

---

# **Python Frequently Asked Questions**

***Release 2.7.18***

**Guido van Rossum  
and the Python development team**

**maio 07, 2020**

**Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)**



---

## Sumário

---

<b>1</b>	<b>Python Geral</b>	<b>1</b>
<b>2</b>	<b>FAQ referente a Programação</b>	<b>9</b>
<b>3</b>	<b>Design e Histórico FAQ</b>	<b>41</b>
<b>4</b>	<b>FAQ de Bibliotecas e Extensões</b>	<b>55</b>
<b>5</b>	<b>FAQ sobre Extensão/Incorporação</b>	<b>69</b>
<b>6</b>	<b>Python no Windows</b>	<b>77</b>
<b>7</b>	<b>FAQ da Interface Gráfica do Usuário</b>	<b>83</b>
<b>8</b>	<b>FAD de “Por que o Python está instalado em meu computador?”</b>	<b>87</b>
<b>A</b>	<b>Glossário</b>	<b>89</b>
<b>B</b>	<b>Sobre esses documentos</b>	<b>99</b>
<b>C</b>	<b>História e Licença</b>	<b>101</b>
<b>D</b>	<b>Copyright</b>	<b>117</b>
	<b>Índice</b>	<b>119</b>



## 1.1 Informações gerais

### 1.1.1 O que é Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

Para saber mais, inicie pelo nosso tutorial [tutorial-index](#). Os links do [Beginner's Guide to Python](#) para outros tutoriais introdutórios e recursos da linguagem Python.

### 1.1.2 O que é a Python Software Foundation?

O Python Software Foundation é uma organização independente e sem fins lucrativos que detém os direitos autorais sobre as versões 2.1 do Python e as mais recentes. A missão do PSF é avançar a tecnologia de código aberto relacionada à linguagem de programação Python e divulgar a utilização do Python. A página inicial do PSF pode ser acessada pelo link a seguir <https://www.python.org/psf/>.

Doações para o PSF estão isentas de impostos nos EUA. Se utilizares o Python e achares útil, contribua através da [página de doação da PSF](#).

### 1.1.3 Existem restrições de direitos autorais sobre o uso de Python?

Podemos fazer tudo o que quisermos com os fontes, desde que deixemos os direitos autorais e exibamos esses direitos em qualquer documentação sobre o Python que produzirmos. Se honrarmos as regras dos direitos autorais, não há quaisquer problema em utilizar o Python em versões comerciais, vendê-lo, copiá-lo na forma de código-fonte ou o seu binária (modificado ou não modificado), ou para vender produtos que incorporem o Python de alguma forma. Ainda gostaríamos de saber sobre todo o uso comercial de Python, é claro.

Veja a [página da licença PSF](#) para encontrar mais explicações e um link para o texto completo da licença.

O logotipo do Python é marca registrada e, em certos casos, é necessária permissão para usá-la. Consulte a [Política de Uso da Marca comercial](#) para obter mais informações.

### 1.1.4 Em primeiro lugar, por que o Python foi desenvolvido?

Aqui está um resumo *muito* breve de como que tudo começou, escrito por Guido van Rossum:

Eu tive vasta experiência na implementação de linguagens interpretada no grupo ABC da CWI e, ao trabalhar com esse grupo, aprendi muito sobre o design de linguagens. Esta é a origem de muitos recursos do Python, incluindo o uso do recuo para o agrupamento de instruções e a inclusão de tipos de dados de alto nível (embora existam diversos detalhes diferentes em Python).

Eu tinha uma série de queixas sobre a linguagem ABC, mas também havia gostado de muitos das suas características. Era impossível estender ABC (ou melhorar a implementação) para remediar minhas queixas – na verdade, a falta de extensibilidade era um dos maiores problemas. Eu tinha alguma experiência com o uso de Modula-2+ e conversei com os designers do Modula-3 e li o relatório do Modula-3. Modula-3 foi a origem da sintaxe e semântica usada nas exceções, e alguns outros recursos do Python.

Eu estava trabalhando no grupo de sistema operacional distribuído da Amoeba na CWI. Precisávamos de uma maneira melhor de administrar o sistema do que escrevendo programas em C ou scripts para a shell Bourne, uma vez que o Amoeba tinha a sua própria interface de chamada do sistema, que não era facilmente acessível a partir do shell Bourne. Minha experiência com o tratamento de erros em Amoeba me conscientizou da importância das exceções como um recurso das linguagens de programação.

Percebi que uma linguagem de script com uma sintaxe semelhante a do ABC, mas com acesso às chamadas do sistema Amoeba, preencheria a necessidade. Percebi também que seria uma boa escrever uma linguagem específica para o Amoeba, então, decidi que precisava de uma linguagem realmente extensível.

Durante as férias do Natal de 1989, tive bastante tempo disponível e então decidi tentar a construção de algo. Durante o ano seguinte, continuei trabalhando em minhas horas vagas, e o Python foi usado no projeto Amoeba com crescente sucesso, e o feedback dos colegas me fez implementar muitas melhorias.

Em fevereiro de 1991, depois de mais de um ano de desenvolvimento, decidi publicar na USENET. O resto está no arquivo `Misc/HISTORY`.

### 1.1.5 Para o que Python é excelente?

Python é uma linguagem de programação de propósito geral, de alto nível e que pode ser aplicada em muitos tipos diferentes de problemas.

A linguagem vem com uma larga biblioteca padrão que cobre áreas como processamento de cadeias de caracteres (expressões regulares, Unicode, cálculo de diferença entre arquivos), protocolos da Internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, programação CGI), engenharia de software (testes unitários, logging, análise de desempenho, parsing de código Python), e interfaces do sistema operacional (chamadas de sistema, sistemas de arquivos, sockets TCP/IP). Veja a tabela de conteúdo [library-index](#) para ter uma ideia do que está disponível. Uma grande variedade de extensões de terceiros também está disponível. Consulte o [Índice de Pacotes Python](#) para encontrar pacotes que possam interessar a você.

### 1.1.6 Como funciona o esquema de numeração de versões do Python?

As versões do Python são numeradas na forma A.B.C ou A.B. A é o número da versão principal – só é incrementado quando ocorre mudanças realmente importantes na linguagem. O B é o número da versão menor, incrementado quando ocorre mudanças menores. O C é o micro-nível – é incrementado para cada versão de correções de erros. Veja a [PEP 6](#) para obter mais informações sobre as versões de correções de erros.

Nem todos as versões são lançamentos de correções de erros. Na corrida por uma nova versão principal, uma série de lançamentos de desenvolvimento são feitas, denotadas como alfa, beta ou candidata. As versões alfas são lançamentos iniciais (early releases) em que as interfaces ainda não estão finalizadas; não é inesperado ver uma mudança de interface entre duas versões alfa. As betas são mais estáveis, preservando as interfaces existentes, mas possivelmente adicionando novos módulos, e as candidatas a lançamento são congeladas, sem alterações, exceto quando necessário para corrigir erros críticos.

As versões de lançamento alfa, beta e candidata possuem um sufixo adicional. O sufixo para uma versão alfa é “aN” para algum número pequeno N, o sufixo para uma versão beta é “bN” para algum número pequeno N e o sufixo para uma versão candidata a ser lançada é “cN” para algum pequeno número N. Em outras palavras, todas as versões rotuladas como 2.0aN precedem as versões rotuladas como 2.0bN, que precedem as versões rotuladas como 2.0cN e \* essas\* precederam a 2.0.

Também podemos encontrar números de versão com um sufixo “+”, por exemplo, “2.2+”. Estas são versões não lançadas, construídas diretamente do repositório de desenvolvimento do CPython. Na prática, após uma última versão menor, a versão é incrementada para a próxima versão secundária, que se torna a versão “a0”, por exemplo, “2.4a0”.

Veja também a documentação para `sys.version`, `sys.hexversion`, e `sys.version_info`.

### 1.1.7 Como faço para obter uma cópia dos fonte do Python?

A última distribuição fonte do Python sempre está disponível no [python.org](https://www.python.org/downloads/), em <https://www.python.org/downloads/>. As últimas fontes de desenvolvimento podem ser obtidas em <https://github.com/python/cpython/>.

A distribuição fonte é um arquivo .tar com .gzip contendo o código-fonte C completo, a documentação formatada com o Sphinx, módulos de biblioteca Python, programas de exemplo e várias peças úteis de software livremente distribuível. A fonte compilará e executará sem a necessidade de configurações extras na maioria das plataformas UNIX.

Consult the [Getting Started section of the Python Developer's Guide](#) for more information on getting the source code and compiling it.

### 1.1.8 Como faço para obter a documentação do Python?

A documentação padrão para a versão atualmente estável do Python está disponível em <https://docs.python.org/3/>. Em PDF, texto simples e versões HTML para download também estão disponíveis em <https://docs.python.org/3/download.html>.

A documentação é escrita em reStructuredText e processada pela [ferramenta de documentação Sphinx](#). Os fonte do reStructuredText para documentação fazem parte da distribuição fonte do Python.

### 1.1.9 Eu nunca programei antes. Existe um tutorial básico do Python?

Existem inúmeros tutoriais e livros disponíveis. A documentação padrão inclui `tutorial-index`.

Consulte o [Guia do Iniciante](#) para encontrar informações para quem está começando agora na programação Python, incluindo uma lista com tutoriais.

### 1.1.10 Existe um grupo de discussão ou lista de discussão dedicada ao Python?

Existe um grupo de notícias `comp.lang.python`, e uma lista de discussão, `python-list`. O grupo notícias e a lista de discussão são conectados um ou outro – se poderes ler as notícias, não será necessário se inscrever na lista de discussão. `comp.lang.python` possui bastante postagem, recebendo centenas de postagens todos os dias, e os leitores do Usenet geralmente são mais capazes de lidar com esse volume.

Os anúncios de novas versões do software e eventos podem ser encontrados em `comp.lang.python.announce`, uma lista moderada de baixo tráfego que recebe cerca de cinco postagens por dia. Está disponível como a [lista de discussão python-announce](#).

Mais informações sobre outras listas de discussão e grupos de notícias podem ser encontradas em <https://www.python.org/community/lists/>.

### 1.1.11 Como faço para obter uma versão de teste beta do Python?

As versões alfa e beta estão disponíveis em <https://www.python.org/downloads/>. Todos os lançamentos são anunciados nos grupos de notícias `comp.lang.python` e `comp.lang.python.announce` e na página inicial do Python em <https://www.python.org/>; um feed RSS de notícias está disponível.

You can also access the development version of Python through Git. See [The Python Developer's Guide](#) for details.

### 1.1.12 Como eu envio um relatório de erros e correções para o Python?

Para reportar um erro ou enviar uma correção, use o Roundup rodando em <https://bugs.python.org/>.

Você deve ter uma conta no Roundup para poder enviar um relatório de erro; isso nos permite contactá-lo se tivermos mais perguntas a serem feitas. Também permitirá que o Roundup lhe envie atualizações à medida que trabalhamos na correção do erro por você relatado. Se tiveres usado o SourceForge anteriormente para relatar erros do Python, você pode obter sua senha do Roundup através do [procedimento para recuperar senha](#).

For more information on how Python is developed, consult [the Python Developer's Guide](#).

### 1.1.13 Existem alguns artigos publicados sobre o Python para que eu possa fazer referência?

Provavelmente será melhor citar o seu livro favorito sobre o Python.

O primeiro artigo sobre Python foi escrito em 1991 e atualmente o mesmo se encontra bastante desatualizado.

Guido van Rossum e Jelke de Boer, “Interactively Testing Remote Servers Using the Python Programming Language”, CWI Quarterly, Volume 4, Edição 4 (dezembro de 1991), Amsterdam, pp. 283–303.q



### 1.1.14 Existem alguns livros sobre o Python?

Sim, há muitos publicados e muitos outros que estão sendo nesse momento escritos!! Veja o wiki python.org em <https://wiki.python.org/moin/PythonBooks> para obter uma listagem.

Você também pode pesquisar livrarias online sobre “Python” e filtrar as referências a respeito do Monty Python; ou talvez procure por “Python” e “linguagem”.

### 1.1.15 Onde está armazenado o site [www.python.org](http://www.python.org)?

A infraestrutura do projeto Python está localizada em todo o mundo e é gerenciada pela equipe de infraestrutura do Python. Detalhes [aqui](#).

### 1.1.16 Por que o nome Python?

Quando o Guido van Rossum começou a implementar o Python, o mesmo também estava lendo os scripts publicados do “Monty Python’s Flying Circus”, uma série de comédia da BBC da década de 1970. Van Rossum pensou que precisava de um nome curto, único e ligeiramente misterioso, então resolveu chamar a sua linguagem de Python.

### 1.1.17 Eu tenho que gostar de “Monty Python’s Flying Circus”?

Não, mas isso ajuda. :)

## 1.2 Python no mundo real

### 1.2.1 Quão estável é o Python?

Muito estável. Novos lançamentos estáveis são divulgados aproximadamente de 6 a 18 meses desde 1991, e isso provavelmente continuará. Atualmente, o ritmo de publicação vem sendo a cada 18 meses das versões principais.

Os desenvolvedores lançam versões “bugfix” de versões mais antigas, então a estabilidade dos lançamentos existentes melhora gradualmente. As liberações de correções de erros, indicadas por um terceiro componente do número da versão (por exemplo, 3.5.3, 3.6.2), são gerenciadas para estabilidade; somente correções para problemas conhecidos são incluídas em uma versão de correções de erros, e é garantido que as interfaces permanecerão as mesmas durante uma série de liberações de correções de erros.

As últimas versões estáveis sempre podem ser encontradas na página de download do Python <<https://www.python.org/downloads/>>‘\_. Existem duas versões prontas para produção do Python: 2.x e 3.x. A versão recomendada é 3.x, que é suportada pelas bibliotecas mais usadas. Embora 2.x ainda seja amplamente utilizado, *não será mantido após 1 de janeiro de 2020*.

### 1.2.2 Quantas pessoas usam o Python?

Provavelmente existem dezenas de milhares de usuários, embora seja difícil obter uma contagem exata.

O Python está disponível para download gratuito, portanto, não há números de vendas, e o mesmo está disponível em vários diferentes sites e é empacotado em muitas distribuições Linux, portanto, utilizar as estatísticas de downloads não seria a melhor forma para contabilizarmos a base de usuários.

O grupo de notícias comp.lang.python é bastante ativo, mas nem todos os usuários Python postam no grupo ou mesmo o leem regularmente.

### 1.2.3 Existe algum projeto significativo feito em Python?

Veja a lista em <https://www.python.org/about/success> para obter uma listagem de projetos que usam o Python. Consultar as [conferências passadas do Python](#) revelará as contribuições de várias empresas e de diferentes organizações.

Os projetos Python de alto perfil incluem o [gerenciador de lista de e-mail Mailman](#) e o [servidor de aplicativos Zope](#). Várias distribuições Linux, mais notavelmente o [Red Hat](#), escreveram parte ou a totalidade dos seus instaladores e software de administração do sistema em Python. Empresas que usam Python internamente incluem Google, Yahoo e Lucasfilm Ltd.

### 1.2.4 Quais são os novos desenvolvimentos esperados para o Python no futuro?

Consulte <https://www.python.org/dev/peps/> para ver a lista de propostas de aprimoramento do python (PEPs). As PEPs são documentos de design que descrevem novos recursos que foram sugeridos para o Python, fornecendo uma especificação técnica concisa e a sua lógica. Procure uma PEP intitulado de “Python X.Y Release Schedule”, onde X.Y é uma versão que ainda não foi lançada publicamente.

Novos desenvolvimentos são discutidos na [lista de discussão python-dev](#).

### 1.2.5 É razoável propor mudanças incompatíveis com o Python?

Normalmente não. Já existem milhões de linhas de código Python em todo o mundo, de modo que qualquer alteração na linguagem que invalide mais de uma fração muito pequena dos programas existentes deverá ser desaprovada. Mesmo que possamos fornecer um programa de conversão, ainda haverá o problema de atualizar toda a documentação; muitos livros foram escritos sobre o Python, e não queremos invalidá-los todos de uma vez só.

Fornecer um caminho de atualização gradual será necessário se um recurso precisar ser alterado. A [PEP 5](#) descreve o procedimento e em seguida introduz alterações incompatíveis com versões anteriores ao mesmo tempo em que minimiza a interrupção dos usuários.

### 1.2.6 O Python é uma boa linguagem para quem está começando na programação agora?

Sim.

Ainda é bastante comum que os alunos iniciem com uma linguagem procedimental e estaticamente tipada como Pascal e o C ou um subconjunto do C++ ou do Java. Os alunos podem ser melhor atendidos ao aprender Python como sua primeira linguagem. Python possui uma sintaxe muito simples e consistente e uma grande quantidade de bibliotecas padrão e, o mais importante, o uso do Python em um curso de programação para iniciantes permite aos alunos se concentrarem em habilidades de programação importantes, como a decomposição do problema e o design do tipo de dados. Com Python, os alunos podem ser introduzidos rapidamente em conceitos básicos, como repetições e procedimentos. Provavelmente os mesmos até poderão trabalhar com objetos definidos por ele mesmos logo em seu primeiro curso.

Para um aluno que nunca programou antes, usar uma linguagem estaticamente tipada parece não que não é natural. Isso apresenta uma complexidade adicional que o aluno deverá dominar e geralmente retarda o ritmo do curso. Os alunos estão tentando aprender a pensar como um computador, decompor problemas, projetar interfaces consistentes e encapsular dados. Embora aprender a usar uma linguagem tipicamente estática seja importante a longo prazo, não é necessariamente o melhor tópico a ser abordado no primeiro momento de um curso de programação.

Muitos outros aspectos do Python fazem do mesmo uma excelente linguagem para quem está aprendendo a programar. Como Java, Python possui uma biblioteca padrão grande para que os estudantes possam receber projetos de programação muito cedo no curso e que possam *fazer* trabalhos úteis. As atribuições não estão restritas à calculadora padrão de quatro funções e os programas para verificar o peso. Ao usar a biblioteca padrão, os alunos podem ter a satisfação de trabalhar em aplicações reais à medida que aprendem os fundamentos da programação. O uso da biblioteca padrão também ensina os alunos sobre a reutilização de código. Os módulos de terceiros, como o PyGame, também são úteis para ampliar o alcance dos estudantes.

