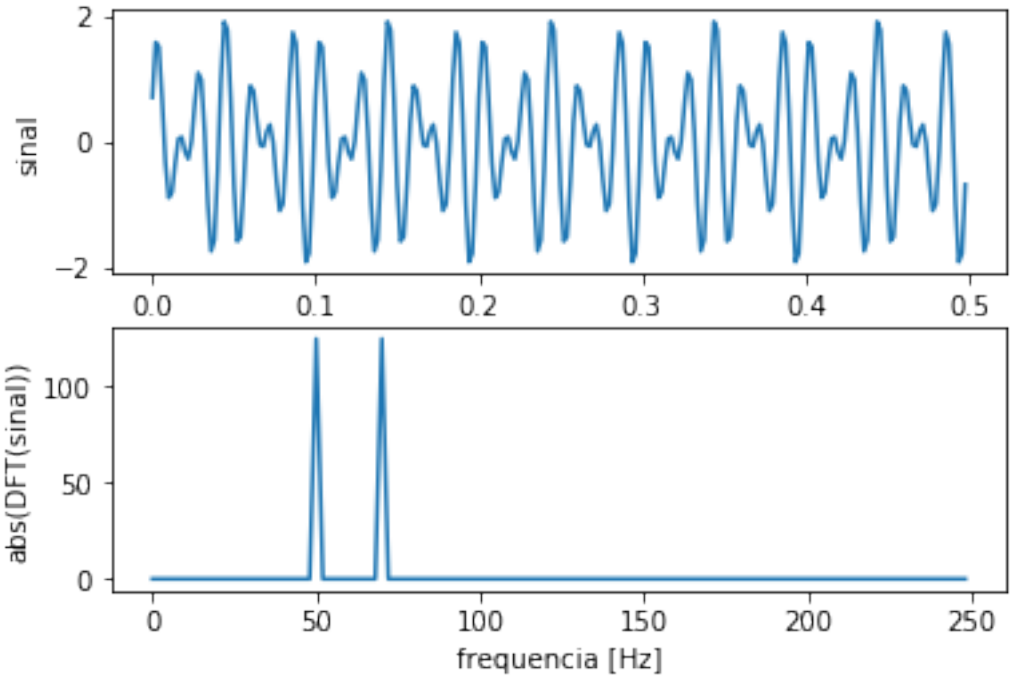
se nossos palpites iniciais fossem $a = 2$,$b = 1$ e $c = 0.6$. Uma vez que tomamos o algoritmo "aproximadamente na área correta"no espaço de parâmetros, o ajuste geralmente funciona bem.

16.8 Transformadas de Fourier

No exemplo seguinte, criamos um sinal como uma superposição de uma onda senoidal de 50 Hz e 70 Hz (com uma ligeira mudança de fase entre eles). Então, aplicamos a transformada de Fourier no sinal e plotamos o valor absoluto dos coeficientes (complexos) da transformada discreta de Fourier pela freqüência. Esperamos ver picos em 50Hz e 70Hz.

In [34]:
```
import scipy
import matplotlib.pyplot as plt
pi = scipy.pi

signal_length = 0.5   # [segundos]
sample_rate=500       # taxa de amostragem [Hz]
dt = 1./sample_rate   # tempo entre as duas amostras[s]

df = 1/signal_length  # frequência entre pontos
                      # no domínio da frequência [Hz]
t=scipy.arange(0,signal_length,dt) # vetor tempo
n_t=len(t)            # comprimento do vetor de tempo

# cria sinal
y=scipy.sin(2*pi*50*t)+scipy.sin(2*pi*70*t+pi/4)

# calcula a transformada de Fourier
f=scipy.fft(y)

# opera sobre as frequências significativas na transformada de Fourier
freqs=df*scipy.arange(0,(n_t-1)/2.,dtype='d') # d = float com precisão dupla
n_freq=len(freqs)

# plota dados de entrada x tempo
plt.subplot(2,1,1)
plt.plot(t,y,label='dados de entrada')
plt.xlabel('tempo [s]')
plt.ylabel('sinal')

# plota espectro de frequências
plt.subplot(2,1,2)
plt.plot(freqs,abs(f[0:n_freq]),
         label='abs(transformada Fourier)')
plt.xlabel('frequência [Hz]')
plt.ylabel('abs(DFT(sinal))')
```

Out[34]:
```
<matplotlib.text.Text at 0x1124df208>
```



O gráfico inferior mostra a transformada discreta de Fourier (DFT) calculada a partir dos dados mostrados no gráfico superior.