O interpretador interativo do Python permite aos alunos testarem recursos da linguagem enquanto estão programando. Os mesmos podem manter uma janela com o interpretador executado enquanto digitam o fonte do seu programa numa outra janela. Se eles não conseguirem se lembrar dos métodos de uma lista, eles podem fazer algo assim:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getslice__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
 '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
 'pop', 'remove', 'reverse', 'sort']
>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -- append object to end

>>> L.append(1)
>>> L
[1]
```

With the interpreter, documentation is never far from the student as he's programming.

Há também boas IDEs para o Python. O IDLE é uma IDE multiplataforma para o Python e que foi escrito em Python usando o Tkinter. O PythonWin é uma IDE específica para o Windows. Os usuários do Emacs estarão felizes em saber que existe um ótimo modo Python para Emacs. Todos esses ambientes de programação fornecem destaque de sintaxe, recuo automático e acesso ao interpretador interativo durante o tempo de desenvolvimento. Consulte [o wiki do Python](#) para obter uma lista completa dos ambientes de desenvolvimento para o Python.

Se você quiser discutir o uso do Python na educação, poderás estar interessado em se juntar à [lista de discussão edu-sig](#).

## 1.3 Upgrading Python

### 1.3.1 What is this bsddb185 module my application keeps complaining about?

Starting with Python2.3, the distribution includes the *PyBSDDB package* <<http://pybsddb.sf.net/>> as a replacement for the old bsddb module. It includes functions which provide backward compatibility at the API level, but requires a newer version of the underlying *Berkeley DB* library. Files created with the older bsddb module can't be opened directly using the new module.

Using your old version of Python and a pair of scripts which are part of Python 2.3 (db2pickle.py and pickle2db.py, in the Tools/scripts directory) you can convert your old database files to the new format. Using your old Python version, run the db2pickle.py script to convert it to a pickle, e.g.:

```
python2.2 <pathto>/db2pickle.py database.db database.pck
```

Rename your database file:

```
mv database.db olddatabase.db
```

Now convert the pickle file to a new format database:

```
python <pathto>/pickle2db.py database.db database.pck
```

The precise commands you use will vary depending on the particulars of your installation. For full details about operation of these two scripts check the doc string at the start of each one.

---

### FAQ referente a Programação

---

## 2.1 Questões Gerais

### 2.1.1 Existe um depurador a nível de código-fonte que possui breakpoints, single-stepping e etc.?

Sim.

O módulo `pdb` é um depurador cujo funcionamento ocorre em modo Console simples mas, adequado para o Python. Faz parte da biblioteca padrão do Python e está documentado em [documentado in the Library Reference Manual](#) 1. Caso necessário, também é possível a construção do seu próprio depurador usando o código do `pdb` como um exemplo.

O Ambiente de Desenvolvimento Interativo IDLE, que faz parte da distribuição padrão do Python (normalmente disponível em `Tools/scripts/idle`), inclui um depurador gráfico.

O PythonWin é uma IDE feita para o Python que inclui um depurador de GUI baseado no `pdb`. O depurador Pythonwin colora os pontos de interrupção e tem alguns recursos legais, como a depuração de programas que não são Pythonwin. O Pythonwin está disponível como parte do projeto [Python for Windows Extensions](#) e como parte da distribuição ActivePython (veja <https://www.activestate.com/activepython/>).

O [Boa Constructor](#) é uma IDE e GUI que usa `wxWidgets`. Oferece criação e manipulação de quadros visualmente, um inspetor de objetos, muitas visualizações do fonte, como navegadores de objetos, hierarquias de herança, documentação HTML gerada por uma sequência de documentos, um depurador avançado, ajuda integrada e suporte ao Zope.

O [Eric](#) é uma IDE construída com o `PyQt` e fazendo uso do componente de edição Scintilla.

`Pydb` é uma versão do `pdb` padrão do depurador Python, modificado para uso com o DDD (depurador de exibição de dados), um front-end popular do depurador gráfico. `Pydb` pode ser encontrado em <http://bashdb.sourceforge.net/pydb/> e DDD pode ser encontrado em <https://www.gnu.org/software/ddd>.

Há uma série de IDE comerciais para desenvolvimento com o Python que incluem depuradores gráficos. Dentre tantas temos:

- Wing IDE (<https://wingware.com/>)

- Komodo IDE (<https://komodoide.com/>)
- PyCharm (<https://www.jetbrains.com/pycharm/>)

### 2.1.2 Existe uma ferramenta que ajuda na detecção de bugs ou a realizar análises estáticas?

Sim.

O PyChecker é uma ferramenta de análise estática que encontra erros no código-fonte do Python e exibe avisos sobre a complexidade e a estilização do código. Você pode obter PyChecker em <http://pychecker.sourceforge.net/>.

O Pylint é outra ferramenta que verifica se um módulo satisfaz um padrão de codificação e também permite escrever plug-ins para adicionar um recurso personalizado. Além da verificação de erros que o PyChecker executa, o Pylint oferece alguns recursos adicionais, como a verificação do comprimento da linha, se os nomes das variáveis estão bem formados e de acordo com padrão internacional de codificação, se as interfaces declaradas foram totalmente implementadas e muito mais. O <https://docs.pylint.org/> fornece uma lista completa dos recursos do Pylint.

### 2.1.3 Como posso criar um binário independente a partir de um script Python?

Não precisa possuir a capacidade de compilar o código Python para C se o que deseja é um programa autônomo que os usuários possam baixar e executar sem ter que instalar a distribuição Python primeiro. Existem várias ferramentas que determinam o conjunto de módulos exigidos por um programa e vinculam esses módulos junto com o binário do Python para produzir um único executável.

Um deles é usar a ferramenta de freeze, que está incluída na árvore de origem do Python como `Tools/freeze`. A mesma converte o código bytecode do Python em matrizes C; com um compilador C poderás incorporar todos os módulos em um novo programa, que será então vinculado aos módulos padrão do Python.

Ela funciona escaneando seu código recursivamente pelas instruções de importação (ambas as formas) e procurando pelos módulos no caminho padrão do Python e também no diretório fonte (para módulos internos). Então torna o bytecode de módulos escritos em Python em código C (inicializadores de vetor que podem ser transformado em objetos código usando o módulo marshal) e cria um arquivo de configurações customizado que só contém aqueles módulos internos que são na realidade usados no programa. A ferramenta então compila os códigos gerados em C e liga como o resto do interpretador Python para formar um binário autônomo que age exatamente como seu script.

Obviamente, freeze requer um compilador C. Existem diversos outros serviços que não requerem o compilador C. Uma opção é Thomas Heller's py2exe (Somente no Windows)

<http://www.py2exe.org/>

Another tool is Anthony Tuininga's `cx_Freeze`.

### 2.1.4 Existem padrões para a codificação ou um guia de estilo utilizado pela comunidade Python?

Sim. O guia de estilo esperado para módulos e biblioteca padrão possui o nome de PEP8 e pode acessar a sua documentação em [PEP 8](#).

## 2.1.5 Meu programa está muito lento. Como faço para melhorar a performance?

That's a tough one, in general. There are many tricks to speed up Python code; consider rewriting parts in C as a last resort.

In some cases it's possible to automatically translate Python to C or x86 assembly language, meaning that you don't have to modify your code to gain increased speed.

`Pyrex` can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms.

`Psyco` is a just-in-time compiler that translates Python code into x86 assembly language. If you can use it, Psyco can provide dramatic speedups for critical functions.

The rest of this answer will discuss various tricks for squeezing a bit more speed out of Python code. *Never* apply any optimization tricks unless you know you need them, after profiling has indicated that a particular function is the heavily executed hot spot in the code. Optimizations almost always make the code less clear, and you shouldn't pay the costs of reduced clarity (increased development time, greater likelihood of bugs) unless the resulting performance benefit is worth it.

There is a page on the wiki devoted to [performance tips](#).

Guido van Rossum has written up an anecdote related to optimization at <https://www.python.org/doc/essays/list2str>.

One thing to notice is that function and (especially) method calls are rather expensive; if you have designed a purely OO interface with lots of tiny functions that don't do much more than get or set an instance variable or call another method, you might consider using a more direct way such as directly accessing instance variables. Also see the standard module `profile` which makes it possible to find out where your program is spending most of its time (if you have some patience – the profiling itself can slow your program down by an order of magnitude).

Remember that many standard optimization heuristics you may know from other programming experience may well apply to Python. For example it may be faster to send output to output devices using larger writes rather than smaller ones in order to reduce the overhead of kernel system calls. Thus CGI scripts that write all output in “one shot” may be faster than those that write lots of small pieces of output.

Also, be sure to use Python's core features where appropriate. For example, slicing allows programs to chop up lists and other sequence objects in a single tick of the interpreter's mainloop using highly optimized C implementations. Thus to get the same effect as:

```
L2 = []
for i in range(3):
    L2.append(L1[i])
```

it is much shorter and far faster to use

```
L2 = list(L1[:3]) # "list" is redundant if L1 is a list.
```

Note that the functionally-oriented built-in functions such as `map()`, `zip()`, and friends can be a convenient accelerator for loops that perform a single task. For example to pair the elements of two lists together:

```
>>> zip([1, 2, 3], [4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

or to compute a number of sines:

```
>>> map(math.sin, (1, 2, 3, 4))
[0.841470984806, 0.909297426826, 0.14112000806, -0.756802495308]
```

The operation completes very quickly in such cases.

Other examples include the `join()` and `split()` methods of string objects. For example if `s1..s7` are large (10K+) strings then `"".join([s1, s2, s3, s4, s5, s6, s7])` may be far faster than the more obvious `s1+s2+s3+s4+s5+s6+s7`, since the “summation” will compute many subexpressions, whereas `join()` does all the copying in one pass. For manipulating strings, use the `replace()` and the `format()` methods on string objects. Use regular expressions only when you’re not dealing with constant string patterns. You may still use the old `%` operations `string % tuple` and `string % dictionary`.

Be sure to use the `list.sort()` built-in method to do sorting, and see the [sorting mini-HOWTO](#) for examples of moderately advanced usage. `list.sort()` beats other techniques for sorting in all but the most extreme circumstances.

Another common trick is to “push loops into functions or methods.” For example suppose you have a program that runs slowly and you use the profiler to determine that a Python function `ff()` is being called lots of times. If you notice that `ff()`:

```
def ff(x):
    ... # do something with x computing result...
    return result
```

tends to be called in loops like:

```
list = map(ff, oldlist)
```

or:

```
for x in sequence:
    value = ff(x)
    ... # do something with value...
```

then you can often eliminate function call overhead by rewriting `ff()` to:

```
def ffseq(seq):
    resultseq = []
    for x in seq:
        ... # do something with x computing result...
        resultseq.append(result)
    return resultseq
```

and rewrite the two examples to `list = ffseq(oldlist)` and to:

```
for value in ffseq(sequence):
    ... # do something with value...
```

Single calls to `ff(x)` translate to `ffseq([x])[0]` with little penalty. Of course this technique is not always appropriate and there are other variants which you can figure out.

You can gain some performance by explicitly storing the results of a function or method lookup into a local variable. A loop like:

```
for key in token:
    dict[key] = dict.get(key, 0) + 1
```

resolves `dict.get` every iteration. If the method isn’t going to change, a slightly faster implementation is:

```
dict_get = dict.get # look up the method once
for key in token:
    dict[key] = dict_get(key, 0) + 1
```

Default arguments can be used to determine values once, at compile time instead of at run time. This can only be done for functions or objects which will not be changed during program execution, such as replacing



```
def degree_sin(deg):
    return math.sin(deg * math.pi / 180.0)
```

with

```
def degree_sin(deg, factor=math.pi/180.0, sin=math.sin):
    return sin(deg * factor)
```

Because this trick uses default arguments for terms which should not be changed, it should only be used when you are not concerned with presenting a possibly confusing API to your users.

## 2.2 Núcleo da Linguagem

### 2.2.1 Porque estou recebo o erro `UnboundLocalError` quando a variável possui um valor associado?

Talvez você se surpreenda ao receber `UnboundLocalError` num código que anteriormente funcionava quando este for modificado e adicionando uma declaração de atribuição em algum lugar no corpo de uma função.

Este código:

```
>>> x = 10
>>> def bar():
...     print x
>>> bar()
10
```

funciona, mas este código:

```
>>> x = 10
>>> def foo():
...     print x
...     x += 1
```

resultará em um `UnboundLocalError`:

```
>>> foo()
Traceback (most recent call last):
...
UnboundLocalError: local variable 'x' referenced before assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in `foo` assigns a new value to `x`, the compiler recognizes it as a local variable. Consequently when the earlier `print x` attempts to print the uninitialized local variable and an error results.

No exemplo acima, podemos acessar a variável do escopo externo declarando-o globalmente:

```
>>> x = 10
>>> def foobar():
...     global x
...     print x
...     x += 1
>>> foobar()
10
```

Esta declaração explícita é necessária para lembrá-lo de que (ao contrário da situação superficialmente análoga com variáveis de classe e instância), você realmente está modificando o valor da variável no escopo externo:

```
>>> print x
11
```

### 2.2.2 Quais são as regras para variáveis locais e globais em Python?

Em Python, as variáveis que são apenas utilizadas (referenciadas) dentro de uma função são implicitamente globais. Se uma variável for associada a um valor em qualquer lugar dentro do corpo da função, assume-se que a mesma será local, a menos que seja explicitamente declarado como global.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you'd be using `global` all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the `global` declaration for identifying side-effects.

### 2.2.3 Por que os lambdas definidos em um loop com valores diferentes retornam o mesmo resultado?

Suponha que utilize um loop `for` para definir algumas funções lambdas (ou mesmo funções simples), por exemplo.:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Isso oferece uma lista que contém 5 lambdas que calculam  $x^2$ . Poderás esperar que, quando invocado, os mesmos retornem, respectivamente, 0, 1, 4, 9, e 16. No entanto, quando realmente tentares, verás que todos retornam 16:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Isso acontece porque `x` não é local para os lambdas, mas é definido no escopo externo, e é acessado quando o lambda for chamado — não quando é definido. No final do loop, o valor de `x` será 4, e então, todas as funções agora retornarão  $4^2$ , ou seja, 16. Também poderás verificar isso alterando o valor de `x` e vendo como os resultados dos lambdas mudam:

```
>>> x = 8
>>> squares[2]()
64
```

Para evitar isso, precisarás salvar os valores nas variáveis locais para os lambdas, para que eles não dependam do valor de `x` global:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Aqui, `n=x` cria uma nova variável `n` local para o lambda e calculada quando o lambda será definido para que ele tenha o mesmo valor que `x` tenha nesse ponto no loop. Isso significa que o valor de `n` será 0 no primeiro “ciclo” do lambda, 1 no segundo “ciclo”, 2 no terceiro, e assim por diante. Portanto, cada lambda agora retornará o resultado correto:

```
>>> squares[2] ()
4
>>> squares[4] ()
16
```

Observe que esse comportamento não é peculiar dos lambdas, o mesmo também ocorre com as funções regulares.

## 2.2.4 Como definir variáveis globais dentro de módulos?

A maneira canônica de compartilhar informações entre módulos dentro de um único programa é criando um módulo especial (geralmente chamado de config ou cfg). Basta importar o módulo de configuração em todos os módulos da sua aplicação; O módulo ficará disponível como um nome global. Como há apenas uma instância de cada módulo, todas as alterações feitas no objeto do módulo se refletem em todos os lugares. Por exemplo:

config.py:

```
x = 0    # Default value of the 'x' configuration setting
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print config.x
```

Observe o uso de um único módulo também é, por definição, a implementação do Design Patterns Singleton!

## 2.2.5 Quais são as “melhores práticas” quando fazemos uso da importação de módulos?

Em geral, não use `from modulename import *`. Ao fazê-lo, o namespace do importador é mais difícil e torna muito mais difícil para as ferramentas linters detectar nomes indefinidos.

Faça a importação de módulos na parte superior do arquivo. Isso deixa claro quais outros módulos nosso código necessita e evita dúvidas sobre por exemplo, se o nome do módulo está no escopo. Usar uma importação por linha facilita a adição e exclusão de importações de módulos, porém, usar várias importações num única linha, ocupa menos espaço da tela.

É uma boa prática importar os módulos na seguinte ordem:

1. módulos de biblioteca padrão, por exemplo: `sys`, `os`, `getopt`, `re`
2. módulos e biblioteca de terceiros (qualquer instalação feita contida no repositório de códigos na pasta `site-packages`) - por exemplo `mx.DateTime`, `ZODB`, `PIL.Image`, etc.
3. módulos desenvolvidos localmente

Only use explicit relative package imports. If you're writing code that's in the `package.sub.m1` module and want to import `package.sub.m2`, do not just write `import m2`, even though it's legal. Write `from package.sub import m2` or `from . import m2` instead.

Às vezes, é necessário transferir as importações para uma função ou classe para evitar problemas com importação circular. Gordon McMillan diz:

As importações circulares estão bem onde ambos os módulos utilizam a forma de importação “import 1”. Eles falham quando o 2º módulo quer pegar um nome do primeiro (“from module import name”) e a importação está no nível superior. Isso porque os nomes no primeiro ainda não estão disponíveis, porque o primeiro módulo está ocupado importando o 2º.

Nesse caso, se o segundo módulo for usado apenas numa função, a importação pode ser facilmente movida para dentro do escopo dessa função. No momento em que a importação for chamada, o primeiro módulo terá finalizado a inicialização e o segundo módulo poderá ser importado sem maiores complicações.

Também poderá ser necessário mover as importações para fora do nível superior do código se alguns dos módulos forem específicos de uma determinada plataforma (SO). Nesse caso, talvez nem seja possível importar todos os módulos na parte superior do arquivo. Nessas situações devemos importar os módulos que são específicos de cada plataforma antes de necessitar utilizar os mesmos.

Apenas mova as importações para um escopo local, como dentro da definição de função, se for necessário resolver algum tipo de problema, como exemplo, evitar importações circulares ou tentar reduzir o tempo de inicialização do módulo. Esta técnica é especialmente útil se muitas das importações forem desnecessárias, dependendo de como o programa é executado. Também podemos desejar mover as importações para uma função se os módulos forem usados somente nessa função. Note que carregar um módulo pela primeira vez pode ser demorado devido ao tempo de inicialização de cada módulo, no entanto, carregar um módulo várias vezes é praticamente imperceptível, tendo somente o custo de processamento de pesquisas no dicionário de nomes. Mesmo que o nome do módulo tenha saído do escopo, o módulo provavelmente estará disponível em `sys.modules`.

## 2.2.6 Por que os valores padrão são compartilhados entre objetos?

Este tipo de erro geralmente pega programadores neófitos. Considere esta função:

```
def foo(mydict={}): # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

A primeira vez que chamares essa função, `mydict` irá conter um único item. A segunda vez, `mydict` irá conter dois itens, porque quando `foo()` começar a ser executado, `mydict` começará com um item já existente.

Muitas vezes, espera-se que ao invocar uma função seja criado novos objetos referente aos valores padrão. Isso não é o que acontecerá. Os valores padrão são criados exatamente uma vez, quando a função está sendo definida. Se esse objeto for alterado, como o dicionário neste exemplo, as chamadas subsequentes para a essa função se referirão a este objeto alterado.

Por definição, objetos imutáveis, como números, strings, tuplas e o `None`, estão protegidos de sofrerem alteração. Alterações em objetos mutáveis, como dicionários, listas e instâncias de classe, podem levar à confusão.

Por causa desse recurso, é uma boa prática de programação para evitar o uso de objetos mutáveis contendo valores padrão. Em vez disso, utilize `None` como o valor padrão e dentro da função, verifique se o parâmetro é `None` e crie uma nova lista /dicionário/ o que quer que seja. Por exemplo, escreva o seguinte código:

```
def foo(mydict={}):
    ...
```

mas:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {} # create a new dict for local namespace
```

Esse recurso pode ser útil. Quando tiveres uma função que consome muito tempo para calcular, uma técnica comum é armazenar em cache os parâmetros e o valor resultante de cada chamada para a função e retornar o valor em cache se o mesmo valor for solicitado novamente. Isso se chama “memoizar”, e pode ser implementado da seguinte forma:

```
# Callers will never provide a third parameter for this function.
def expensive(arg1, arg2, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result          # Store result in the cache
    return result
```

Poderias usar uma variável global contendo um dicionário ao invés do valor padrão; isso é uma questão de gosto.

## 2.2.7 Como passar parâmetros opcionais ou parâmetros na forma de keyword de uma função a outra?

Preceda os argumentos com o uso de especificadores (asteriscos) `*` ou `**` na lista de parâmetros da função; Isso faz com que os argumentos posicionais como uma tupla e os keyword arguments sejam passados como um dicionário. Poderás, também, passar esses argumentos ao invocar outra função usando `*` e `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

In the unlikely case that you care about Python versions older than 2.0, use `apply()`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    apply(g, (x,) + args, kwargs)
```

## 2.2.8 Qual a diferença entre argumentos e parâmetros?

*Parâmetros* são definidos pelos nomes que aparecem na definição da função, enquanto *argumentos* são os valores que serão passados para a função no momento em que esta estiver sendo invocada. Os parâmetros irão definir quais os tipos de argumentos que uma função pode receber. Por exemplo, dada a definição da função:

```
def func(foo, bar=None, **kwargs):
    pass
```

`foo`, `bar` e `kwargs` são parâmetros de `func`. Dessa forma, ao invocar `func`, por exemplo:

```
func(42, bar=314, extra=somevar)
```

os valores 42, 314, e `somevar` são os argumentos.

## 2.2.9 Por que ao alterar a lista 'y' também altera a lista 'x'?

Se escreveres um código como:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

Poderás estar se perguntando por que acrescentar um elemento a `y` também mudou `x`.

Há 2 fatores que produzem esse resultado, são eles:

- 1) As variáveis são simplesmente nomes que referem-se a objetos. Ao escrevermos `y = x` não criará uma cópia da lista – criará uma nova variável `y` que irá se referir ao mesmo objeto que `x` está se referindo. Isso significa que existe apenas um objeto (lista), e ambos nomes (variáveis) `x` e `y` estão associados ao mesmo.
- 2) Listas são objetos mutáveis *mutable*, o que significa que você pode alterar o seu conteúdo.

Após invocar para `append()`, o conteúdo do objeto mutável mudou de `[]` para `[10]`. Uma vez que ambas as variáveis referem-se ao mesmo objeto, usar qualquer um dos nomes acessará o valor modificado `[10]`.

Se por acaso, atribuímos um objeto imutável a `x`:

```
>>> x = 5 # ints are immutable
>>> y = x
>>> x = x + 1 # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

veremos que nesse caso `x` e `y` não são mais iguais. Isso ocorre porque os números inteiros são *immutable*, e quando fizermos `x=x+1` não estaremos mudando o int 5 e incrementando o seu valor; em vez disso, estamos criando um novo objeto (o int 6) e associando `x` (isto é, mudando para o objeto no qual `x` se refere). Após esta tarefa, temos dois objetos (os ints 6 e 5) e duas variáveis que referem-se a elas (`x` agora se refere a 6 mas `y` ainda refere-se a 5).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, `+=` mutates lists but not tuples or ints (`a_list += [1, 2, 3]` is equivalent to `a_list.extend([1, 2, 3])` and mutates `a_list`, whereas `some_tuple += (1, 2, 3)` and `some_int += 1` create new objects).

Em outras palavras:

- Se tivermos objetos mutáveis (`list`, `dict`, `set`, etc.), podemos usar algumas operações específicas para alterá-lo e todas as variáveis que se referem a ela sofrerão também a mudança.
- Caso tenhamos um objeto imutável (`str`, `int`, `tuple`, etc.), todas as variáveis que se referem as mesmas sempre verão o mesmo valor, mas as operações que transformam-se nesses valores sempre retornarão novos objetos.

Se quiseres saber se duas variáveis se referem ao mesmo objeto ou não, podes usar a palavra-chave `is`, ou a função embutida `id()`.

## 2.2.10 Como escrever uma função com parâmetros de saída (invocada por referência)?

Lembre-se de que os argumentos são passados por atribuição em Python. Uma vez que a tarefa apenas cria referências a objetos, não existe “alias” entre um nome de argumento naquele que invocado e o destinatário, portanto, não há referência de chamada por si. Podes alcançar o efeito desejado de várias maneiras.

- 1) Retornando um Tupla com os resultados:

```
def func2(a, b):
    a = 'new-value'           # a and b are local names
    b = b + 1                 # assigned to new objects
    return a, b               # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print x, y                   # output: new-value 100
```

Esta quase sempre é a solução mais clara.

- 2) Utilizando variáveis globais. Essa forma de trabalho não é segura para uso com thread e portanto, a mesma não é recomendada.
- 3) Pela passagem de um objeto mutável (que possa ser alterado no local)

```
def func1(a):
    a[0] = 'new-value'        # 'a' references a mutable list
    a[1] = a[1] + 1           # changes a shared object

args = ['old-value', 99]
func1(args)
print args[0], args[1]       # output: new-value 100
```

- 4) Pela passagem de um dicionário que seja mutável:

```
def func3(args):
    args['a'] = 'new-value'    # args is a mutable dictionary
    args['b'] = args['b'] + 1  # change it in-place

args = {'a': 'old-value', 'b': 99}
func3(args)
print args['a'], args['b']
```

- 5) Ou agrupando valores numa instância de classe:

```
class callByRef:
    def __init__(self, **args):
        for (key, value) in args.items():
            setattr(self, key, value)

def func4(args):
    args.a = 'new-value'       # args is a mutable callByRef
    args.b = args.b + 1        # change object in-place

args = callByRef(a='old-value', b=99)
func4(args)
print args.a, args.b
```

Quase nunca existe uma boa razão para complicar isso.

A sua melhor escolha será retornar uma Tupla contendo os múltiplos resultados.

### 2.2.11 Como fazer uma função de ordem superior em Python?

Existem duas opções: podes usar escopos aninhados ou poderás usar objetos invocáveis. Por exemplo, suponha que desejasses definir que `linear(a, b)` retorne uma função  $f(x)$  que calcule o valor  $a * x + b$ . Usando escopos aninhados, temos:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Ou utilizando objetos invocáveis:

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

Em ambos os casos:

```
taxes = linear(0.3, 2)
```

dado um objeto invocável, onde `taxes(10e6) == 0.3 * 10e6 + 2`.

A abordagem do objeto invocável tem a desvantagem de que é um pouco mais lento e resulta num código ligeiramente mais longo. No entanto, note que uma coleção de callables pode compartilhar sua assinatura via herança:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Objetos podem encapsular o estado para vários métodos:

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Aqui `inc()`, `dec()` e `reset()` funcionam como funções que compartilham a mesma variável contadora.



## 2.2.12 Como faço para copiar um objeto no Python?

Basicamente, tente utilizar a função `copy.copy()` ou a função `copy.deepcopy()` para casos gerais. Nem todos os objetos podem ser copiados, mas a maioria poderá.

Alguns objetos podem ser copiados com mais facilidade. Os dicionários têm um método `copy()`:

```
newdict = olddict.copy()
```

As sequências podem ser copiadas através do uso de fatiamento:

```
new_l = l[:]
```

## 2.2.13 Como posso encontrar os métodos ou atributos de um objeto?

Para uma instância X de uma classe definida pelo usuário, `dir(x)` retorna uma lista organizada alfabeticamente dos nomes contidos, os atributos da instância e os métodos e atributos definidos por sua classe.

## 2.2.14 Como que o meu código pode descobrir o nome de um objeto?

Generally speaking, it can't, because objects don't really have names. Essentially, assignment always binds a name to a value; The same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print b
<__main__.A instance at 0x16D07CC>
>>> print a
<__main__.A instance at 0x16D07CC>
```

Provavelmente, a classe tem um nome: mesmo que seja vinculada a dois nomes e invocada através do nome B, a instância criada ainda é relatada como uma instância da classe A. No entanto, é impossível dizer se o nome da instância é A ou B, uma vez que ambos os nomes estão vinculados ao mesmo valor.

De um modo geral, não deveria ser necessário que o seu código “conheça os nomes” de valores específicos. A menos que escrevas deliberadamente programas introspectivos, isso geralmente é uma indicação de que uma mudança de abordagem pode ser benéfica.

Em `comp.lang.python`, Fredrik Lundh deu uma excelente analogia em resposta a esta pergunta:

The same way as you get the name of that cat you found on your porch: the cat (object) itself cannot tell you its name, and it doesn't really care – so the only way to find out what it's called is to ask all your neighbours (namespaces) if it's their cat (object)...

...and don't be surprised if you'll find that it's known by many names, or no name at all!

## 2.2.15 O que há com a precedência do operador vírgula?

A vírgula não é um operador em Python. Considere este código:

```
>>> "a" in "b", "a"
(False, 'a')
```

Uma vez que a vírgula não seja um operador, mas um separador entre as expressões acima, o código será avaliado como se tivéssemos entrado:

```
("a" in "b"), "a"
```

não:

```
"a" in ("b", "a")
```

The same is true of the various assignment operators (=, += etc). They are not truly operators but syntactic delimiters in assignment statements.

## 2.2.16 Existe um equivalente ao operador “?:” ternário do C?

Yes, this feature was added in Python 2.5. The syntax would be as follows:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25

small = x if x < y else y
```

For versions previous to 2.5 the answer would be ‘No’.

## 2.2.17 Is it possible to write obfuscated one-liners in Python?

Yes. Usually this is done by nesting lambda within lambda. See the following three examples, due to Ulf Bartelt:

```
# Primes < 1000
print filter(None, map(lambda y: y*reduce(lambda x, y: x*y!=0,
map(lambda x, y: y%x, range(2, int(pow(y, 0.5)+1))), 1), range(2, 1000)))

# First 10 Fibonacci numbers
print map(lambda x, f=lambda x, f: (f(x-1, f)+f(x-2, f)) if x>1 else 1: f(x, f),
range(10))

# Mandelbrot set
print (lambda Ru, Ro, Iu, Io, IM, Sx, Sy: reduce(lambda x, y: x+y, map(lambda y,
Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, Sy=Sy, L=lambda yc, Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, i=IM,
Sx=Sx, Sy=Sy: reduce(lambda x, y: x+y, map(lambda x, xc=Ru, yc=yc, Ru=Ru, Ro=Ro,
i=i, Sx=Sx, F=lambda xc, yc, x, y, k, f: lambda xc, yc, x, y, k, f: (k<=0) or (x*x+y*y
>=4.0) or 1+f(xc, yc, x*x-y*y+xc, 2.0*x*y+yc, k-1, f): f(xc, yc, x, y, k, f): chr(
64+F(Ru+x*(Ro-Ru)/Sx, yc, 0, 0, i)), range(Sx)): L(Iu+y*(Io-Iu)/Sy), range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24)
#      \___ ___/ \___ ___/ | | | lines on screen
#          V       V      | | columns on screen
#          |       |      | maximum of "iterations"
```

(continua na próxima página)

(continuação da página anterior)

```
#           /           /_____ range on y axis
#           /_____ range on x axis
```

Não tente isso em casa, crianças!

## 2.3 Números e Strings

### 2.3.1 Como faço para especificar números inteiros hexadecimais e octal?

Para especificar um dígito no formato octal, preceda o valor octal com um zero e, em seguida, um “o” minúsculo ou maiúsculo. Por exemplo, para definir a variável “a” para o valor octal “10” (8 em decimal), digite:

```
>>> a = 0o10
>>> a
8
```

Hexadecimal é bem fácil. Basta preceder o número hexadecimal com um zero e, em seguida, um “x” minúsculo ou maiúsculo. Os dígitos hexadecimais podem ser especificados em letras maiúsculas e minúsculas. Por exemplo, no interpretador Python:

```
>>> a = 0xa5
>>> a
165
>>> b = 0xB2
>>> b
178
```

### 2.3.2 Por que `-22 // 10` retorna `-3`?

É principalmente direcionado pelo desejo de que `i % j` possui o mesmo sinal que `j`. Se quiseres isso, e também se desejares:

```
i == (i // j) * j + (i % j)
```

then integer division has to return the floor. C also requires that identity to hold, and then compilers that truncate `i // j` need to make `i % j` have the same sign as `i`.

There are few real use cases for `i % j` when `j` is negative. When `j` is positive, there are many, and in virtually all of them it's more useful for `i % j` to be  $\geq 0$ . If the clock says 10 now, what did it say 200 hours ago? `-190 % 12 == 2` is useful; `-190 % 12 == -10` is a bug waiting to bite.

**Nota:** On Python 2, `a / b` returns the same as `a // b` if `__future__.division` is not in effect. This is also known as “classic” division.

### 2.3.3 Como faço para converter uma String em um número?

Para inteiros, use o tipo embutido `int()`, por exemplo, `int('144') == 144`. Da mesma forma, `float()` converterá para um valor do tipo ponto flutuante, por exemplo `float('144') == 144.0`.

By default, these interpret the number as decimal, so that `int('0144') == 144` and `int('0x144')` raises `ValueError`. `int(string, base)` takes the base to convert from as a second optional argument, so `int('0x144', 16) == 324`. If the base is specified as 0, the number is interpreted using Python's rules: a leading '0' indicates octal, and '0x' indicates a hex number.

Do not use the built-in function `eval()` if all you need is to convert strings to numbers. `eval()` will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass `__import__('os').system("rm -rf $HOME")` which would erase your home directory.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because Python regards numbers starting with '0' as octal (base 8).

### 2.3.4 Como faço para converter um número numa string?

To convert, e.g., the number 144 to the string '144', use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the `formatstrings` section, e.g. `"{:04d}".format(144)` yields '0144' and `"{: .3f}".format(1.0/3.0)` yields '0.333'. In Python 2, the division (/) operator returns the floor of the mathematical result of division if the arguments are ints or longs, but it returns a reasonable approximation of the division result if the arguments are floats or complex:

```
>>> print('{: .3f}'.format(1/3))
0.000
>>> print('{: .3f}'.format(1.0/3))
0.333
```

In Python 3, the default behaviour of the division operator (see [PEP 238](#)) has been changed but you can have the same behaviour in Python 2 if you import `division` from `__future__`:

```
>>> from __future__ import division
>>> print('{: .3f}'.format(1/3))
0.333
```

You may also use the `%` operator on strings. See the library reference manual for details.

### 2.3.5 Como faço para modificar uma string no lugar?

You can't, because strings are immutable. If you need an object with this ability, try converting the string to a list or use the `array` module:

```
>>> import io
>>> s = "Hello, world"
>>> a = list(s)
>>> print a
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd']
>>> a[7:] = list("there!")
>>> ''.join(a)
'Hello, there!'

>>> import array
```

(continua na próxima página)

(continuação da página anterior)

```
>>> a = array.array('c', s)
>>> print a
array('c', 'Hello, world')
>>> a[0] = 'y'; print a
array('c', 'yello, world')
>>> a.tostring()
'yello, world'
```

### 2.3.6 Como faço para invocar funções/métodos através de uma String?

Existem várias técnicas.

- A melhor forma é usar um dicionário que mapeie a Strings para funções. A principal vantagem desta técnica é que as Strings não precisam combinar os nomes das funções. Esta é também a principal técnica utilizada para emular uma construção de maiúsculas e minúsculas

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

- Utilize a função embutida `getattr()`:

```
import foo
getattr(foo, 'bar')()
```

Observe que a função `getattr()` funciona com qualquer objeto, incluindo classes, instâncias de classe, módulos e assim por diante.

A mesma é usado em vários lugares na biblioteca padrão, como este:

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- Utilize a função `locals()` ou a função `eval()` para resolver o nome da função

```
def myFunc():
    print "hello"

fname = "myFunc"

f = locals()[fname]
f()
```

(continua na próxima página)

(continuação da página anterior)

```
f = eval(fname)
f()
```

Note: Usar a função `eval()` é lento e perigoso. Se você não tiver controle absoluto sobre o conteúdo da String, alguém pode passar uma String que resulte numa função arbitrária sendo executada dentro da sua aplicação

### 2.3.7 Existe um equivalente em Perl `chomp()` para remover linhas novas de uma String?

Starting with Python 2.2, you can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other trailing whitespace. If the string `S` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```
>>> lines = ("line 1 \r\n"
...         "\r\n"
...         "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Geralmente isso só é desejado ao ler um texto linha por linha, usando `S.rstrip()` dessa maneira funciona bem.

For older versions of Python, there are two partial substitutes:

- If you want to remove all trailing whitespace, use the `rstrip()` method of string objects. This removes all trailing whitespace, not just a single newline.
- Otherwise, if there is only one line in the string `S`, use `S.splitlines()[0]`.

### 2.3.8 Existe uma função `scanf()` ou `sscanf()` ou algo equivalente?

Não como tal.

Para a análise de entrada simples, a abordagem mais fácil geralmente é dividir a linha em palavras delimitadas por espaços em branco usando o método `str.split()` de objetos Strings e, em seguida, converter as Strings decimais para valores numéricos usando a função `int()` ou a função `float()`. A função `split()` suporta um parâmetro “sep” opcional que é útil se a linha utilizar algo diferente de espaço em branco como separador.

Para entradas de textos mais complicadas, as expressões regulares são mais poderosas do que as funções C's `sscanf()` e mais adequadas para essa tarefa.

### 2.3.9 What does ‘UnicodeError: ASCII [decoding,encoding] error: ordinal not in range(128)’ mean?

This error indicates that your Python installation can handle only 7-bit ASCII strings. There are a couple ways to fix or work around the problem.

If your programs must handle data in arbitrary character set encodings, the environment the application runs in will generally identify the encoding of the data it is handing you. You need to convert the input to Unicode data using that encoding. For example, a program that handles email or web input will typically find character set encoding information in Content-Type headers. This can then be used to properly convert input data to Unicode. Assuming the string referred to by `value` is encoded as UTF-8:

```
value = unicode(value, "utf-8")
```

will return a Unicode object. If the data is not correctly encoded as UTF-8, the above call will raise a `UnicodeError` exception.

If you only want strings converted to Unicode which have non-ASCII data, you can try converting them first assuming an ASCII encoding, and then generate Unicode objects if that fails:

```
try:
    x = unicode(value, "ascii")
except UnicodeError:
    value = unicode(value, "utf-8")
else:
    # value was valid ASCII data
    pass
```

It's possible to set a default encoding in a file called `sitecustomize.py` that's part of the Python library. However, this isn't recommended because changing the Python-wide default encoding may cause third-party extension modules to fail.

Note that on Windows, there is an encoding known as "mbcs", which uses an encoding specific to your current locale. In many cases, and particularly when working with COM, this may be an appropriate default encoding to use.

## 2.4 Sequencias (Tuples/Lists)

### 2.4.1 Como faço para converter tuplas em listas?

O construtor de tipo `tuple(seq)` converte qualquer sequência (na verdade, qualquer iterável) numa tupla com os mesmos itens na mesma ordem.

Por exemplo, `tuple([1, 2, 3])` produz `(1, 2, 3)` e `tuple('abc')` produz `('a', 'b', 'c')`. Se o argumento for uma tupla, a mesma não faz uma cópia, mas retorna o mesmo objeto, por isso é barato invocar a função `tuple()` quando você não tiver certeza que determinado objeto já é uma tupla.

O construtor de tipos `list(seq)` converte qualquer sequência ou iterável em uma lista com os mesmos itens na mesma ordem. Por exemplo, `list((1, 2, 3))` produz `[1, 2, 3]` e `list('abc')` produz `['a', 'b', 'c']`. Se o argumento for uma lista, o mesmo fará uma cópia como em `seq[:]`.

### 2.4.2 O que é um índice negativo?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

### 2.4.3 How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function, which is new in Python 2.4:

```
for x in reversed(sequence):  
    ... # do something with x ...
```

This won't touch your original sequence, but build a new copy with reversed order to iterate over.

With Python 2.3, you can use an extended slice syntax:

```
for x in sequence[::-1]:  
    ... # do something with x ...
```

### 2.4.4 How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

<https://code.activestate.com/recipes/52560/>

If you don't mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```
if mylist:  
    mylist.sort()  
    last = mylist[-1]  
    for i in range(len(mylist)-2, -1, -1):  
        if last == mylist[i]:  
            del mylist[i]  
        else:  
            last = mylist[i]
```

If all elements of the list may be used as dictionary keys (i.e. they are all hashable) this is often faster

```
d = {}  
for x in mylist:  
    d[x] = 1  
mylist = list(d.keys())
```

In Python 2.5 and later, the following is possible instead:

```
mylist = list(set(mylist))
```

This converts the list into a set, thereby removing duplicates, and then back into a list.

### 2.4.5 How do you make an array in Python?

Utilize uma lista:

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that the Numeric extensions and others define array-like structures with various characteristics as well.



To get Lisp-style linked lists, you can emulate cons cells using tuples:

```
lisp_list = ("like", ("this", ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of lisp car is `lisp_list[0]` and the analogue of cdr is `lisp_list[1]`. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

## 2.4.6 Como faço para criar uma lista multidimensional?

Você provavelmente tentou fazer um Array multidimensional como isso:

```
>>> A = [None] * 2] * 3
```

This looks correct if you print it:

```
>>> A
[[None, None], [None, None], [None, None]]
```

Mas quando atribuíres um valor, o mesmo aparecerá em vários lugares:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

A razão é que replicar uma lista com `*` não cria cópias, ela apenas cria referências aos objetos existentes. O `*3` cria uma lista contendo 3 referências para a mesma lista que contém 2 itens cada. Mudanças numa linha serão mostradas em todas as linhas, o que certamente não é o que você deseja.

A abordagem sugerida é criar uma lista de comprimento desejado primeiro e, em seguida, preencher cada elemento com uma lista recém-criada:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

Isso gera uma lista contendo 3 listas diferentes contendo 2 itens cada. Você também pode usar uma list comprehension:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Ou, você pode usar uma extensão que forneça um tipo de dados de Array; [NumPy](#) é o melhor conhecido.

## 2.4.7 How do I apply a method to a sequence of objects?

Usando list comprehension:

```
result = [obj.method() for obj in mylist]
```

More generically, you can try the following function:

```
def method_map(objects, method, arguments):
    """method_map([a,b], "meth", (1,2)) gives [a.meth(1,2), b.meth(1,2)]"""
    nobjects = len(objects)
    methods = map(getattr, objects, [method]*nobjects)
    return map(apply, methods, [arguments]*nobjects)
```

## 2.4.8 Porque `a_tuple[i] += ['item']` levanta uma exceção quando a adição funciona?

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we'll use a `list` and `+=` as our exemplar.

Se você escrever:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

The reason for the exception should be immediately clear: 1 is added to the object `a_tuple[0]` points to (1), producing the result object, 2, but when we attempt to assign the result of the computation, 2, to element 0 of the tuple, we get an error because we can't change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is doing is approximately this:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__` is equivalent to calling `extend` on the list and returning the list. That's why we say that for lists, `+=` is a “shorthand” for `list.extend`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

This is equivalent to:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by `a_list` has been mutated, and the pointer to the mutated object is assigned back to `a_list`. The end result of the assignment is a no-op, since it is a pointer to the same object that `a_list` was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

The `__iadd__` succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

## 2.5 Dictionaries

### 2.5.1 How can I get a dictionary to display its keys in a consistent order?

You can't. Dictionaries store their keys in an unpredictable order, so the display order of a dictionary's elements will be similarly unpredictable.

This can be frustrating if you want to save a printable version to a file, make some changes and then compare it with some other printed dictionary. In this case, use the `pprint` module to pretty-print the dictionary; the items will be presented in order sorted by the key.

A more complicated solution is to subclass `dict` to create a `SortedDict` class that prints itself in a predictable order. Here's one simpleminded implementation of such a class:

```
class SortedDict(dict):
    def __repr__(self):
        keys = sorted(self.keys())
        result = ("{!r}: {!r}".format(k, self[k]) for k in keys)
        return "{{}".format(", ".join(result))

    __str__ = __repr__
```

This will work for many common situations you might encounter, though it's far from a perfect solution. The largest flaw is that if some values in the dictionary are also dictionaries, their values won't be presented in any particular order.

### 2.5.2 I want to do a complicated sort: can you do a Schwartzian Transform in Python?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its "sort value". In Python, use the `key` argument for the `sort()` function:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

### 2.5.3 How can I sort one list by values from another list?

Merge them into a single list of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs
[('what', 'something'), ('I'm', 'else'), ('sorting', 'to'), ('by', 'sort')]
>>> pairs.sort()
>>> result = [ x[1] for x in pairs ]
>>> result
['else', 'sort', 'to', 'something']
```

Uma alternativa para o último passo é:

```
>>> result = []
>>> for p in pairs: result.append(p[1])
```

If you find this more legible, you might prefer to use this instead of the final list comprehension. However, it is almost twice as slow for long lists. Why? First, the `append()` operation has to reallocate memory, and while it uses some tricks to avoid doing that each time, it still has to do it occasionally, and that costs quite a bit. Second, the expression “`result.append`” requires an extra attribute lookup, and third, there’s a speed reduction from having to make all those function calls.