16.9 Otimização

Muitas vezes precisamos encontrar o máximo ou mínimo de uma função particular $f(x)$, onde f é uma função escalar, mas x poderia ser um vetor. As aplicações típicas são a minimização de algumas variáveis, tais como custo, risco e erro, ou a maximização de produtividade, eficiência e lucro. Rotinas de otimização normalmente fornecem um método para minimizar uma determinada função: se precisamos maximizar $f(x)$, criamos uma nova função $g(x)$ que inverte o sinal de f, ou seja, $g(x) = -f(x)$ e minimizamos $g(x)$.

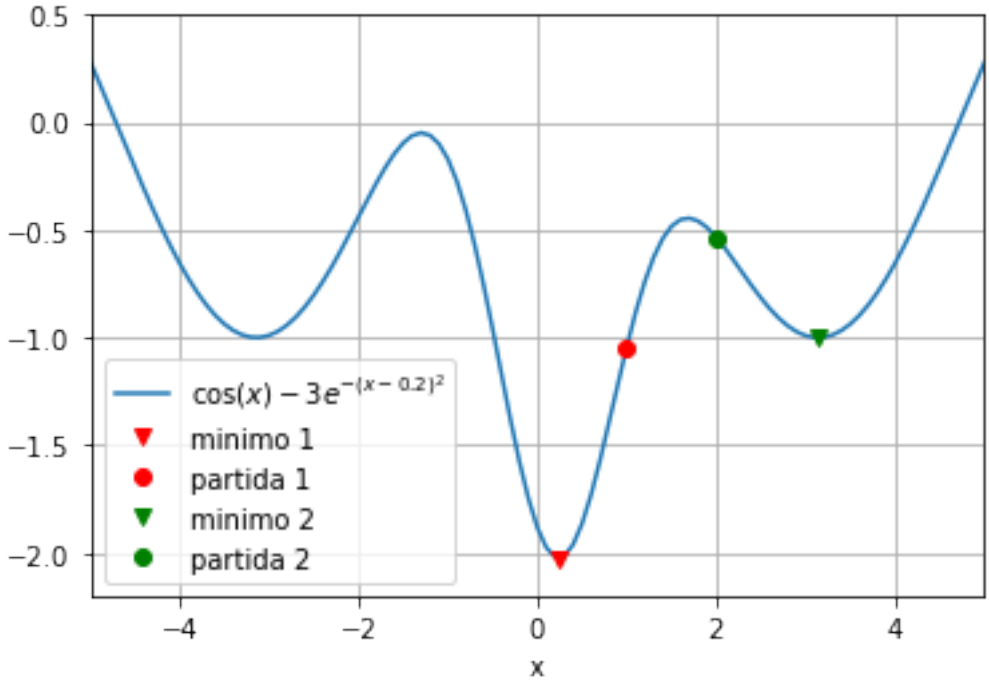
Abaixo, damos um exemplo mostrando (i) a definição da função de teste e (ii) a chamada da função `scipy.optimize.fmin`, que toma como argumento a função f a ser minimizada e um valor inicial x_0 a partir do qual se inicia a busca pelo mínimo, e retorna o valor de x para o qual $f(x)$ é (localmente) minimizado. Normalmente, a busca pelo mínimo é uma busca local, i.e, o algoritmo segue o gradiente local. Nós repetimos a busca pelo mínimo para dois valores ($x_0 = 1.0$ e $x_0 = 2.0$, respectivamente) para demonstrar que, dependendo do valor de partida, podemos encontrar diferentes mínimos para a função f.

A maioria dos comandos (após as duas chamadas de `fmin`) no arquivo `fmin1.py` cria o gráfico da função, os pontos de partida para as buscas e o mínimo obtido:

```
In [6]: from scipy import arange, cos, exp
        from scipy.optimize import fmin
        import pylab

        def f(x):
            return cos(x) - 3 * exp( -(x - 0.2) ** 2)
```

```
        # encontra mínimos de f(x),
        # começa de 1.0 e 2.0 respectivamente
        minimum1 = fmin(f, 1.0)
        print("Busca iniciada em x=1., minimo é", minimum1)
        minimum2 = fmin(f, 2.0)
        print("Busca iniciada em x=2., minimo é", minimum2)

        # plota função
        x = arange(-10, 10, 0.1)
        y = f(x)
        pylab.plot(x, y, label='$\cos(x)-3e^{-(x-0.2)^2}$')
        pylab.xlabel('x')
        pylab.grid()
        pylab.axis([-5, 5, -2.2, 0.5])

        # adiciona minimo1 para plot
        pylab.plot(minimum1, f(minimum1), 'vr',
                   label='minimo 1')
        # adiciona ponto de partida 1 para plot
        pylab.plot(1.0, f(1.0), 'or', label='partida 1')

        # adiciona minimo2 para plot
        pylab.plot(minimum2,f(minimum2),'vg',\
                   label='minimo 2')

        # adiciona ponto de partida 2 para plot
        pylab.plot(2.0,f(2.0),'og',label='partida 2')

        pylab.legend(loc='lower left')
```

```
Optimization terminated successfully.
         Current function value: -2.023866
         Iterations: 16
         Function evaluations: 32
Busca iniciada em x=1., minimo é [ 0.23964844]
Optimization terminated successfully.
         Current function value: -1.000529
         Iterations: 16
         Function evaluations: 32
Busca iniciada em x=2., minimo é [ 3.13847656]


Out[6]: <matplotlib.legend.Legend at 0x112c5f828>
```



A chamada da função `fmin` produzirá algum diagnóstico de saída, como você pode ver acima.