## 2.6 Objetos

### 2.6.1 O que é uma classe?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic `Mailbox` class that provides basic accessor methods for a mailbox, and subclasses such as `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` that handle various specific mailbox formats.

### 2.6.2 O que é um método?

A method is a function on some object `x` that you normally call as `x.name(arguments...)`. Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

### 2.6.3 O que é o self?

Self is merely a conventional name for the first argument of a method. A method defined as `meth(self, a, b, c)` should be called as `x.meth(a, b, c)` for some instance `x` of the class in which the definition occurs; the called method will think it is called as `meth(x, a, b, c)`.

Veja também *Por que o 'self' deve ser usado explicitamente em definições de método e chamadas?*.

### 2.6.4 How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, long, float, complex))`.

Note that most programs do not use `isinstance()` on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ... # code to search a mailbox
    elif isinstance(obj, Document):
        ... # code to search a document
    elif ...
```

A better approach is to define a `search()` method on all the classes and just call it:

```
class Mailbox:
    def search(self):
        ... # code to search a mailbox

class Document:
    def search(self):
        ... # code to search a document

obj.search()
```

### 2.6.5 O que é delegation?

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile
```

(continua na próxima página)

(continuação da página anterior)

```
def write(self, s):
    self.__outfile.write(s.upper())

def __getattr__(self, name):
    return getattr(self.__outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self.__outfile.write()` method. All other methods are delegated to the underlying `self.__outfile` object. The delegation is accomplished via the `__getattr__` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of `__setattr__()` is roughly equivalent to the following:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Most `__setattr__()` implementations must modify `self.__dict__` to store local state for `self` without causing an infinite recursion.

## 2.6.6 How do I call a method defined in a base class from a derived class that overrides it?

If you're using new-style classes, use the built-in `super()` function:

```
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

If you're using classic classes: For a class definition such as `class Derived(Base): ...` you can call method `meth()` defined in `Base` (or one of `Base`'s base classes) as `Base.meth(self, arguments...)`. Here, `Base.meth` is an unbound method, so you need to provide the `self` argument.

## 2.6.7 How can I organize my code to make it easier to change the base class?

You could define an alias for the base class, assign the real base class to it before your class definition, and use the alias throughout your class. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```
BaseAlias = <real base class>

class Derived(BaseAlias):
    def meth(self):
        BaseAlias.meth(self)
    ...
```

## 2.6.8 How do I create static class data and static class methods?

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```
class C:
    count = 0    # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count    # or return self.count
```

`c.count` also refers to `C.count` for any `c` such that `isinstance(c, C)` holds, unless overridden by `c` itself or by some class on the base-class search path from `c.__class__` back to `C`.

Caution: within a method of `C`, an assignment like `self.count = 42` creates a new and unrelated instance named “count” in `self`’s own dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

```
C.count = 314
```

Static methods are possible since Python 2.2:

```
class C:
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
    static = staticmethod(static)
```

With Python 2.4’s decorators, this can also be written as

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

## 2.6.9 How can I overload constructors (or methods) in Python?

This answer actually applies to all methods, but the question usually comes up first in the context of constructors.

Em C++ escreveríamos

```
class C {  
    C() { cout << "No arguments\n"; }  
    C(int i) { cout << "Argument is " << i << "\n"; }  
}
```

In Python you have to write a single constructor that catches all cases using default arguments. For example:

```
class C:  
    def __init__(self, i=None):  
        if i is None:  
            print "No arguments"  
        else:  
            print "Argument is", i
```

Isso não é inteiramente equivalente, mas já está bem próximo.

Você também pode tentar uma lista de argumentos de comprimento variável, por exemplo:

```
def __init__(self, *args):  
    ...
```

A mesma abordagem funciona para todas as definições de métodos.

## 2.6.10 Eu tentei usar `__spam` e recebi um erro sobre `__SomeClassName__spam`.

Variable names with double leading underscores are “mangled” to provide a simple but effective way to define class private variables. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `__classname__spam`, where `classname` is the current class name with any leading underscores stripped.

This doesn’t guarantee privacy: an outside user can still deliberately access the “`__classname__spam`” attribute, and private values are visible in the object’s `__dict__`. Many Python programmers never bother to use private variable names at all.

## 2.6.11 My class defines `__del__` but it is not called when I delete the object.

Há várias razões possíveis para isto.

The `del` statement does not necessarily call `__del__()` – it simply decrements the object’s reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you’re trying to reproduce a problem. Worse, the order in which object’s `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it’s still a good idea to define an explicit `close()` method on objects to be called whenever you’re done with them. The `close()` method can then remove attributes that refer to subobjects. Don’t call `__del__()`



directly – `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the `weakref` module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

If the object has ever been a local variable in a function that caught an expression in an except clause, chances are that a reference to the object still exists in that function's stack frame as contained in the stack trace. Normally, calling `sys.exc_clear()` will take care of this by clearing the last recorded exception.

Finally, if your `__del__()` method raises an exception, a warning message is printed to `sys.stderr`.

## 2.6.12 How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.

## 2.6.13 Why does the result of `id()` appear to be not unique?

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

## 2.7 Módulos

### 2.7.1 Como faço para criar um arquivo .pyc?

When a module is imported for the first time (or when the source is more recent than the current compiled file) a `.pyc` file containing the compiled code should be created in the same directory as the `.py` file.

One reason that a `.pyc` file may not be created is permissions problems with the directory. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server. Creation of a `.pyc` file is automatic if you're importing a module and Python has the ability (permissions, free space, etc...) to write the compiled module back to the directory.

Running Python on a top level script is not considered an import and no `.pyc` will be created. For example, if you have a top-level module `foo.py` that imports another module `xyz.py`, when you run `foo.py` `xyz.pyc` will be created since `xyz` is imported, but no `foo.pyc` file will be created since `foo.py` isn't being imported.

If you need to create `foo.pyc` – that is, to create a `.pyc` file for a module that is not imported – you can, using the `py_compile` and `compileall` modules.

The `py_compile` module can manually compile any module. One way is to use the `compile()` function in that module interactively:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the `compileall` module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

### 2.7.2 Como encontro o nome do módulo atual?

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `'__main__'`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```
def main():
    print 'Running test...'
    ...

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

### 2.7.3 How can I have modules that mutually import each other?

Suponha que tenhas os seguintes módulos:

`foo.py`:

```
from bar import bar_var
foo_var = 1
```

`bar.py`:

```
from foo import foo_var
bar_var = 2
```

The problem is that the interpreter will perform the following steps:

- main imports foo
- Os globais vazios para foo são criados
- foo é compilado e começa a executar
- foo imports bar
- Empty globals for bar are created
- bar is compiled and starts executing

- `bar` imports `foo` (which is a no-op since there already is a module named `foo`)
- `bar.foo_var = foo.foo_var`

The last step fails, because Python isn't done with interpreting `foo` yet and the global symbol dictionary for `foo` is still empty.

The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>.<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don't need imported base classes)
- Declaração `import`
- active code (including globals that are initialized from imported values).

van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

These solutions are not mutually exclusive.

## 2.7.4 `__import__`('x.y.z') returns <module 'x'>; how do I get z?

Consider using the convenience function `import_module()` from `importlib` instead:

```
z = importlib.import_module('x.y.z')
```

## 2.7.5 When I edit an imported module and reimport it, the changes don't show up. Why does this happen?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force rereading of a changed module, do this:

```
import modname
reload(modname)
```

Warning: this technique is not 100% fool-proof. In particular, modules containing statements like

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour:

```
>>> import cls
>>> c = cls.C()           # Create an instance of C
>>> reload(cls)
<module 'cls' from 'cls.pyc'>
>>> isinstance(c, cls.C)  # isinstance is false!!?
False
```

The nature of the problem is made clear if you print out the class objects:

```
>>> c.__class__  
<class cls.C at 0x7352a0>  
>>> cls.C  
<class cls.C at 0x4198d0>
```

### 3.1 Por que o Python usa recuo para agrupamento de declarações?

Guido van Rossum acredita que usar recuo para agrupamento é extremamente elegante e contribui muito para a clareza do programa Python. Muitas pessoas aprendem a amar esta ferramenta depois de um tempo.

Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader. Occasionally C programmers will encounter a fragment of code like this:

```
if (x <= y)
    x++;
    y--;
z++;
```

Only the `x++` statement is executed if the condition is true, but the indentation leads you to believe otherwise. Even experienced C programmers will sometimes stare at it a long time wondering why `y` is being decremented even for `x > y`.

Because there are no begin/end brackets, Python is much less prone to coding-style conflicts. In C there are many different ways to place the braces. If you're used to reading and writing code that uses one style, you will feel at least slightly uneasy when reading (or being required to write) another style.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.

## 3.2 Por que eu estou recebendo resultados estranhos com simples operações aritméticas?

Veja a próxima questão.

## 3.3 Why are floating point calculations so inaccurate?

People are often very surprised by results like this:

```
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It's not. This has nothing to do with Python, but with how the underlying C platform handles floating point numbers, and ultimately with the inaccuracies introduced when writing down numbers as a string of a fixed number of digits.

The internal representation of floating point numbers uses a fixed number of binary digits to represent a decimal number. Some decimal numbers can't be represented exactly in binary, resulting in small roundoff errors.

In decimal math, there are many numbers that can't be represented with a fixed number of decimal digits, e.g.  $1/3 = 0.333333333\ldots$

In base 2,  $1/2 = 0.1$ ,  $1/4 = 0.01$ ,  $1/8 = 0.001$ , etc.  $.2$  equals  $2/10$  equals  $1/5$ , resulting in the binary fractional number  $0.001100110011001\ldots$

Floating point numbers only have 32 or 64 bits of precision, so the digits are cut off at some point, and the resulting number is  $0.19999999999999996$  in decimal, not  $0.2$ .

A floating point number's `repr()` function prints as many digits as necessary to make `eval(repr(f)) == f` true for any float `f`. The `str()` function prints fewer digits and this often results in the more sensible number that was probably intended:

```
>>> 1.1 - 0.9
0.20000000000000007
>>> print 1.1 - 0.9
0.2
```

One of the consequences of this is that it is error-prone to compare the result of some computation to a float with `==`. Tiny inaccuracies may mean that `==` fails. Instead, you have to check that the difference between the two numbers is less than a certain threshold:

```
epsilon = 0.000000000000001 # Tiny allowed error
expected_result = 0.4

if expected_result-epsilon <= computation() <= expected_result+epsilon:
    ...
```

Please see the chapter on floating point arithmetic in the Python tutorial for more information.

## 3.4 Por que strings do Python são imutáveis?

Existem várias vantagens.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as “elemental” as numbers. No amount of activity will change the value 8 to anything else, and in Python, no amount of activity will change the string “eight” to anything else.

## 3.5 Por que o ‘self’ deve ser usado explicitamente em definições de método e chamadas?

A ideia foi emprestada do Modula-3. Acontece dela ser muito útil, por vários motivos.

First, it’s more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don’t know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you’d have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren’t explicitly declared global), there has to be some way to tell the interpreter that an assignment was meant to assign to an instance variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn’t have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit `self.var` solves this nicely. Similarly, for using instance variables, having to write `self.var` means that references to unqualified names inside a method don’t have to search the instance’s directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.

## 3.6 Por que não posso usar uma atribuição em uma expressão?

Many people used to C or Perl complain that they want to use this C idiom:

```
while (line = readline(f)) {
    // do something with line
}
```

where in Python you’re forced to write this:

```
while True:
    line = f.readline()
    if not line:
        break
    ... # do something with line
```

The reason for not allowing assignment in Python expressions is a common, hard-to-find bug in those other languages, caused by this construct:

```
if (x = 0) {  
    // error handling  
}  
else {  
    // code that only works for nonzero x  
}
```

The error is a simple typo: `x = 0`, which assigns 0 to the variable `x`, was written while the comparison `x == 0` is certainly what was intended.

Many alternatives have been proposed. Most are hacks that save some typing but use arbitrary or cryptic syntax or keywords, and fail the simple criterion for language change proposals: it should intuitively suggest the proper meaning to a human reader who has not yet been introduced to the construct.

An interesting phenomenon is that most experienced Python programmers recognize the `while True` idiom and don't seem to be missing the assignment in expression construct much; it's only newcomers who express a strong desire to add this to the language.

There's an alternative way of spelling this that seems attractive but is generally less robust than the “while True” solution:

```
line = f.readline()  
while line:  
    ... # do something with line...  
    line = f.readline()
```

The problem with this is that if you change your mind about exactly how you get the next line (e.g. you want to change it into `sys.stdin.readline()`) you have to remember to change two places in your program – the second occurrence is hidden at the bottom of the loop.

The best approach is to use iterators, making it possible to loop through objects using the `for` statement. For example, in the current version of Python file objects support the iterator protocol, so you can now write simply:

```
for line in f:  
    ... # do something with line...
```

## 3.7 Por que o Python usa métodos para algumas funcionalidades (ex: `lista.index()`) mas funções para outras (ex: `len(lista)`)?

Como Guido disse:

(a) For some operations, prefix notation just reads better than postfix – prefix (and infix!) operations have a long tradition in mathematics which likes notations where the visuals help the mathematician thinking about a problem. Compare the ease with which we rewrite a formula like  $x*(a+b)$  into  $x*a + x*b$  to the clumsiness of doing the same thing using a raw OO notation.

(b) When I read code that says `len(x)` I *know* that it is asking for the length of something. This tells me two things: the result is an integer, and the argument is some kind of container. To the contrary, when I read `x.len()`, I have to already know that `x` is some kind of container implementing an interface or inheriting from a class that has a standard `len()`. Witness the confusion we occasionally have when a class that is not implementing a mapping has a `get()` or `keys()` method, or something that isn't a file has a `write()` method.

—<https://mail.python.org/pipermail/python-3000/2006-November/004643.html>



## 3.8 Por que o `join()` é um método de string em vez de ser um método de lista ou tupla?

Strings se tornaram muito parecidas com outros tipos padrão a partir do Python 1.6, quando métodos que dão a mesma funcionalidade que sempre esteve disponível utilizando as funções do módulo de string foram adicionados. A maior parte desses novos métodos foram amplamente aceitos, mas o que parece deixar alguns programadores desconfortáveis é:

```
"", ".join(['1', '2', '4', '8', '16'])
```

que dá o resultado:

```
"1, 2, 4, 8, 16"
```

Existem dois argumentos comuns contra esse uso.

The first runs along the lines of: “It looks really ugly using a method of a string literal (string constant)”, to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: “I am really telling a sequence to join its members together with a string constant”. Sadly, you aren’t. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split(", ")
```

is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space). In this case a Unicode string returns a list of Unicode strings, an ASCII string returns a list of ASCII strings, and everyone is happy.

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself.

Because this is a string method it can work for Unicode strings as well as plain ASCII strings. If `join()` were a method of the sequence types then the sequence types would have to decide which type of string to return depending on the type of the separator.

If none of these arguments persuade you, then for the moment you can continue to use the `join()` function from the string module, which allows you to write

```
string.join(['1', '2', '4', '8', '16'], ", ")
```

## 3.9 O quão rápidas são as exceções?

Um bloco de `try/except` é extremamente eficiente se nenhuma exceção for levantada. Na verdade, capturar uma exceção custa caro. Em versões do Python anteriores a 2.0 era comum utilizar esse idioma:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

Isso somente fazia sentido quando você esperava que o dicionário tivesse uma chave quase que toda vez. Se esse não fosse o caso, você escrevia desta maneira:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

---

**Nota:** In Python 2.0 and higher, you can code this as `value = mydict.setdefault(key, getvalue(key))`.

---

## 3.10 Por que não existe uma instrução de switch ou case no Python?

You can do this easily enough with a sequence of `if... elif... elif... else`. There have been some proposals for switch statement syntax, but there is no consensus (yet) on whether and how to do range tests. See [PEP 275](#) for complete details and the current status.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```
def function_1(...):
    ...

functions = {'a': function_1,
            'b': function_2,
            'c': self.method_1, ...}

func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```
def visit_a(self, ...):
    ...

...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

It's suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming from an untrusted source, an attacker would be able to call any method on your object.

### 3.11 Can't you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is [Stackless Python](#), which has a completely redesigned interpreter loop that avoids the C stack.

### 3.12 Por que expressões lambda não podem conter instruções?

Expressões lambda no Python não podem conter instruções porque o framework sintático do Python não consegue manipular instruções aninhadas dentro de expressões. No entanto, no Python, isso não é um problema sério. Diferentemente das formas de lambda em outras linguagens, onde elas adicionam funcionalidade, lambdas de Python são apenas notações simplificadas se você tiver muita preguiça de definir uma função.

Funções já são objetos de primeira classe em Python, e podem ser declaradas em um escopo local. Portanto a única vantagem de usar um lambda em vez de uma função definida localmente é que você não precisa inventar um nome para a função – mas esta só é uma variável local para a qual o objeto da função (que é exatamente do mesmo tipo de um objeto que uma expressão lambda carrega) é atribuído!

### 3.13 O Python pode ser compilado para linguagem de máquina, C ou alguma outra linguagem?

[Cython](#) compiles a modified version of Python with optional annotations into C extensions. [Nuitka](#) is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language. For compiling to Java you can consider [VOC](#).

### 3.14 Como o Python gerencia memória?

The details of Python memory management depend on the implementation. The standard C implementation of Python uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Jython relies on the Java runtime so the JVM's garbage collector is used. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

Sometimes objects get stuck in tracebacks temporarily and hence are not deallocated when you might expect. Clear the tracebacks with:

```
import sys
sys.exc_clear()
sys.exc_traceback = sys.last_traceback = None
```

Tracebacks are used for reporting errors, implementing debuggers and related things. They contain a portion of the program state extracted during the handling of an exception (usually the most recent exception).

In the absence of circularities and tracebacks, Python programs do not need to manage memory explicitly.

Why doesn't Python use a more traditional garbage collection scheme? For one thing, this is not a C standard feature and hence it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for *most* common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard `malloc()` and `free()` with versions provided by the GC library, an application embedding Python may want to have its *own* substitute for `malloc()` and `free()`, and may not want Python's. Right now, Python works with anything that implements `malloc()` and `free()` properly.

In Jython, the following code (which is fine in CPython) will probably run out of file descriptors long before it runs out of memory:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Using the current reference counting and destructor scheme, each new assignment to `f` closes the previous file. Using GC, this is not guaranteed. If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement; this will work regardless of GC:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

### 3.15 Why isn't all memory freed when Python exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.

### 3.16 Por que existem tipos de dados separados para tuplas e listas?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal records or C structs; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can't replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list's elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.

### 3.17 Como as listas são implementadas no CPython?

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

### 3.18 Como são os dicionários implementados no CPython?

CPython's dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key; for example, "Python" hashes to -539294296 while "python", a string that differs by a single bit, hashes to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time –  $O(1)$ , in computer science notation – to retrieve a key. It also means that no sorted order of the keys is maintained, and traversing the array as the `.keys()` and `.items()` do will output the dictionary's content in some arbitrary jumbled order.

### 3.19 Por que chaves de dicionário devem ser imutáveis?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can't tell that it was being used as a dictionary key, it can't move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won't be found because its hash value is different. If you tried to look up the old value it wouldn't be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function `tuple(L)` creates a tuple with the same entries as the list `L`. Tuples are immutable and can therefore be used as dictionary keys.

Algumas soluções inaceitáveis que foram propostas:

- Hash lists by their address (object ID). This doesn't work because if you construct a new list with the same value it won't be found; e.g.:

```
mydict = {[1, 2]: '12'}
print mydict[[1, 2]]
```

would raise a `KeyError` exception because the id of the `[1, 2]` used in the second line differs from that in the first line. In other words, dictionary keys should be compared using `==`, not using `is`.

- Make a copy when using a list as a key. This doesn't work because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an infinite loop.
- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.

- Mark lists as read-only once they are used as a dictionary key. The problem is that it's not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if `o1 == o2` (ie `o1.__eq__(o2)` is `True`) then `hash(o1) == hash(o2)` (ie, `o1.__hash__() == o2.__hash__()`), regardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of `ListWrapper`, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.

## 3.20 Por que `lista.sort()` não retorna a lista ordenada?

Em situações nas quais desempenho importa, fazer uma cópia da lista só para ordenar seria desperdício. Portanto, `lista.sort()` ordena a lista. De forma a lembrá-lo desse fato, isso não retorna a lista ordenada. Desta forma, você não vai ser confundido a acidentalmente sobrescrever uma lista quando você precisar de uma cópia ordenada mas também precisar manter a versão não ordenada.

In Python 2.4 a new built-in function – `sorted()` – has been added. This function creates a new list from a provided iterable, sorts it and returns it. For example, here's how to iterate over the keys of a dictionary in sorted order:

```
for key in sorted(mydict):
    ... # do whatever with mydict[key]...
```

## 3.21 How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components. There is also a tool, `PyChecker`, which can be used to find problems due to subclassing.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple “self test.” Even modules which use complex external interfaces can often be tested in isolation using trivial “stub” emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `append()` implementation will actually do this correctly, but it’s trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code with an eye to making it easily tested. One increasingly popular technique, test-directed development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.

## 3.22 Why is there no goto?

You can use exceptions to provide a “structured goto” that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the “go” or “goto” constructs of C, Fortran, and other languages. For example:

```
class label: pass # declare a label

try:
    ...
    if condition: raise label() # goto label
    ...
except label: # where to goto
    pass
...
```

This doesn’t allow you to jump into the middle of a loop, but that’s usually considered an abuse of goto anyway. Use sparingly.

### 3.23 Por que strings brutas (r-strings) não podem terminar com uma contrabarra?

More precisely, they can't end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you're trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```
f = open("/mydir/file.txt") # works fine!
```

If you're trying to build a pathname for a DOS command, try e.g. one of

```
dir = r"\\this\\is\\my\\dos\\dir" "\\\"
dir = r"\\this\\is\\my\\dos\\dir\" "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

### 3.24 Por que o Python não tem uma instrução “with” para atribuição de atributos?

Python has a ‘with’ statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some language have a construct that looks like this:

```
with obj:
    a = 1 # equivalent to obj.a = 1
    total = total + 1 # obj.total = obj.total + 1
```

In Python, such a construct would be ambiguous.

Outras linguagens, como Object Pascal, Delphi, e C++, usam tipos estáticos, então é possível saber, de maneira não ambígua, que membro está sendo atribuído. Esse é o principal ponto da tipagem estática – o compilador *sempre* sabe o escopo de toda variável em tempo de compilação.

O Python usa tipos dinâmicos. É impossível saber com antecedência que atributo vai ser referenciado em tempo de execução. Atributos membro podem ser adicionados ou removidos de objetos dinamicamente. Isso torna impossível saber, de uma leitura simples, que atributo está sendo referenciado: um atributo local, um atributo global ou um atributo membro?

For instance, take the following incomplete snippet:

```
def foo(a):
    with a:
        print x
```

The snippet assumes that “a” must have a member attribute called “x”. However, there is nothing in Python that tells the interpreter this. What should happen if “a” is, let us say, an integer? If there is a global variable named “x”, will it be used inside the with block? As you see, the dynamic nature of Python makes such choices much harder.

O benefício primário do “with” e funcionalidades similares da linguagem (redução de volume de código) pode, entretanto, ser facilmente alcançado no Python por atribuição. Em vez de:



```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

escreva isso:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

Isso também tem o efeito colateral de aumentar a velocidade de execução por que ligações de nome são resolvidas a tempo de execução em Python, e a segunda versão só precisa performar a resolução uma vez.

### 3.25 Por que dois pontos são necessários para as instruções de if/while/def/class?

Os dois pontos são obrigatórios primeiramente para melhorar a leitura (um dos resultados da linguagem experimental ABC). Considere isso:

```
if a == b
    print a
```

versus

```
if a == b:
    print a
```

Note como a segunda é ligeiramente mais fácil de ler. Note com mais atenção como os dois pontos iniciam o exemplo nessa resposta de perguntas frequentes; é um uso padrão em Português.

Outro motivo menor é que os dois pontos deixam mais fácil para os editores com realce de sintaxe; eles podem procurar por dois pontos para decidir quando a recuo precisa ser aumentada em vez de precisarem fazer uma análise mais elaborada do texto do programa.

### 3.26 Por que o Python permite vírgulas ao final de listas e tuplas?

O Python deixa você adicionar uma vírgula ao final de listas, tuplas e dicionários:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

Existem várias razões para permitir isso.

Quando você possui um valor literal para uma lista, tupla, ou dicionário disposta através de múltiplas linhas, é mais fácil adicionar mais elementos porque você não precisa lembrar de adicionar uma vírgula na linha anterior. As linhas também podem ser reordenadas sem criar um erro de sintaxe.

Acidentalmente omitir a vírgula pode levar a erros que são difíceis de diagnosticar. Por exemplo:

```
x = [  
    "fee",  
    "fie"  
    "foo",  
    "fum"  
]
```

Essa lista parece ter quatro elementos, mas na verdade contém três: “fee”, “fiefoo” e “fum”. Sempre adicionar a vírgula evita essa fonte de erro.

Permitir a vírgula no final também pode deixar a geração de código programático mais fácil.

---

FAQ de Bibliotecas e Extensões

---

## 4.1 Questões gerais sobre bibliotecas

### 4.1.1 Como encontrar um módulo ou aplicação para realizar uma tarefa X?

Verifique a Referência de Bibliotecas para ver se há um módulo relevante da biblioteca padrão. (Eventualmente, você aprenderá o que está na biblioteca padrão e poderá pular esta etapa.)

Para pacotes de terceiros, pesquise no [Python Package Index](#) ou tente no [Google](#) ou outro buscador na Web. Pesquisando por “Python” mais uma ou dois argumentos nomeados do seu tópico de interesse geralmente encontrará algo útil.

### 4.1.2 Onde está o código-fonte do `math.py` (`socket.py`, `regex.py`, etc.)?

Se você não conseguir encontrar um arquivo de origem para um módulo, ele pode ser um módulo interno ou carregado dinamicamente, implementado em C, C++ ou outra linguagem compilada. Nesse caso, você pode não ter o arquivo de origem ou pode ser algo como `mathmodule.c`, em algum lugar do diretório de origem C (não no caminho do Python).

Existem (pelo menos) três tipos de módulos no Python:

- 1) módulos escritos em Python (`.py`)
- 2) módulos escritos em C e carregados dinamicamente (`.dll`, `.pyd`, `.so`, `.sl`, etc.);
- 3) módulos escritos em C e vinculados ao interpretador; para obter uma dessas listas, digite:

```
import sys
print sys.builtin_module_names
```

### 4.1.3 Como tornar um script Python executável no Unix?

Você precisa fazer duas coisas: o arquivo do script deve ser executável e a primeira linha deve começar com “#!” seguido do caminho do interpretador Python.

Inicialmente, execute o `chmod +x scriptfile` ou, talvez, o `chmod 755 scriptfile`.

A segunda coisa pode ser feita de várias maneiras. A maneira mais direta é escrever

```
#!/usr/local/bin/python
```

como a primeira linha do seu arquivo, usando o endereço do caminho onde o interpretador Python está instalado.

Se você deseja que o script seja independente de onde o interpretador Python mora, você pode usar o programa **env**. Quase todas as variantes do Unix suportam o seguinte, assumindo que o interpretador Python esteja em um diretório no `PATH` do usuário:

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

Não faça isso para CGI scripts. A variável `PATH` para CGI scripts é normalmente muito pequena, portanto, você precisa usar o caminho completo do interpretador.

Occasionally, a user's environment is so full that the `/usr/bin/env` program fails; or there's no `env` program at all. In that case, you can try the following hack (due to Alex Rezinsky):

```
#!/bin/sh
""" : """
exec python $0 ${1+"$@"}
"""
```

Uma pequena desvantagem é que isso define o script's `__doc__` string. Entretanto, você pode corrigir isso adicionando

```
__doc__ = """...Whatever..."""
```

### 4.1.4 Existe um pacote de curses/termcap para Python?

For Unix variants the standard Python source distribution comes with a `curses` module in the [Modules](#) subdirectory, though it's not compiled by default. (Note that this is not available in the Windows distribution – there is no `curses` module for Windows.)

The `curses` module supports basic curses features as well as many additional functions from `ncurses` and `SVSV curses` such as colour, alternative character set support, pads, and mouse support. This means the module isn't compatible with operating systems that only have BSD curses, but there don't seem to be any currently maintained OSes that fall into this category.

Para Windows: use o [módulo consolelib](#).

### 4.1.5 Existe a função `onexit()` equivalente ao C no Python?

O módulo `atexit` fornece uma função de registro similar ao `onexit()` do C.

### 4.1.6 Por que o meu manipulador de sinal não funciona?

O maior problema é que o manipulador de sinal é declarado com uma lista de argumentos incorretos. Isso é chamado como

```
handler(signum, frame)
```

portanto, isso deve ser declarado com dois argumentos:

```
def handler(signum, frame):
    ...
```

## 4.2 Tarefas comuns

### 4.2.1 Como testar um programa ou componente Python?

A Python vem com dois frameworks de teste. O `doctest` busca por exemplos nas docstrings de um módulo e os executa, comparando o resultado com a saída esperada informada na docstring.

O módulo `unittest` é uma estrutura de teste mais sofisticada, modelada nas estruturas de teste do Java e do Smalltalk.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods – and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

A lógica principal do seu programa pode tão simples quanto

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

no botão do módulo principal do seus programa.

Once your program is organized as a tractable collection of functions and class behaviours you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it's surprisingly easy. You can make coding much more pleasant and fun by writing your test functions in parallel with the “production code”, since this makes it easy to find bugs and even design flaws earlier.

“Support modules” that are not intended to be the main module of a program may include a self-test of the module.

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

Mesmo quando as interfaces externas não estiverem disponíveis, os programas que interagem com interfaces externas complexas podem ser testados usando as interfaces “falsas” implementadas no Python.

## 4.2.2 Como faço para criar uma documentação de doc strings?

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is `epydoc`. `Sphinx` can also include docstring content.

## 4.2.3 Como faço para pressionar uma tecla de cada vez?

For Unix variants there are several solutions. It's straightforward to do this using `curses`, but `curses` is a fairly large module to learn. Here's a solution without `curses`:

```
import termios, fcntl, sys, os
fd = sys.stdin.fileno()

oldterm = termios.tcgetattr(fd)
newattr = termios.tcgetattr(fd)
newattr[3] = newattr[3] & ~termios.ICANON & ~termios.ECHO
termios.tcsetattr(fd, termios.TCSANOW, newattr)

oldflags = fcntl.fcntl(fd, fcntl.F_GETFL)
fcntl.fcntl(fd, fcntl.F_SETFL, oldflags | os.O_NONBLOCK)

try:
    while 1:
        try:
            c = sys.stdin.read(1)
            print "Got character", repr(c)
        except IOError: pass
finally:
    termios.tcsetattr(fd, termios.TCSAFLUSH, oldterm)
    fcntl.fcntl(fd, fcntl.F_SETFL, oldflags)
```

You need the `termios` and the `fcntl` module for any of this to work, and I've only tried it on Linux, though it should work elsewhere. In this code, characters are read and printed one at a time.

`termios.tcsetattr()` turns off `stdin`'s echoing and disables canonical mode. `fcntl.fcntl()` is used to obtain `stdin`'s file descriptor flags and modify them for non-blocking mode. Since reading `stdin` when it is empty results in an `IOError`, this error is caught and ignored.

## 4.3 Threads

### 4.3.1 Como faço para programar usando threads?

Be sure to use the `threading` module and not the `thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `thread` module.

Aahz tem um conjunto de tutoriais `threading` que são úteis; veja em: <http://www.pythoncraft.com/OSCON2001/>.

### 4.3.2 Nenhuma de minhas threads parece funcionar, por quê?

Assim que a thread principal acaba, todas as threads são eliminadas. Sua thread principal está sendo executada tão rápida que não está dando tempo para realizar qualquer trabalho.

Uma solução simples é adicionar um tempo de espera no final do programa até que todos os threads sejam concluídos:

```
import threading, time

def thread_task(name, n):
    for i in range(n): print name, i

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10) # <-----!

```

But now (on many platforms) the threads don't run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn't start a new thread until the previous thread is blocked.

Uma solução simples é adicionar um pequeno tempo de espera no início da função:

```
def thread_task(name, n):
    time.sleep(0.001) # <-----!
    for i in range(n): print name, i

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)

```

Instead of trying to guess a good delay value for `time.sleep()`, it's better to use some kind of semaphore mechanism. One idea is to use the `Queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

### 4.3.3 How do I parcel out work among a bunch of worker threads?

Use the `Queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Aqui está um exemplo simples:

```
import threading, Queue, time

# The worker thread gets jobs off the queue. When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print 'Running worker'
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except Queue.Empty:

```

(continua na próxima página)

(continuação da página anterior)

```

        print 'Worker', threading.currentThread(),
        print 'queue empty'
        break
    else:
        print 'Worker', threading.currentThread(),
        print 'running with argument', arg
        time.sleep(0.5)

# Create queue
q = Queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print 'Main thread sleeping'
time.sleep(5)

```

Quando executado, isso produzirá a seguinte saída:

```

Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started)> running with argument 0
Worker <Thread(worker 2, started)> running with argument 1
Worker <Thread(worker 3, started)> running with argument 2
Worker <Thread(worker 4, started)> running with argument 3
Worker <Thread(worker 5, started)> running with argument 4
Worker <Thread(worker 1, started)> running with argument 5
...

```

Consult the module's documentation for more details; the `Queue` class provides a featureful interface.

#### 4.3.4 Que tipos de variáveis globais mutáveis são seguras para thread?

A *global interpreter lock* (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setcheckinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that “look atomic” really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):



```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

Esses não são:

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

### 4.3.5 Não podemos remover o Bloqueio Global do interpretador?

The *global interpreter lock* (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the “free threading” patches) that removed the GIL and replaced it with fine-grained locking. Unfortunately, even on Windows (where locks are very efficient) this ran ordinary Python code about twice as slow as the interpreter using the GIL. On Linux the performance loss was even worse because pthread locks aren't as efficient.

Since then, the idea of getting rid of the GIL has occasionally come up but nobody has found a way to deal with the expected slowdown, and users who don't use threads would not be happy if their code ran at half the speed. Greg's free threading patch set has not been kept up-to-date for later Python versions.

This doesn't mean that you can't make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done.

It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn't be able to share objects. Unfortunately, this isn't likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.

And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.

E finalmente, uma vez que você tem vários interpretadores que não compartilham seu estado, o que você ganhou ao executar processos separados em cada interpretador?

## 4.4 Entrada e saída

### 4.4.1 Como faço para excluir um arquivo? (E outras perguntas sobre arquivos)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don't exist. `os.removedirs(path)` will remove intermediate directories as long as they're empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

Para renomear um arquivos, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "r+")`, and use `f.truncate(offset)`; `offset` defaults to the current seek position. There's also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where `fd` is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.

### 4.4.2 Como eu copio um arquivo?

The `shutil` module contains a `copyfile()` function. Note that on MacOS 9 it doesn't copy the resource fork and Finder info.

### 4.4.3 Como leio (ou escrevo) dados binários?

To read or write complex binary data formats, it's best to use the `struct` module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```
import struct

f = open(filename, "rb")  # Open in binary mode for portability
s = f.read(8)
x, y, z = struct.unpack(">hhl", s)
```

The `'>'` in the format string forces big-endian data; the letter `'h'` reads one “short integer” (2 bytes), and `'l'` reads one “long integer” (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the `array` module.

### 4.4.4 Porque não consigo usar os `os.read()` em um encadeamento com `os.popen()`;

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()` creates a high-level file object, the same type returned by the built-in `open()` function. Thus, to read `n` bytes from a pipe `p` created with `os.popen()`, you need to use `p.read(n)`.

#### 4.4.5 How do I run a subprocess with pipes connected to both input and output?

Use the `popen2` module. For example:

```
import popen2
fromchild, tochild = popen2.popen2("command")
tochild.write("input\n")
tochild.flush()
output = fromchild.readline()
```

Warning: in general it is unwise to do this because you can easily cause a deadlock where your process is blocked waiting for output from the child while the child is blocked waiting for input from you. This can be caused by the parent expecting the child to output more text than it does or by data being stuck in stdio buffers due to lack of flushing. The Python parent can of course explicitly flush the data it sends to the child before it reads any output, but if the child is a naive C program it may have been written to never explicitly flush its output, even if it is interactive, since flushing is normally automatic.

Note that a deadlock is also possible if you use `popen3()` to read stdout and stderr. If one of the two is too large for the internal buffer (increasing the buffer size does not help) and you `read()` the other one first, there is a deadlock, too.

Note on a bug in `popen2`: unless your program calls `wait()` or `waitpid()`, finished child processes are never removed, and eventually calls to `popen2` will fail because of a limit on the number of child processes. Calling `os.waitpid()` with the `os.WNOHANG` option can prevent this; a good place to insert such a call would be before calling `popen2` again.

In many cases, all you really need is to run some data through a command and get the result back. Unless the amount of data is very large, the easiest way to do this is to write it to a temporary file and run the command with that temporary file as input. The standard module `tempfile` exports a `mktemp()` function to generate unique temporary file names.

```
import tempfile
import os

class Popen3:
    """
    This is a deadlock-safe version of popen that returns
    an object with errorlevel, out (a string) and err (a string).
    (capturestderr may not work under windows.)
    Example: print Popen3('grep spam', '\n\nhere spam\n\n').out
    """
    def __init__(self, command, input=None, capturestderr=None):
        outfile=tempfile.mktemp()
        command="( %s ) > %s" % (command,outfile)
        if input:
            infile=tempfile.mktemp()
            open(infile,"w").write(input)
            command=command+" <"+infile
        if capturestderr:
            errfile=tempfile.mktemp()
            command=command+" 2>"+errfile
        self.errorlevel=os.system(command) >> 8
        self.out=open(outfile,"r").read()
        os.remove(outfile)
        if input:
            os.remove(infile)
        if capturestderr:
            self.err=open(errfile,"r").read()
            os.remove(errfile)
```

Note that many interactive programs (e.g. `vi`) don't work well with pipes substituted for standard input and output. You will have to use pseudo ttys ("ptys") instead of pipes. Or you can use a Python interface to Don Libes' "expect" library.

A Python extension that interfaces to expect is called “expy” and available from <http://expectpy.sourceforge.net>. A pure Python solution that works like expect is [pexpect](#).

### 4.4.6 Como acesso a porta serial (RS232)?

Para Win32, POSIX (Linux, BSD, etc.), Jython:

<http://pyserial.sourceforge.net>

Para Unix, veja uma postagem da Usenet de Mitch Chapman:

<https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com>

### 4.4.7 Por que o `sys.stdout` (`stdin`, `stderr`) não fecha?

Python file objects are a high-level layer of abstraction on top of C streams, which in turn are a medium-level layer of abstraction on top of (among other things) low-level C file descriptors.

For most file objects you create in Python via the built-in `file` constructor, `f.close()` marks the Python file object as being closed from Python’s point of view, and also arranges to close the underlying C stream. This also happens automatically in `f`’s destructor, when `f` becomes garbage.

But `stdin`, `stdout` and `stderr` are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C stream.

To close the underlying C stream for one of these three, you should first be sure that’s what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use `os.close`:

```
os.close(0)    # close C's stdin stream
os.close(1)    # close C's stdout stream
os.close(2)    # close C's stderr stream
```

## 4.5 Programação Rede / Internet

### 4.5.1 Quais ferramentas WWW existem no Python?

See the chapters titled `internet` and `netdata` in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

Um resumo dos frameworks disponíveis é disponibilizado por Paul Boddie em <https://wiki.python.org/moin/WebProgramming>.

Cameron Laird maintains a useful set of pages about Python web technologies at [http://phaseit.net/claird/comp.lang.python/web\\_python](http://phaseit.net/claird/comp.lang.python/web_python).

### 4.5.2 Como faço para imitar a submissão de formulário CGI (METHOD=POST)?

Gostaria de recuperar páginas da WEB resultantes de um formulário POST. Existe algum código que consigo fazer isso facilmente?

Yes. Here's a simple example that uses `httplib`:

```
#!/usr/local/bin/python

import httplib, sys, time

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
httpobj = httplib.HTTP('www.some-server.out-there', 80)
httpobj.putrequest('POST', '/cgi-bin/some-cgi-script')
# now generate the rest of the HTTP headers...
httpobj.putheader('Accept', '/*/*')
httpobj.putheader('Connection', 'Keep-Alive')
httpobj.putheader('Content-type', 'application/x-www-form-urlencoded')
httpobj.putheader('Content-length', '%d' % len(qs))
httpobj.endheaders()
httpobj.send(qs)
# find out what the server said in response...
reply, msg, hdrs = httpobj.getreply()
if reply != 200:
    sys.stdout.write(httpobj.getfile().read())
```

Note that in general for percent-encoded POST operations, query strings must be quoted using `urllib.urlencode()`. For example, to send `name=Guy Steele, Jr.`:

```
>>> import urllib
>>> urllib.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

### 4.5.3 Qual módulo devo usar para ajudar na geração do HTML?

Você pode encontrar uma coleção de links úteis na página wiki de programação da Web <<https://wiki.python.org/moin/WebProgramming>>‘\_.

### 4.5.4 Como envio um e-mail de um script Python?

Use a biblioteca padrão do módulo `smtplib`.