Valor de retorno de `fmin` Note que o valor de retorno da função `fmin` é um `array` do numpy que - para o exemplo acima - contém apenas um número já que temos um único parâmetro (aqui x) para variar. Em geral, `fmin` pode ser usada para encontrar o mínimo em um espaço de parâmetros de dimensão superior se houver vários parâmetros. Nesse caso, o `array` numpy conteria aqueles parâmetros que minimizam a função objetivo. A função objetivo $f(x)$ tem que retornar um escalar mesmo que haja mais parâmetros, ou seja, mesmo se x for um vetor.

16.10 Outros métodos numéricos

SciPy e NumPy proporcionam o acesso a um grande número de outros algoritmos numéricos, incluindo interpolação, transformadas de Fourier, otimização, funções especiais (tais como funções de Bessel), processamento de sinais e filtros, geração de números aleatórios, e mais. Você pode começar a explorar as capacidades desses dois módulos usando a função `help` ou a documentação fornecida na web.

16.11 scipy.io: entrada e saída no SciPy

O SciPy fornece rotinas para ler e escrever arquivos `.mat` do Matlab. Aqui está um exemplo de criação de um arquivo compatível com Matlab armazenando uma matriz (1x11), e posterior leitura desses dados em uma matriz numpy do Python usando a biblioteca de entrada e saída do SciPy:

Primeiro criamos um arquivo `.mat` no Octave (Octave é, na maioria dos casos, compatível com Matlab):

```
octave: 1> a = -1: 0,5: 4
a =
Columns 1 through 6:
```

```
    -1,0000 -0,5000 0,0000 0,5000 1,0000 1,5000
Columns 7 through 11:
    2,0000 2,5000 3,0000 3,5000 4,0000
octave: 2> save -6% octave_a.mat a     % salva como versão 6
```

Então, carregamos este *array* no Python:

```
In [36]: from scipy.io import loadmat
         mat_contents = loadmat('static/data/octave_a.mat')

In [37]: mat_contents

Out[37]: {'__globals__': [],
          '__header__': b'MATLAB 5.0 MAT-file, written by Octave 4.0.3, 2017-06-29 01:52:49
          '__version__': '1.0',
          'a': array([[-1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ]])}

In [38]: mat_contents['a']

Out[38]: array([[-1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ]])
```

A função `loadmat` retorna um dicionário: a chave (*key*) para cada item no dicionário é uma *string* cujo nome é o mesmo da matriz que foi salva no Matlab.

Um arquivo Matlab pode conter vários *arrays*. Cada um deles é apresentado por um par chave-valor (*key:value*) no dicionário.

Vamos salvar dois *arrays* a partir do Python para demonstrar isso:

```
In [39]: import scipy.io
         import numpy as np

         # cria dois arrays numpy
         a = np.linspace(0, 50, 11)
         b = np.ones((4, 4))

         # salva como arquivo mat
         # cria dicionario para savemat
         tmp_d = {'a': a,
                  'b': b}
         scipy.io.savemat('static/data/a_b.mat', tmp_d)
```

This program creates the file `a_b.mat`, which we can subsequently read using Matlab or here Octave:

Este programa cria o arquivo `a_b.mat`, que podemos subsequentemente ler usando o Matlab ou

```
HAL47:code fangohr$ octave
GNU Octave, version 3.2.4
Copyright (C) 2009 John W. Eaton and others.
<snip>

octave:1> whos
Variables in the current scope:
```

```
  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       ans         1x11                        92  cell

Total is 11 elements using 92 bytes

octave:2> load data.mat
octave:3> whos
Variables in the current scope:

  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       a          11x1                        88  double
       ans         1x11                       92  cell
       b           4x4                       128  double

Total is 38 elements using 308 bytes

octave:4> a
a =

    0
    5
   10
   15
   20
   25
   30
   35
   40
   45
   50

octave:5> b
b =

   1   1   1   1
   1   1   1   1
   1   1   1   1
   1   1   1   1
```

Note que há outras funções para leitura/escrita em formatos usados por IDL, Netcdf e outros na `scipy.io`.

Para saber mais veja Tutorial Scipy.