Aqui está um remetente de e-mail interativo muito simples. Este método funcionará em qualquer host que suporte o protocolo SMTP.

```
import sys, smtplib

fromaddr = raw_input("From: ")
toaddrs = raw_input("To: ").split(',')
print "Enter message, end with ^D:"
msg = ''
while True:
```

(continua na próxima página)

(continuação da página anterior)

```
line = sys.stdin.readline()
if not line:
    break
msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

A Unix-only alternative uses `sendmail`. The location of the `sendmail` program varies between systems; sometimes it is `/usr/lib/sendmail`, sometimes `/usr/sbin/sendmail`. The `sendmail` manual page will help you out. Here's some sample code:

```
import os

SENDMAIL = "/usr/sbin/sendmail" # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n") # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print "Sendmail exit status", sts
```

## 4.5.5 Como evito o bloqueio no método `connect()` de um soquete?

The `select` module is commonly used to help with asynchronous I/O on sockets.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno`. `errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the `errno` value. To poll, you can call `connect_ex()` again later – 0 or `errno.EISCONN` indicate that you're connected – or you can pass this socket to `select` to check if it's writable.

## 4.6 Base de Dados

### 4.6.1 Existem interfaces para banco de dados em Python?

Sim.

Python 2.3 includes the `bsddb` package which provides an interface to the BerkeleyDB library. Interfaces to disk-based hashes such as `DBM` and `GDBM` are also included with standard Python.

Suporte para a maioria dos bancos de dados relacionais está disponível. Para mais detalhes, veja a página wiki de programação de banco de dados em <https://wiki.python.org/moin/DatabaseProgramming>

### 4.6.2 Como você implementa objetos persistentes no Python?

The `pickle` library module solves this in a very general way (though you still can't store things like open files, sockets or windows), and the `shelve` library module uses `pickle` and `(g)dbm` to create persistent mappings containing arbitrary Python objects. For better performance, you can use the `cPickle` module.

A more awkward way of doing things is to use `pickle`'s little sister, `marshal`. The `marshal` module provides very fast ways to store noncircular basic Python types to files and strings, and back again. Although `marshal` does not do fancy things like store instances or handle shared references properly, it does run extremely fast. For example, loading a half megabyte of data may take less than a third of a second. This often beats doing something more complex and general such as using `gdbm` with `pickle/shelve`.

### 4.6.3 Why is cPickle so slow?

By default `pickle` uses a relatively old and slow format for backward compatibility. You can however specify other protocol versions that are faster:

```
largeString = 'z' * (100 * 1024)
myPickle = cPickle.dumps(largeString, protocol=1)
```

### 4.6.4 If my program crashes with a bsddb (or anydbm) database open, it gets corrupted. How come?

Databases opened for write access with the `bsddb` module (and often by the `anydbm` module, since it will preferentially use `bsddb`) must explicitly be closed using the `.close()` method of the database. The underlying library caches database contents which need to be converted to on-disk form and written.

If you have initialized a new `bsddb` database but not written anything to it before the program crashes, you will often wind up with a zero-length file and encounter an exception the next time the file is opened.

### 4.6.5 I tried to open Berkeley DB file, but bsddb produces bsddb.error: (22, 'Invalid argument'). Help! How can I restore my data?

Don't panic! Your data is probably intact. The most frequent cause for the error is that you tried to open an earlier Berkeley DB file with a later version of the Berkeley DB library.

Many Linux systems now have all three versions of Berkeley DB available. If you are migrating from version 1 to a newer version use `db_dump185` to dump a plain text version of the database. If you are migrating from version 2 to version 3 use `db2_dump` to create a plain text version of the database. In either case, use `db_load` to create a new native database for the latest version installed on your computer. If you have version 3 of Berkeley DB installed, you should be able to use `db2_load` to create a native version 2 database.

You should move away from Berkeley DB version 1 files because the hash file code contains known bugs that can corrupt your data.

## 4.7 Matemáticos e Numéricos

### 4.7.1 Como gero número aleatórios no Python?

O módulo padrão `random` implementa um gerador de números aleatórios. O uso é simples:

```
import random
random.random()
```

Isso retorna um número flutuante aleatório no intervalo  $[0, 1)$ .

Existem também muitos outros geradores aleatórios neste módulo, como:

- `randrange(a, b)` escolhe um número inteiro no intervalo entre  $[a, b)$ .
- `uniform(a, b)` escolhe um número float no intervalo  $[a, b)$ .
- `normalvariate(mean, sdev)` samples the normal (Gaussian) distribution.

Algumas funções de nível elevado operam diretamente em sequência, como:

- `choice(S)` escolhe um elemento aleatório de uma determinada sequência
- `shuffle(L)` shuffles a list in-place, i.e. permutes it randomly

Existe também uma classe `Random` que você pode instanciar para criar vários geradores de números aleatórios independentes.



---

## FAQ sobre Extensão/Incorporação

---

### 5.1 Posso criar minhas próprias funções em C?

Sim, você pode construir módulos embutidos contendo funções, variáveis, exceções e até mesmo novos tipos em C. Isso é explicado no documento `extending-index`.

A maioria dos livros intermediários ou avançados em Python também abordará esse tópico.

### 5.2 Can I create my own functions in C++?

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.

### 5.3 Writing C is hard; are there any alternatives?

There are a number of alternatives to writing your own C extensions, depending on what you're trying to do.

If you need more speed, [Psyco](#) generates x86 assembly code from Python bytecode. You can use Psyco to compile the most time-critical functions in your code, and gain a significant improvement with very little effort, as long as you're running on a machine with an x86-compatible processor.

[Cython](#) and its relative [Pyrex](#) are compilers that accept a slightly modified form of Python and generate the corresponding C code. Pyrex makes it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as [SWIG](#). [SIP](#), [CXX Boost](#), or [Weave](#) are also alternatives for wrapping C++ libraries.

## 5.4 How can I execute arbitrary Python statements from C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns 0 for success and -1 when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

## 5.5 How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

## 5.6 How do I extract C values from a Python object?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For strings, `PyString_Size()` returns its length and `PyString_AsString()` a pointer to its value. Note that Python strings may contain null bytes so C's `strlen()` should not be used.

To test the type of an object, first make sure it isn't `NULL`, and then use `PyString_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc.) as well as many other useful protocols.

## 5.7 How do I use `Py_BuildValue()` to create a tuple of arbitrary length?

You can't. Use `t = PyTuple_New(n)` instead, and fill it with objects using `PyTuple_SetItem(t, i, o)` – note that this “eats” a reference count of `o`, so you have to `Py_INCREF()` it. Lists have similar functions `PyList_New(n)` and `PyList_SetItem(l, i, o)`. Note that you *must* set all the tuple items to some value before you pass the tuple to Python code – `PyTuple_New(n)` initializes them to `NULL`, which isn't a valid Python value.

## 5.8 How do I call an object's method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```
PyObject *
PyObject_CallMethod(PyObject *object, char *method_name,
                   char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`ing the return value.

To call, e.g., a file object's “seek” method with arguments 10, 0 (assuming the file object pointer is “f”):

```

res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
}
else {
    Py_DECREF(res);
}

```

Note that since `PyObject_CallObject()` *always* wants a tuple for the argument list, to call a function without arguments, pass “()” for the format, and to call a function with one argument, surround the argument in parentheses, e.g. “(i)”.

## 5.9 How do I catch the output from `PyErr_Print()` (or anything that prints to `stdout/stderr`)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call `print_error`, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the `StringIO` class in the standard library.

Sample code and use for catching `stdout`:

```

>>> class StdoutCatcher:
...     def __init__(self):
...         self.data = ''
...     def write(self, stuff):
...         self.data = self.data + stuff
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print 'foo'
>>> print 'hello world!'
>>> sys.stderr.write(sys.stdout.data)
foo
hello world!

```

## 5.10 How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

```
module = PyImport_ImportModule("<module name>");
```

If the module hasn't been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules["<module name>"]`. Note that it doesn't enter the module into any namespace – it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "<attr name>");
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.

## 5.11 How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the “Extending and Embedding” document. Realize that for the Python run-time system, there isn’t a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see *Writing C is hard; are there any alternatives?*.

## 5.12 I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn’t seem worth the effort.)

## 5.13 How do I debug an extension?

When using GDB with dynamically loaded extensions, you can’t set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

## 5.14 I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python don’t include the `/usr/lib/python2.x/config/` directory, which contains various files required for compiling Python extensions.

For Red Hat, install the `python-devel` RPM to get the necessary files.

For Debian, run `apt-get install python-dev`.

## 5.15 What does “SystemError: \_PyImport\_FixupExtension: module yourmodule not loaded” mean?

This means that you have created an extension module named “yourmodule”, but your module init function does not initialize with that name.

Every module init function will have a line similar to:

```
module = Py_InitModule("yourmodule", yourmodule_functions);
```

If the string passed to this function is not the same name as your extension module, the `SystemError` exception will be raised.

## 5.16 How do I tell “incomplete input” from “invalid input”?

Sometimes you want to emulate the Python interactive interpreter’s behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an “if” statement or you didn’t close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the `codeop` module, which approximates the parser’s behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call `PyRun_InteractiveLoop()` (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the `PyOS_ReadlineFunctionPointer()` to point at your custom input function. See `Modules/readline.c` and `Parser/myreadline.c` for more hints.

However sometimes you have to run the embedded Python interpreter in the same thread as your rest application and you can’t allow the `PyRun_InteractiveLoop()` to stop while waiting for user input. The one solution then is to call `PyParser_ParseString()` and test for `e.error` equal to `E_EOF`, which means the input is incomplete. Here’s a sample code fragment, untested, inspired by code from Alex Farber:

```
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>

int testcomplete(char *code)
/* code should end in \n */
/* return -1 for error, 0 for incomplete, 1 for complete */
{
    node *n;
    perrdetail e;

    n = PyParser_ParseString(code, &_PyParser_Grammar,
                             Py_file_input, &e);

    if (n == NULL) {
        if (e.error == E_EOF)
            return 0;
        return -1;
    }

    PyNode_Free(n);
}
```

(continua na próxima página)

(continuação da página anterior)

```

return 1;
}

```

Another solution is trying to compile the received string with `Py_CompileString()`. If it compiles without errors, try to execute the returned code object by calling `PyEval_EvalCode()`. Otherwise save the input for later. If the compilation fails, find out if it's an error or just more input is required - by extracting the message string from the exception tuple and comparing it to the string "unexpected EOF while parsing". Here is a complete example using the GNU readline library (you may want to ignore **SIGINT** while calling `readline()`):

```

#include <stdio.h>
#include <readline.h>

#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[])
{
    int i, j, done = 0;                                /* lengths of line, code */
    char ps1[] = ">>> ";
    char ps2[] = "... ";
    char *prompt = ps1;
    char *msg, *line, *code = NULL;
    PyObject *src, *glb, *loc;
    PyObject *exc, *val, *trb, *obj, *dum;

    Py_Initialize ();
    loc = PyDict_New ();
    glb = PyDict_New ();
    PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

    while (!done)
    {
        line = readline (prompt);

        if (NULL == line)                                /* Ctrl-D pressed */
        {
            done = 1;
        }
        else
        {
            i = strlen (line);

            if (i > 0)
                add_history (line);                        /* save non-empty lines */

            if (NULL == code)                            /* nothing in code yet */
                j = 0;
            else
                j = strlen (code);

            code = realloc (code, i + j + 2);
            if (NULL == code)                            /* out of memory */
                exit (1);
        }
    }
}

```

(continua na próxima página)

(continuação da página anterior)

```

if (0 == j)                                /* code was empty, so */
    code[0] = '\0';                        /* keep strncat happy */

strncat (code, line, i);                   /* append line to code */
code[i + j] = '\n';                       /* append '\n' to code */
code[i + j + 1] = '\0';

src = Py_CompileString (code, "<stdin>", Py_single_input);

if (NULL != src)                           /* compiled just fine - */
{
    if (ps1 == prompt ||                   /* ">>> " or */
        '\n' == code[i + j - 1])          /* "... " and double '\n' */
    {                                       /* so execute it */
        dum = PyEval_EvalCode ((PyCodeObject *)src, glb, loc);
        Py_XDECREF (dum);
        Py_XDECREF (src);
        free (code);
        code = NULL;
        if (PyErr_Occurred ())
            PyErr_Print ();
        prompt = ps1;
    }
}                                           /* syntax error or E_EOF? */
else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
{
    PyErr_Fetch (&exc, &val, &trb);      /* clears exception! */

    if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
        !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
    {
        Py_XDECREF (exc);
        Py_XDECREF (val);
        Py_XDECREF (trb);
        prompt = ps2;
    }
    else                                   /* some other syntax error */
    {
        PyErr_Restore (exc, val, trb);
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
    }
}
else                                     /* some non-syntax error */
{
    PyErr_Print ();
    free (code);
    code = NULL;
    prompt = ps1;
}

free (line);
}
}

```

(continua na próxima página)

(continuação da página anterior)

```
Py_XDECREF(glb);
Py_XDECREF(loc);
Py_Finalize();
exit(0);
}
```

## 5.17 How do I find undefined g++ symbols `__builtin_new` or `__pure_virtual`?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change `LINKCC` in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).

## 5.18 Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as `int`, `list`, `dict`, etc.

The Boost Python Library (BPL, <http://www.boost.org/libs/python/doc/index.html>) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).

## 5.19 When importing module X, why do I get “undefined symbol: PyUnicodeUCS2\*”?

You are using a version of Python that uses a 4-byte representation for Unicode characters, but some C extension module you are importing was compiled using a Python that uses a 2-byte representation for Unicode characters (the default).

If instead the name of the undefined symbol starts with `PyUnicodeUCS4`, the problem is the reverse: Python was built using 2-byte Unicode characters, and the extension module was compiled using a Python with 4-byte Unicode characters.

This can easily occur when using pre-built extension packages. RedHat Linux 7.x, in particular, provided a “python2” binary that is compiled with 4-byte Unicode. This only causes the link failure if the extension uses any of the `PyUnicode_*()` functions. It is also a problem if an extension uses any of the Unicode-related format specifiers for `Py_BuildValue()` (or similar) or parameter specifications for `PyArg_ParseTuple()`.

You can check the size of the Unicode character a Python interpreter is using by checking the value of `sys.maxunicode`:

```
>>> import sys
>>> if sys.maxunicode > 65535:
...     print 'UCS4 build'
... else:
...     print 'UCS2 build'
```

The only way to solve this problem is to use extension modules compiled with a Python binary built using the same size for Unicode characters.



---

Python no Windows

---

## 6.1 Como faço para executar um programa Python no Windows?

Esta não é necessariamente uma questão direta. Se você já está familiarizado com a execução de programas através das linha de comando do Windows, então tudo parecerá óbvio; caso contrário, poderá precisar de um pouco mais de orientação.

Unless you use some sort of integrated development environment, you will end up *typing* Windows commands into what is variously referred to as a “DOS window” or “Command prompt window”. Usually you can create such a window from your Start menu; under Windows 7 the menu selection is *Start ▶ Programs ▶ Accessories ▶ Command Prompt*. You should be able to recognize when you have started such a window because you will see a Windows “command prompt”, which usually looks like this:

```
C:\>
```

The letter may be different, and there might be other things after it, so you might just as easily see something like:

```
D:\YourName\Projects\Python>
```

dependendo de como seu computador foi configurado e o que mais você tem feito com ele recentemente. Uma vez que você tenha iniciado a janela, você estará no caminho para executar os seus programas Python.

Você deve observar que seu código Python deve ser processado por outro programa chamado interpretador. O interpretador lê o seu código, compila em bytecodes, e depois executa os bytecodes para rodar o seu programa. Então, como você pode organizar o interpretador para lidar com seu Python?

First, you need to make sure that your command window recognises the word “python” as an instruction to start the interpreter. If you have opened a command window, you should try entering the command `python` and hitting return.:

```
C:\Users\YourName> python
```

You should then see something like:

```
Python 2.7.3 (default, Apr 10 2012, 22:71:26) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You have started the interpreter in “interactive mode”. That means you can enter Python statements or expressions interactively and have them executed or evaluated while you wait. This is one of Python’s strongest features. Check it by entering a few expressions of your choice and seeing the results:

```
>>> print "Hello"
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Many people use the interactive mode as a convenient yet highly programmable calculator. When you want to end your interactive Python session, hold the `Ctrl` key down while you enter a `Z`, then hit the “Enter” key to get back to your Windows command prompt.

You may also find that you have a Start-menu entry such as *Start ▶ Programs ▶ Python 2.7 ▶ Python (command line)* that results in you seeing the `>>>` prompt in a new window. If so, the window will disappear after you enter the `Ctrl-Z` character; Windows is running a single “python” command in the window, and closes it when you terminate the interpreter.

If the `python` command, instead of displaying the interpreter prompt `>>>`, gives you a message like:

```
'python' is not recognized as an internal or external command,
operable program or batch file.
```

or:

```
Bad command or filename
```

then you need to make sure that your computer knows where to find the Python interpreter. To do this you will have to modify a setting called `PATH`, which is a list of directories where Windows will look for programs.

You should arrange for Python’s installation directory to be added to the `PATH` of every command window as it starts. If you installed Python fairly recently then the command

```
dir C:\py*
```

will probably tell you where it is installed; the usual location is something like `C:\Python27`. Otherwise you will be reduced to a search of your whole disk ... use *Tools ▶ Find* or hit the *Search* button and look for “python.exe”. Supposing you discover that Python is installed in the `C:\Python27` directory (the default at the time of writing), you should make sure that entering the command

```
c:\Python27\python
```

starts up the interpreter as above (and don’t forget you’ll need a “`Ctrl-Z`” and an “Enter” to get out of it). Once you have verified the directory, you can add it to the system path to make it easier to start Python by just running the `python` command. This is currently an option in the installer as of CPython 2.7.

More information about environment variables can be found on the [Using Python on Windows](#) page.

## 6.2 Como eu faço para criar programas Python executáveis?

No Windows, o instalador padrão do Python já associa a extensão `.py` com o tipo de arquivo (`Python.File`) e dá àquele tipo de arquivo um comando aberto que executa o interpretador (`D:\Program Files\Python\python.exe "%1" %*`). Isso é o bastante para fazer scripts executáveis pelo prompt de comando como `'foo.py'`. Se você preferir executar o script simplesmente digitando `'foo'` sem extensão você precisa adicionar `.py` à variável de ambiente `PATHEXT`.

## 6.3 Por que Python às vezes demora tanto para iniciar?

Geralmente, Python inicia muito rapidamente no Windows, mas ocasionalmente há relatos de erros que, de repente, o Python começa a demorar muito tempo para iniciar. Isso é ainda mais intrigante, porque Python funciona bem em outros sistemas Windows que parecem estar configurados de forma idêntica.

O problema pode ser causado por uma desconfiguração de software antivírus na máquina problemática. Alguns antivírus são conhecidos por introduzir sobrecarga de duas ordens de magnitude no início quando estão configurados para monitorar todas as leituras do sistema de arquivos. Tente verificar a configuração do antivírus nos seus sistemas para assegurar que eles estão de fato configurados identicamente. O McAfee, quando configurado para escanear toda a atividade do sistema de arquivos, é um ofensor conhecido.

## 6.4 Como eu faço para criar um executável a partir de um código Python?

See <http://www.py2exe.org/> for a distutils extension that allows you to create console and GUI executables from Python code.

## 6.5 Um arquivo `*.pyd` é o mesmo que um DLL?

Yes, `.pyd` files are dll's, but there are a few differences. If you have a DLL named `foo.pyd`, then it must have a function `initfoo()`. You can then write Python `"import foo"`, and Python will search for `foo.pyd` (as well as `foo.py`, `foo.pyc`) and if it finds it, will attempt to call `initfoo()` to initialize it. You do not link your `.exe` with `foo.lib`, as that would cause Windows to require the DLL to be present.

Observe que o caminho de pesquisa para `foo.pyd` é `PYTHONPATH`, não o mesmo que o Windows usa para procurar por `foo.dll`. Além disso, `foo.pyd` não precisa estar presente para executar seu programa, enquanto que se você vinculou seu programa a uma dll, a dll será necessária. Obviamente, o `foo.pyd` é necessário se você quiser dizer `import foo`. Em uma DLL, o vínculo é declarado no código-fonte com `__declspec(dllexport)`. Em um `.pyd`, o vínculo é definido em uma lista de funções disponíveis.

## 6.6 Como eu posso embutir Python dentro de uma aplicação do Windows?

A incorporação do interpretador Python em um aplicativo do Windows pode ser resumida da seguinte forma:

1. Do `_not_` build Python into your `.exe` file directly. On Windows, Python must be a DLL to handle importing modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to `pythonNN.dll`; it is typically installed in `C:\Windows\System`. `NN` is the Python version, a number such as "27" for Python 2.7.

Você pode vincular ao Python de duas maneiras diferentes. A vinculação em tempo de carregamento significa vincular contra `pythonNN.lib`, enquanto a vinculação em tempo de execução significa vincular a `pythonNN.dll`. (Nota geral: `pythonNN.lib` é a chamada "import lib" correspondente a `pythonNN.dll`. Apenas define símbolos para o vinculador.)

A vinculação em tempo de execução simplifica bastante as opções de vinculação; tudo acontece em tempo de execução. Seu código deve carregar `pythonNN.dll` usando a rotina `LoadLibraryEx()` do Windows. O código também deve usar rotinas de acesso e dados em `pythonNN.dll` (ou seja, as APIs C do Python) usando ponteiros obtidos pela rotina `GetProcAddress()` do Windows. As macros podem tornar o uso desses ponteiros transparente para qualquer código C que chama rotinas na API C do Python.

Nota da Borland: converter `pythonNN.lib` ao formato OMF usando `Coff2Omf.exe` primeiro.

2. Se você usa SWIG, é fácil criar um "módulo de extensão" do Python que disponibilizará os dados e os métodos da aplicação para o Python. O SWIG cuidará de todos os detalhes obscuros para você. O resultado é o código C que você vincula ao arquivo `.exe` (!) Você não precisa criar um arquivo DLL, o que também simplifica a vinculação.
3. O SWIG criará uma função `init` (uma função C) cujo nome depende do nome do módulo de extensão. Por exemplo, se o nome do módulo for `leo`, a função `init` será chamada `initleo()`. Se você usa classes de sombra SWIG, como deveria, a função `init` será chamada `initleooc()`. Isso inicializa uma classe auxiliar principalmente oculta usada pela classe `shadow`.

O motivo pelo qual você pode vincular o código C na etapa 2 ao seu arquivo `.exe` é que chamar a função de inicialização equivale a importar o módulo para o Python! (Este é o segundo fato chave não documentado.)

4. Em suma, você pode utilizar o código a seguir para inicializar o interpretador Python com seu módulo de extensão.