17 Para onde ir a partir daqui?

Aprender uma linguagem de programação é o primeiro passo para tornar-se um "computacionalista"que queira fazer avançar a ciência e a engenharia através de modelagem computacional e simulação.

Enumeramos algumas habilidades adicionais que podem ser muito benéficas para o trabalho diário do cientista computacional, mas é claro que elas não são exaustivas.

17.1 Programação avançada

Este texto enfatizou a formação de uma base sólida em termos de programação, abrangendo fluxo de controle, estruturas de dados e elementos da programação procedural e funcional. Não falamos de Orientação a Objetos em grande detalhe, nem discutimos alguns dos recursos mais avançados do Python, como iteradores e decoradores, por exemplo.

17.2 Linguagem de programação compilada

Quando o desempenho começa a ser a prioridade máxima, talvez seja necessário usar código compilado, e provavelmente incorporá-lo em um código Python para realizar os cálculo nas partes onde há deficiência de desempenho (*bottlenecks*).

Fortran, C e C++ são escolhas sensíveis por enquanto. Talvez Julia num futuro próximo.

Também precisamos aprender a integrar código compilado com Python usando ferramentas como Cython, Boost, Ctypes e Swig.

17.3 Teste

Uma boa codificação é suportada por uma série de testes de unidades e sistemas que podem ser executados rotineiramente para verificar se o código funciona corretamente. Ferramentas como `doctest`, `nose` e `pytest` são inestimáveis, e devemos aprender, pelo menos, como usar o `pytest` (ou o `nose`).

17.4 Modelos de simulação

Uma série de ferramentas-padrão para simulações, como Monte Carlo, Dinâmica Molecular, modelos baseados em *lattice*, diferenças finitas e elementos finitos, são comumente usados para resolver desafios específicos de simulação - é útil ter pelo menos uma visão geral deles.

17.5 Engenharia de software para códigos de pesquisa

Os códigos de pesquisa trazem desafios particulares: os requisitos podem mudar durante o tempo de execução do projeto. Precisamos de grande flexibilidade e reprodutibilidade. Uma série de técnicas estão disponíveis para dar suporte de forma eficaz.

17.6 Dados e visualização

Lidar com grandes quantidades de dados, processá-los e visualizá-los pode ser um desafio. Conhecimento fundamental de projeto de banco de dados, visualização 3D e ferramentas modernas de processamento de dados, como o pacote Pandas do Python, ajudam com isso.

17.7 Controle de versão

O uso de uma ferramenta de controle de versão, como git ou mercurial, deve ser uma abordagem padrão e melhora significativamente a eficácia da escrita de código, ajuda com o trabalho em equipe e - talvez o mais importante - suporta a reprodutibilidade de resultados computacionais.

17.8 Execução paralela

A execução paralela de códigos é uma maneira de tornar códigos em grande escala mais ágeis. Isso pode ser pelo uso de MPI para a comunicação *inter-node* ou OpenMP para a paralelização *intra-node*, ou um modo híbrido que integre ambos.

O recente aumento da computação GPU fornece ainda outra via de paralelização, e também os processadores multinúcleo, como o Intel Phi.

17.8.1 Agradecimentos

- Marc Molinari, por revisar este manuscrito em 2007.
- Neil O'Brien, por contribuir para a seção SymPy.
- Jacek Generowicz, por me apresentar o Python no último milênio, e por gentilmente compartilhar inúmeras idéias de seu curso de Python.
- EPSRC, pelo suporte parcial deste trabalho (GR/T09156/01 e EP/G03690X/1).
- Estudantes e outros leitores que comentaram e/ou apontaram erros de digitação, etc.

[1] a linha vertical é mostrar apenas a divisão entre os componentes originais apenas; matematicamente, a matriz aumentada comporta-se como qualquer outra matriz 2 Œ 3, e nós a codificamos no SymPy como faríamos com qualquer outra.

[2] a partir da documentação `help(preview)`: "Atualmente isso depende do pexpect, que não está disponível para Windows".

[3] O valor exato para o limite superior está disponível em `sys.maxint`.

[4] Adicionamos, por completeza, que um programa C (ou C++/Fortran) que execute o mesmo laço será cerca de 100 vezes mais rápido do que o laço `float` do Python e, portanto, cerca de 100*200 = 20000 vezes mais rápido que o laço simbólico.

[5] Neste texto, geralmente fazemos a importação do `numpy` com o nome `N` assim: `import numpy as N`. Se você não tiver `numpy` na sua máquina, você pode substituir esta linha por `import Numeric as N`, ou `import numarray as N`.

[6] Nota histórica: isso mudou da versão Scipy 0.7 para 0.8. Antes da 0,8, o valor de retorno era um `float` se um problema unidimensional fosse resolvido.