```
#include "python.h"
...
Py_Initialize(); // Initialize Python.
initmyApp(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

5. Existem dois problemas com a API C do Python que se tornarão aparentes se você utiliza um compilador que não seja o MSVC, o compilador utilizado no `pythonNN.dll`.

Problema 1: As chamadas funções de "Nível Muito Alto" que recebem argumentos `FILE *` não funcionarão em um ambiente com vários compiladores porque a noção de cada struct `FILE` de um compilador será diferente. Do ponto de vista da implementação, essas são funções de nível muito baixo.

Problema 2: SWIG gera o seguinte código ao gerar envólucros para funções sem retorno:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Infelizmente, `Py_None` é uma macro que se expande para uma referência a uma estrutura de dados complexa chamada `_Py_NoneStruct` dentro de `pythonNN.dll`. Novamente, esse código falhará em um ambiente com vários compiladores. Substitua esse código por:

```
return Py_BuildValue("");
```

Pode ser possível usar o comando `%typemap` do SWIG para fazer a alteração automaticamente, embora eu não tenha conseguido fazer isso funcionar (eu sou um completo novato em SWIG).

6. Usar um script de shell do Python para criar uma janela do interpretador Python de dentro da aplicação do Windows não é uma boa ideia; a janela resultante será independente do sistema de janelas da sua aplicação. Em vez disso, você (ou a classe `wxPythonWindow`) deve criar uma janela “nativa” do interpretador. É fácil conectar essa janela ao interpretador Python. Você pode redirecionar a E/S do Python para qualquer objeto que suporte leitura e gravação; portanto, tudo que você precisa é de um objeto Python (definido no seu módulo de extensão) que contenha métodos `read()` e `write()`.

## 6.7 Como eu impeço editores de adicionarem tabulações na minha source do Python?

As perguntas frequentes não recomendam a utilização de tabulações, e o guia de estilo Python, :pep:8, recomenda 4 espaços para código de Python distribuído; esse também é o padrão do python-mode do Emacs.

Sob qualquer editor, misturar tabulações e espaços é uma má ideia. O MSVC não é diferente nesse aspecto e é facilmente configurado para usar espaços: Selecione *Tools ▸ Options ▸ Tabs* e, para o tipo de arquivo “Default”, defina “Tab size” e “Indent size” para 4 e selecione o botão de opção “Insert spaces”.

If you suspect mixed tabs and spaces are causing problems in leading whitespace, run Python with the `-t` switch or run the `tabnanny` module to check a directory tree in batch mode.

## 6.8 Como faço para verificar uma tecla pressionada sem bloquear?

Use o módulo `msvcrt`. Este é um módulo padrão de extensão específico do Windows. Ele define uma função “`kbhit()`” que verifica se um toque no teclado está presente, e “`getch()`” que recebe um caractere sem exibi-lo.

## 6.9 How do I emulate `os.kill()` in Windows?

Prior to Python 2.7 and 3.2, to terminate a process, you can use `ctypes`:

```
import ctypes

def kill(pid):
    """kill function for Win32"""
    kernel32 = ctypes.windll.kernel32
    handle = kernel32.OpenProcess(1, 0, pid)
    return (0 != kernel32.TerminateProcess(handle, 0))
```

In 2.7 and 3.2, `os.kill()` is implemented similar to the above function, with the additional feature of being able to send `Ctrl+C` and `Ctrl+Break` to console subprocesses which are designed to handle those signals. See `os.kill()` for further details.

## 6.10 How do I extract the downloaded documentation on Windows?

Sometimes, when you download the documentation package to a Windows machine using a web browser, the file extension of the saved file ends up being .EXE. This is a mistake; the extension should be .TGZ.

Simply rename the downloaded file to have the .TGZ extension, and WinZip will be able to handle it. (If your copy of WinZip doesn't, get a newer one from <https://www.winzip.com>.)

---

## FAQ da Interface Gráfica do Usuário

---

### 7.1 Que ferramentas de GUI independentes da plataforma existem para o Python?

Depending on what platform(s) you are aiming at, there are several.

#### 7.1.1 Tkinter

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called Tkinter. This is probably the easiest to install and use. For more info about Tk, including pointers to the source, see the Tcl/Tk home page at <https://www.tcl.tk>. Tcl/Tk is fully portable to the Mac OS X, Windows, and Unix platforms.

#### 7.1.2 wxWidgets

WxWidgets (<https://www.wxwidgets.org>) é uma biblioteca de classe GUI livre e portátil escrita em C++ que fornece uma aparência e sensação nativas em várias plataformas, com Windows, Mac OS X, GTK, X11, todos listados como Metas estáveis atuais. As ligações de idiomas estão disponíveis para vários idiomas, incluindo Python, Perl, Ruby, etc.

wxPython (<http://www.wxpython.org>) is the Python binding for wxwidgets. While it often lags slightly behind the official wxWidgets releases, it also offers a number of features via pure Python extensions that are not available in other language bindings. There is an active wxPython user and developer community.

Ambos wxWidgets e wxPython são gratuitos, de código aberto, software com licenças permissivas que permitem seu uso em produtos comerciais, bem como em freeware ou shareware.

### 7.1.3 Qt

Existem ligações disponíveis para o kit de ferramentas Qt (usando PyQt <<https://riverbankcomputing.com/software/pyqt/intro>> \_ ou PySide <<https://wiki.qt.io/PySide>> \_) e Para o KDE (PyKDE4 <[https://techbase.kde.org/Languages/Python/Using\\_PyKDE\\_4](https://techbase.kde.org/Languages/Python/Using_PyKDE_4)> \_). O PyQt é atualmente mais maduro do que o PySide, mas você deve comprar uma licença PyQt da Riverbank Computing <<https://www.riverbankcomputing.com/commercial/license-faq>> \_ se você deseja escrever aplicativos proprietários. PySide é gratuito para todas as aplicações.

Qt 4.5 para cima é licenciado sob a licença LGPL; Além disso, as licenças comerciais estão disponíveis na *The Qt Company* <<https://www.qt.io/licensing/>> \_.

### 7.1.4 Gtk+

PyGtk bindings for the **Gtk+** toolkit have been implemented by James Henstridge; see <<http://www.pygtk.org>>.

### 7.1.5 FLTK

As ligações Python para o kit de ferramentas **FLTK**, um sistema de janelas multiplataforma simples, porém poderoso e maduro, estão disponíveis no projeto **PyFLTK**.

### 7.1.6 OpenGL

Para ligações OpenGL, veja **PyOpenGL** <<http://pyopengl.sourceforge.net>> \_.

## 7.2 Que kits de ferramentas GUI específicos da plataforma existem para o Python?

By installing the **PyObjc** Objective-C bridge, Python programs can use Mac OS X's Cocoa libraries.

*Pythonwin* de Mark Hammond inclui uma interface para o Microsoft Foundation Classes e um ambiente de programação Python que está escrito principalmente em Python usando as classes MFC.

## 7.3 Perguntas do Tkinter

### 7.3.1 Como eu pauso as aplicações Tkinter?

Freeze é uma ferramenta para criar aplicativos autônomos. Ao congelar aplicativos Tkinter, os aplicativos não serão verdadeiramente autônomos, pois o aplicativo ainda precisará das bibliotecas Tcl e Tk.

Uma solução é enviar o aplicativo com as bibliotecas Tcl e Tk e apontá-las em tempo de execução usando as variáveis de ambiente `TCL_LIBRARY` e `TK_LIBRARY`.

Para obter aplicativos verdadeiramente autônomos, os scripts Tcl que formam a biblioteca também precisam ser integrados no aplicativo. Uma ferramenta que suporta isso é SAM (módulos autônomos), que faz parte da distribuição Tix (<http://tix.sourceforge.net/>).

Compile o Tix com SAM habilitado, execute a chamada apropriada para `Tclsam_init()` etc. dentro do `Modules/tkappinit.c` do Python e faça um link com `libtclsam` e `libtkbam` (você também pode incluir as bibliotecas Tix).



### 7.3.2 Posso ter eventos Tk manipulados enquanto aguardo pelo E/S?

Em plataformas diferentes do Windows, sim, e você nem precisa de threads! Mas você terá que reestruturar seu código de E/S um pouco. O Tk tem o equivalente à chamada `XtAddInput()` do Xt, que permite que você registre uma função de retorno de chamada que será chamada a partir do loop principal do Tk quando E/S é possível em um descritor de arquivo. Consulte `tkinter-file-handlers`.

### 7.3.3 Não consigo obter ligações chave para trabalhar em Tkinter: por quê?

Uma queixa frequentemente ouvida é que os manipuladores de eventos vinculados a eventos com o método `bind()` não são manipulados mesmo quando a tecla apropriada é pressionada.

A causa mais comum é que o widget para o qual a ligação se aplica não possui “foco no teclado”. Confira a documentação do Tk para o comando de foco. Normalmente, um widget é dado o foco do teclado clicando nele (mas não para rótulos, veja a opção `takefocus`).



---

### FAD de “Por que o Python está instalado em meu computador?”

---

#### 8.1 O que é Python?

Python é uma linguagem de programação. É usada para muitas e diversas aplicações. É usada em escolas e faculdades como uma linguagem de programação introdutória, porque Python é fácil de aprender, mas também é usada por desenvolvedores profissionais de software em lugares como Google, Nasa e Lucasfilm Ltd.

Se você quiser aprender mais sobre Python, comece com o [Beginner's Guide to Python](#).

#### 8.2 Porque Python está instalado em minha máquina?

Se você encontrar Python instalado em seu sistema, mas não se lembra de tê-lo instalado, então podem haver muitas maneiras diferentes dele ter ido parar lá

- Possivelmente outro usuário do computador pretendia aprender programação e o instalou; você terá que descobrir quem estava usando a máquina e pode o ter instalado.
- A third-party application installed on the machine might have been written in Python and included a Python installation. For a home computer, the most common such application is [PySol](#), a solitaire game that includes over 1000 different games and variations.
- Algumas máquinas Windows já possuem o Python instalado. No presente momento nós temos conhecimento de computadores da Hewlett-Packard e da Compaq que incluem Python. Aparentemente algumas das ferramentas administrativas da HP/Compaq são escritas em Python.
- All Apple computers running Mac OS X have Python installed; it's included in the base installation.

## 8.3 Eu posso apagar o Python?

Isso depende de como o Python veio.

Se alguém o instalou deliberadamente, você pode removê-lo sem machucar ninguém. No Windows use o Adicionar ou remover programas que se encontra no Painel de Controle.

Se o Python foi instalado por um aplicativo de terceiros, você pode removê-lo mas aquela aplicação não irá mais funcionar. Você deve usar o desinstalador daquele aplicativo para remover o Python diretamente.

Se o Python veio junto com seu sistema operacional, removê-lo não é recomendado. Se você o remover, qualquer ferramenta que tenha sido escrita em Python não vão mais funcionar, e algumas delas podem ser importantes para você. A reinstalação do sistema inteiro seria necessária para consertar as coisas de novo.

>>> O prompt padrão do shell interativo do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

. . . The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

**2to3** Uma ferramenta que tenta converter código Python 2.x em código Python 3.x tratando a maioria das incompatibilidades que podem se detectadas com análise do código-fonte e navegação na árvore sintática.

O 2to3 está disponível na biblioteca padrão como `lib2to3`; um ponto de entrada é disponibilizado como `Tools/scripts/2to3`. Veja `2to3-reference`.

**classe base abstrata** Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

**argumento** A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `nome=`) na chamada de uma função ou passada como um valor em um dicionário precedido por `**`. Por exemplo, 3 e 5 são ambos argumentos nomeados na chamada da função `complex()` a seguir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional*: um argumento que não é um argumento nomeado. Argumentos posicionais podem aparecer no início da lista de argumentos e/ou podem ser passados com elementos de um *iterável* precedido por `*`. Por exemplo, 3 e 5 são ambos argumentos posicionais nas chamadas a seguir:

```
complex(3, 5)
complex(*(3, 5))
```

Argumentos são atribuídos às variáveis locais nomeadas no corpo da função. Veja a seção [calls](#) para as regras de atribuição. Sintaticamente, qualquer expressão pode ser usada para representar um argumento; avaliada a expressão, o valor é atribuído à variável local.

See also the [parameter](#) glossary entry and the FAQ question on [the difference between arguments and parameters](#).

**atributo** Um valor associado a um objeto que é referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

**BDFL** Benevolent Dictator For Life, a.k.a. [Guido van Rossum](#), Python's creator.

**objeto byte ou similar** An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

**bytecode** Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a [virtual machine](#) that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

Uma lista de instruções bytecode pode ser encontrada na documentação para o módulo `dis`.

**Classe** Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

**classic class** Any class which does not inherit from `object`. See [new-style class](#). Classic classes have been removed in Python 3.

**coerção** The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

**número complexo** Uma extensão ao familiar sistema de números reais em que todos os números são expressos como uma soma de uma parte real e uma parte imaginária. Números imaginários são múltiplos reais da unidade imaginária (a raiz quadrada de  $-1$ ), normalmente escrita como *i* em matemática ou *j* em engenharia. O Python tem suporte nativo para números complexos, que são escritos com esta última notação; a parte imaginária escrita com um sufixo *j*, p.ex., `3+1j`. Para ter acesso aos equivalentes para números complexos do módulo `math`, utilize `cmath`. O uso de números complexos é uma funcionalidade matemática bastante avançada. Se você não sabe se irá precisar deles, é quase certo que você pode ignorá-los sem problemas.

**gerenciador de contexto** Um objeto que controla o ambiente visto numa instrução `with` por meio da definição dos métodos `__enter__()` e `__exit__()`. Veja [PEP 343](#).

**CPython** A implementação canônica da linguagem de programação Python, como disponibilizada pelo [python.org](#). O termo “CPython” é quando for necessário distinguir esta implementação de outras como Jython ou IronPython.

**decorador** Uma função que retorna outra função, geralmente aplicada como uma transformação de função usando a sintaxe `@wrapper`. Exemplos comuns para decoradores são `classmethod()` e `staticmethod()`.

A sintaxe do decorador é meramente um açúcar-sintático, as duas definições de funções a seguir são semanticamente equivalentes:

```
def f(...):  
    ...  
f = staticmethod(f)
```

(continua na próxima página)

(continuação da página anterior)

```
@staticmethod
def f(...):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de function definitions e class definitions para obter mais informações sobre decoradores.

**descriptor** Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors.

**dicionário** An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

**visualização de dicionário** The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

**docstring** Uma string literal que aparece como primeira expressão numa classe, função ou módulo. Ainda que sejam ignoradas quando a suíte é executada, é reconhecida pelo compilador que a coloca no atributo `__doc__` da classe, função ou módulo que a encapsula. Como ficam disponíveis por meio de introspecção, docstrings são o lugar canônico para documentação do objeto.

**duck-typing (tipagem pato)** Um estilo de programação que não verifica o tipo do objeto para determinar se ele possui a interface correta; em vez disso, o método ou atributo é simplesmente chamado ou utilizado (“Se se parece com um pato e grasna como um pato, então deve ser um pato.”) Enfatizando interfaces ao invés de tipos específicos, o código bem desenvolvido aprimora sua flexibilidade por permitir substituição polimórfica. Tipagem pato evita necessidade de testes que usem `type()` ou `isinstance()`. (Note, porém, que a tipagem pato pode ser complementada com o uso de *classes base abstratas*.) Ao invés disso, são normalmente empregados testes `hasattr()` ou programação *EAFP*.

**EAFP** Iniciais da expressão em inglês “easier to ask for forgiveness than permission” que significa “é mais fácil pedir perdão que permissão”. Este estilo de codificação comum em Python assume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LYBL*, comum em outras linguagens como C, por exemplo.

**expressão** A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

**extension module (módulo de extensão)** Um módulo escrito em C ou C++, usando a API C de Python para interagir tanto com código de usuário quanto do núcleo.

**objeto arquivo** Um objeto que expõe uma API orientada a arquivos (com métodos tais como `read()` ou `write()`) para um recurso subjacente. Dependendo da maneira como foi criado, um objeto arquivo pode mediar o acesso a um arquivo real no disco ou outro tipo de dispositivo de armazenamento ou de comunicação (por exemplo a entrada/saída padrão, buffers em memória, soquetes, pipes, etc.). Objetos arquivo também são chamados de *file-like objects* ou *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

**objeto como-arquivo** Um sinônimo do termo *file object*.

**localizador** An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See [PEP 302](#) for details.

**divisão pelo piso** Divisão matemática que arredonda para baixo para o inteiro mais próximo. O operador de divisão pelo piso é `//`. Por exemplo, a expressão `11 // 4` retorna o valor 2 ao invés de `2.75`, que seria retornado pela divisão de ponto flutuante. Note que `(-11) // 4` é `-3` porque é `-2.75` arredondado *para baixo*. Consulte a [PEP 238](#).

**função** Uma série de instruções que retorna algum valor para um chamador. Também pode ser passado zero ou mais *argumentos* que podem ser usados na execução do corpo. Veja também *parâmetro*, *métodos* e a seção *function*.

**\_\_future\_\_** A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

**garbage collection (coletor de lixo)** The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

**gerador** A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**generator expression** An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

**GIL** Veja *global interpreter lock*.

**global interpreter lock (bloqueio global do interpretador)** O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos internos críticos como o `dict`) ganhem segurança implícita contra acesso concorrente. Travar todo o interpretador facilita que o interpretador em si seja multitarefa, às custas de muito do paralelismo já provido por máquinas multiprocessador.

No entanto, alguns módulos de extensão, tanto da biblioteca padrão quanto de terceiros, são desenvolvidos de forma a liberar o GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, o GIL é sempre liberado nas operações de E/S.

No passado, esforços para criar um interpretador que lidasse plenamente com threads (travando dados compartilhados numa granularidade bem mais fina) não foram bem sucedidos devido a queda no desempenho ao serem



executados em processadores de apenas um núcleo. Acredita-se que superar essa questão de desempenho acabaria tornando a implementação muito mais complicada e bem mais difícil de manter.

**hasheável** An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

A hashabilidade faz com que um objeto possa ser usado como chave de um dicionário e como membro de um conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

**IDLE** Um ambiente de desenvolvimento integrado para Python. IDLE é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

**imutável** Um objeto que possui um valor fixo. Objetos imutáveis incluem números, strings e tuplas. Estes objetos não podem ser alterados. Um novo objeto deve ser criado se um valor diferente tiver de ser armazenado. Objetos imutáveis têm um papel importante em lugares onde um valor constante de hash seja necessário, como por exemplo uma chave em um dicionário.

**integer division** Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to 2 in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *\_\_future\_\_*.

**importing (importando)** O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

**importer** Um objeto que localiza e carrega um módulo; Tanto um *finder* e o objeto *loader*.

**interactive** Python tem um interpretador interativo, o que significa que você pode digitar comandos e expressões no prompt do interpretador, executá-los imediatamente e ver seus resultados. Apenas execute `python` sem argumentos (possivelmente selecionando-o a partir do menu de aplicações de seu sistema operacional). O interpretador interativo é uma maneira poderosa de testar novas ideias ou aprender mais sobre módulos e pacotes (lembre-se do comando `help(x)`).

**interpreted** Python é uma linguagem interpretada, em oposição àquelas que são compiladas, embora esta distinção possa ser nebulosa devido à presença do compilador de bytecode. Isto significa que os arquivos-fontes podem ser executados diretamente sem necessidade explícita de se criar um arquivo executável. Linguagens interpretadas normalmente têm um ciclo de desenvolvimento/depuração mais curto que as linguagens compiladas, apesar de seus programas geralmente serem executados mais lentamente. Veja também *interativo*.

**iterável** An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

**iterador** An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code

which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Mais informações podem ser encontradas em `typeiter`.

**key function (função chave)** Uma função chave ou função colação é algo que retorna um valor utilizado para ordenação ou classificação. Por exemplo, `locale.strxfrm()` é usada para produzir uma chave de ordenação que leva o `locale` em consideração para fins de ordenação.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example, the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

**argumento nomeado** Veja o *argument*.

**lambda** Uma função de linha anônima consistindo de uma única *expression*, que é avaliada quando a função é chamada. A sintaxe para criar uma função `lambda` é `lambda [parameters]: expression`

**LBYL** Iniciais da expressão em inglês “look before you leap”, que significa algo como “olhe antes de pisar”. Este estilo de codificação testa as pré-condições explicitamente antes de fazer chamadas ou buscas. Este estilo contrasta com a abordagem *EAFP* e é caracterizada pela presença de muitos comandos `if`.

Em um ambiente multithread, a abordagem LBYL pode arriscar a introdução de uma condição de corrida entre “o olhar” e “o pisar”. Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover *key* do *mapping* após o teste, mas antes da olhada. Esse problema pode ser resolvido com bloqueios ou usando a abordagem *EAFP*.

**lista** Uma *sequence* embutida no Python. Apesar do seu nome, é mais próximo de um vetor em outras linguagens do que uma lista encadeada, como o acesso aos elementos é da ordem  $O(1)$ .

**list comprehension** A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

**loader** An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

**método mágico** Um sinônimo informal para um *special method*.

**mapeamento** A container object that supports arbitrary key lookups and implements the methods specified in the Mapping or MutableMapping abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

**metaclass** A classe de uma classe. Definições de classe criam um nome de classe, um dicionário de classe e uma lista de classes base. A metaclasses é responsável por receber estes três argumentos e criar a classe. A maioria das linguagens de programação orientadas a objetos provê uma implementação default. O que torna o Python especial é o fato de ser possível criar metaclasses personalizadas. A maioria dos usuários nunca vai precisar deste recurso, mas quando houver necessidade, metaclasses possibilitam soluções poderosas e elegantes. Metaclasses têm sido utilizadas para gerar registros de acesso a atributos, para incluir proteção contra acesso concorrente, rastrear a criação de objetos, implementar singletons, dentre muitas outras tarefas.

More information can be found in metaclasses.

**method (método)** Uma função que é definida dentro do corpo de uma classe. Se chamada como um atributo de uma instância daquela classe, o método receberá a instância do objeto como seu primeiro *argumento* (que comumente é chamado de `self`). Veja *função* e *nested scope*.

**method resolution order (ordem de resolução de método)** Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja *A ordem de resolução de métodos do Python 2.3*.

**módulo** Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um namespace contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

**MRO** Veja *method resolution order*.

**mutável** Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *immutable*.

**tupla nomeada** Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

**namespace** The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

**nested scope (escopo aninhado)** The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

**new-style class (novo estilo de classes)** Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattr__()`.

More information can be found in *newstyle*.

**object (objeto)** Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *new-style class*.

**pacote** Um *module* Python é capaz de conter submódulos ou recursivamente, sub-pacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

**parâmetro** A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *posicional-ou-nomeado*: especifica um argumento que pode ser tanto *posicional* quanto *nomeado*. Esse é o tipo padrão de parâmetro, por exemplo *foo* e *bar* a seguir:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).

- *var-posicional*: especifica quem uma sequência arbitrária de argumentos posicionais pode ser fornecida (em adição a qualquer argumento posicional já aceito por outros parâmetros). Tal parâmetro pode ser definido colocando um `*` antes do nome, por exemplo *args* a seguir:

```
def func(*args, **kwargs): ...
```

- *var-nomeado*: especifica que, arbitrariamente, muitos argumentos nomeados podem ser fornecidos (em adição a qualquer argumento nomeado já aceito por outros parâmetros). Tal parâmetro pode ser definido colocando-se `**` antes do nome, por exemplo *kwargs* no exemplo acima.

Parâmetros podem especificar tanto argumentos opcionais quanto obrigatórios, assim como valores padrões para alguns argumentos opcionais.

See also the [argument](#) glossary entry, the FAQ question on [the difference between arguments and parameters](#), and the function section.

**PEP** Proposta de melhoria do Python. Uma PEP é um documento de design que fornece informação para a comunidade Python, ou descreve uma nova funcionalidade para o Python ou seus predecessores ou ambientes. PEPs devem prover uma especificação técnica concisa e um racional para funcionalidades propostas.

PEPs tem a intenção de ser os mecanismos primários para propor novas funcionalidades significativas, para coletar opiniões da comunidade sobre um problema, e para documentar as decisões de design que foram adicionadas ao Python. O autor da PEP é responsável por construir um consenso dentro da comunidade e documentar opiniões dissidentes.

Veja [PEP 1](#).

**positional argument (argumento posicional)** Veja o [argument](#).

**Python 3000** Apelido para a versão do Python 3.x linha de lançamento (cunhado há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: “Py3k”.

**Pythonic** Uma ideia ou um pedaço de código que segue de perto os idiomas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outros idiomas. Por exemplo, um idioma comum em Python é fazer um loop sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(food)):
    print food[i]
```

Ao contrário do método limpo, ou então, Pythonico:

```
for piece in food:
    print piece
```

**reference count** O número de referências para um objeto. Quando a contagem de referências de um objeto atinge zero, ele é desalocado. Contagem de referências geralmente não é visível no código Python, mas é um elemento chave da implementação *CPython*. O módulo `sys` define a função `getrefcount()` que programadores podem chamar para retornar a contagem de referências para um objeto em particular.

**\_\_slots\_\_** A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

**sequência** An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

**fatia** An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

**método especial** Um método que é chamado implicitamente pelo Python para executar uma certa operação em um tipo, como uma adição por exemplo. Tais métodos tem nomes iniciando e terminando com dois underscores. Métodos especiais estão documentados em `specialnames`.

**instrução** Uma instrução é parte de uma suíte (um “bloco” de código). Uma instrução é ou uma *expression* ou uma de várias construções com uma palavra-chave, tal como `if`, `while` ou `for`.

**struct sequence** A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

**aspas triplas** Uma string que está definida com três ocorrências de aspas duplas (“) ou apóstrofes (‘). Enquanto elas não fornecem nenhuma funcionalidade não disponível com strings de aspas simples, elas são úteis para inúmeras razões. Elas permitem que você inclua aspas simples e duplas não encerradas dentro de uma string, e elas podem utilizar múltiplas linhas sem o uso de caractere de continuação, fazendo-as especialmente úteis quando escrevemos documentação em docstrings.

**type** O tipo de um objeto Python determina qual tipo de objeto ele é; cada objeto tem um tipo. Um tipo de objeto é acessível pelo atributo `__class__` ou pode ser recuperado com `type(obj)`.

**Novas linhas universais** A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See [PEP 278](#) and [PEP 3116](#), as well as `str.splitlines()` for an additional use.

**ambiente virtual** Um ambiente de execução isolado que permite usuários Python e aplicações instalarem e atualizarem pacotes Python sem interferir no comportamento de outras aplicações Python em execução no mesmo sistema.

**máquina virtual** Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

**Zen of Python** Lista de princípios de projeto e filosofias do Python que são úteis para a compreensão e uso da linguagem. A lista é exibida quando se digita `“import this”` no console interativo.



---

### Sobre esses documentos

---

Esses documentos são gerados a partir de [reStructuredText](#) pelo [Sphinx](#), um processador de documentos especificamente escrito para documentação do Python.

O desenvolvimento da documentação e de suas ferramentas é um esforço totalmente voluntário, como o Python em si. Se você quer contribuir, por favor dê uma olhada na página [reporting-bugs](#) para informações sobre como fazer. Novos voluntários são sempre bem vindos!

Agradecimentos especiais para:

- Fred L. Drake, Jr., o criador do primeiro conjunto de ferramentas para documentar o Python e escritor de boa parte do conteúdo;
- O projeto [Docutils](#) para criar [reStructuredText](#) e o pacote [Docutils](#);
- Fredrik Lundh por seu projeto [Referência Alternativa para Python](#) do qual Sphinx teve muitas ideias boas.

### B.1 Contribuidores da Documentação do Python

Muitas pessoas tem contribuído para a linguagem Python, sua biblioteca padrão e sua documentação. Veja [Misc/ACKS](#) na distribuição do código do Python para ver uma lista parcial de contribuidores.

Tudo isso só foi possível com o esforço e a contribuição da comunidade Python, por isso temos essa maravilhosa documentação – Obrigado a todos!





## História e Licença

### C.1 História do software

O Python foi criado no início dos anos 1990 por Guido van Rossum na Stichting Mathematisch Centrum (CWI, veja <https://www.cwi.nl/>) na Holanda como um sucessor de uma linguagem chamada ABC. Guido continua a ser o principal autor de Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporação para Iniciativas Nacionais de Pesquisa (CNRI, veja <https://www.cnri.reston.va.us/>) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe principal de desenvolvimento do Python mudaram-se para o BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe da PythonLabs mudou para a Digital Creations (agora Zope Corporation; veja <https://www.zope.org/>). Em 2001, formou-se a Python Software Foundation (PSF, ver <https://www.python.org/psf/>), uma organização sem fins lucrativos criada especificamente para possuir propriedade intelectual relacionada a Python. A Zope Corporation é um membro patrocinador do PSF.

Todas as versões do Python são de código aberto (consulte <https://opensource.org/> para a definição de código aberto). Historicamente, a maioria, mas não todas, versões do Python também são compatíveis com GPL; a tabela abaixo resume os vários lançamentos.

Release	Derivado de	Ano	Proprietário	GPL compatível?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e acima	2.1.1	2001-agora	PSF	sim

**Nota:** Compatível com GPL não significa que estamos distribuindo Python sob a GPL. Todas as licenças do Python, ao contrário da GPL, permitem distribuir uma versão modificada sem fazer alterações em código aberto. As licenças compatíveis com GPL possibilitam combinar o Python com outro software lançado sob a GPL; os outros não.

---

Graças aos muitos voluntários externos que trabalharam sob a direção de Guido para tornar esses lançamentos possíveis.

## C.2 Termos e condições para acessar ou usar Python

### C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.18

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),  
and  
the Individual or Organization ("Licensee") accessing and otherwise using  
Python  
2.7.18 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby  
grants Licensee a nonexclusive, royalty-free, world-wide license to  
reproduce,  
analyze, test, perform and/or display publicly, prepare derivative works,  
distribute, and otherwise use Python 2.7.18 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's notice  
of  
copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All  
Rights  
Reserved" are retained in Python 2.7.18 alone or in any derivative version  
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or  
incorporates Python 2.7.18 or any part thereof, and wants to make the  
derivative work available to others as provided herein, then Licensee  
hereby  
agrees to include in any such work a brief summary of the changes made to  
Python  
2.7.18.
4. PSF is making Python 2.7.18 available to Licensee on an "AS IS" basis.  
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF  
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION  
OR  
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT  
THE  
USE OF PYTHON 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18  
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT  
OF  
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, OR ANY  
DERIVATIVE  
THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.7.18, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.2 ACORDO DE LICENÇA DA BEOPEN.COM PARA PYTHON 2.0

### CONTRATO DE LICENÇA DE FONTE ABERTA DO BEOPEN PYTHON VERSÃO 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at

(continua na próxima página)

(continuação da página anterior)

<http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.3 CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed

(continua na próxima página)

(continuação da página anterior)

under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.4 ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

## C.3 Licenças e Reconhecimentos para Software Incorporado

Esta seção é uma lista incompleta, mas crescente, de licenças e confirmações para softwares de terceiros incorporados na distribuição do Python.

### C.3.1 Mersenne Twister

O módulo `_random` inclui código baseado em um download de <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. A seguir estão os comentários literais do código original:

A C-program for MT19937, with initialization improved 2002/1/26.  
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`  
or `init_by_array(init_key, key_length)`.

(continua na próxima página)

(continuação da página anterior)

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:

1. Redistributions of source code must retain the above copyright  
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  
notice, this list of conditions and the following disclaimer in the  
documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote  
products derived from this software without specific prior written  
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR  
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR  
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

## C.3.2 Sockets

O módulo `socket` usa as funções `getaddrinfo()` e `getnameinfo()`, que são codificadas em arquivos de origem separados do Projeto WIDE, <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:

1. Redistributions of source code must retain the above copyright  
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  
notice, this list of conditions and the following disclaimer in the  
documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors  
may be used to endorse or promote products derived from this software  
without specific prior written permission.

(continua na próxima página)

(continuação da página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

### C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
-----
/                               Copyright (c) 1996.                               \
|                               The Regents of the University of California.          |
|                               All rights reserved.                                |
|                                                                                 |
|  Permission to use, copy, modify, and distribute this software for               |
|  any purpose without fee is hereby granted, provided that this en-               |
|  tire notice is included in all copies of any software which is or               |
|  includes a copy or modification of this software and in all                     |
|  copies of the supporting documentation for such software.                       |
|                                                                                 |
|  This work was produced at the University of California, Lawrence                  |
|  Livermore National Laboratory under contract no. W-7405-ENG-48                  |
|  between the U.S. Department of Energy and The Regents of the                   |
|  University of California for the operation of UC LLNL.                         |
|                                                                                 |
|                               DISCLAIMER                                           |
|                                                                                 |
|  This software was prepared as an account of work sponsored by an                |
|  agency of the United States Government. Neither the United States               |
|  Government nor the University of California nor any of their em-                |
|  ployees, makes any warranty, express or implied, or assumes any                 |
|  liability or responsibility for the accuracy, completeness, or                  |
|  usefulness of any information, apparatus, product, or process                   |
|  disclosed, or represents that its use would not infringe                       |
|  privately-owned rights. Reference herein to any specific commer-                |
|  cial products, process, or service by trade name, trademark,                    |
|  manufacturer, or otherwise, does not necessarily constitute or                  |
|  imply its endorsement, recommendation, or favoring by the United               |
|  States Government or the University of California. The views and                |
|  opinions of authors expressed herein do not necessarily state or                |
|  reflect those of the United States Government or the University                 |
|  of California, and shall not be used for advertising or product                 |
|  \ endorsement purposes.                                                         /
-----
```

### C.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software.  If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
    http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
    references to Ghostscript; clarified derivation from RFC 1321;
    now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
    added conditionalization for C++ compilation from Martin
    Porschke <porschke@bnl.gov>.
1999-05-03 lpd Original version.
```



### C.3.5 Serviços de soquete assíncrono

Os módulos `asyncchat` e `asyncore` contêm o seguinte aviso:

```
Copyright 1996 by Sam Rushing
```

```
    All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.
```

```
SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

### C.3.6 Gerenciamento de cookies

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>
```

```
    All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

```
Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

### C.3.7 Rastreamento de execução

O módulo `trace` contém o seguinte aviso:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

### C.3.8 Funções `UUencode` e `UUdecode`

O módulo `uu` contém o seguinte aviso:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(continua na próxima página)

(continuação da página anterior)

```
version is still 5 times faster, though.  
- Arguments more compliant with Python standard
```

### C.3.9 Chamadas de Procedimento Remoto XML

The `xmlrpclib` module contains the following notice:

```
The XML-RPC client interface is  
  
Copyright (c) 1999-2002 by Secret Labs AB  
Copyright (c) 1999-2002 by Fredrik Lundh  
  
By obtaining, using, and/or copying this software and/or its  
associated documentation, you agree that you have read, understood,  
and will comply with the following terms and conditions:  
  
Permission to use, copy, modify, and distribute this software and  
its associated documentation for any purpose and without fee is  
hereby granted, provided that the above copyright notice appears in  
all copies, and that both that copyright notice and this permission  
notice appear in supporting documentation, and that the name of  
Secret Labs AB or the author not be used in advertising or publicity  
pertaining to distribution of the software without specific, written  
prior permission.  
  
SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD  
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-  
ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR  
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY  
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,  
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS  
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  
OF THIS SOFTWARE.
```

### C.3.10 test\_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.  
  
Permission is hereby granted, free of charge, to any person obtaining  
a copy of this software and associated documentation files (the  
"Software"), to deal in the Software without restriction, including  
without limitation the rights to use, copy, modify, merge, publish,  
distribute, sublicense, and/or sell copies of the Software, and to  
permit persons to whom the Software is furnished to do so, subject to  
the following conditions:  
  
The above copyright notice and this permission notice shall be  
included in all copies or substantial portions of the Software.  
  
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,  
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(continua na próxima página)

(continuação da página anterior)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

### C.3.11 Selezione o kqueue

The select and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

### C.3.12 strtod e dtoa

O arquivo `Python/dtoa.c`, que fornece as funções C `dtoa` e `strtod` para conversão de duplas de C para e de strings, é derivado do arquivo com o mesmo nome de David M. Gay, atualmente disponível em <http://www.netlib.org/fp/>. O arquivo original, conforme recuperado em 16 de março de 2009, contém os seguintes avisos de direitos autorais e de licenciamento:

```
/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 */
```

(continua na próxima página)

(continuação da página anterior)

```
* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
* WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
*****/
```

### C.3.13 OpenSSL

Os módulos `hashlib`, `posix`, `ssl`, `crypt` usam a biblioteca OpenSSL para desempenho adicional se forem disponibilizados pelo sistema operacional. Além disso, os instaladores do Windows e do Mac OS X para Python podem incluir uma cópia das bibliotecas do OpenSSL, portanto incluímos uma cópia da licença do OpenSSL aqui:

#### LICENSE ISSUES

```
=====
```

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact [openssl-core@openssl.org](mailto:openssl-core@openssl.org).

#### OpenSSL License

```
-----
```

```
/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without
 * prior written permission. For written permission, please contact
 * openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 * nor may "OpenSSL" appear in their names without prior written
 * permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
```

(continua na próxima página)

(continuação da página anterior)

```

*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscape SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software

```

(continua na próxima página)

(continuação da página anterior)

```

*      must display the following acknowledgement:
*      "This product includes cryptographic software written by
*      Eric Young (eay@cryptsoft.com)"
*      The word 'cryptographic' can be left out if the routines from the library
*      being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*      the apps directory (application code) you must include an acknowledgement:
*      "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
*  THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
*  ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
*  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
*  ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
*  FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
*  DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
*  OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
*  HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
*  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
*  OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
*  SUCH DAMAGE.
*
*  The licence and distribution terms for any publically available version or
*  derivative of this code cannot be changed.  i.e. this code cannot simply be
*  copied and put under another distribution licence
*  [including the GNU Public Licence.]
*/

```

### C.3.14 expat

A extensão pyexpat é construída usando uma cópia incluída das fontes de expatriadas, a menos que a compilação esteja configurada `--with-system-expat`:

```

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```

### C.3.15 libffi

A extensão `_ctypes` é construída usando uma cópia incluída das fontes `libffi`, a menos que a compilação esteja configurada `--with-system-libffi`:

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

### C.3.16 zlib

A extensão `zlib` é construída usando uma cópia incluída das fontes `zlib` se a versão do `zlib` encontrada no sistema for muito antiga para ser usada na construção:

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
```



## APÊNDICE D

---

### Copyright

---

Python e essa documentação é:

Copyright © 2001-2020 Python Software Foundation. Todos os direitos reservados.

Copyright © 2000 BeOpen.com. Todos os direitos reservados.

Copyright © 1995-2000 Corporation for National Research Initiatives. Todos os direitos reservados.

Copyright © 1991-1995 Stichting Mathematisch Centrum. Todos os direitos reservados.

---

Veja: *História e Licença* para informações completas de licença e permissões.



## Não alfabético

..., [89](#)  
2to3, [89](#)  
>>>, [89](#)  
\_\_future\_\_, [92](#)  
\_\_slots\_\_, [96](#)

## A

ambiente virtual, [97](#)  
argument  
    difference from parameter, [17](#)  
argumento, [89](#)  
argumento nomeado, [94](#)  
aspas triplas, [97](#)  
atributo, [90](#)

## B

BDFL, [90](#)  
bytecode, [90](#)

## C

Classe, [90](#)  
classe base abstrata, [89](#)  
classic class, [90](#)  
coerção, [90](#)  
CPython, [90](#)

## D

decorador, [90](#)  
descritor, [91](#)  
dicionário, [91](#)  
divisão pelo piso, [92](#)  
docstring, [91](#)  
duck-typing (*tipagem pato*), [91](#)

## E

EAFP, [91](#)  
expressão, [91](#)  
extension module (*módulo de extensão*), [91](#)

## F

fatia, [97](#)  
função, [92](#)

## G

garbage collection (*coletor de lixo*), [92](#)  
generator, [92](#)  
generator expression, [92](#)  
gerador, [92](#)  
gerenciador de contexto, [90](#)  
GIL, [92](#)  
global interpreter lock (*bloqueio global do interpretador*), [92](#)

## H

hasheável, [93](#)

## I

IDLE, [93](#)  
importer, [93](#)  
importing (*importando*), [93](#)  
imutável, [93](#)  
instrução, [97](#)  
integer division, [93](#)  
interactive, [93](#)  
interpreted, [93](#)  
iterador, [93](#)  
iterável, [93](#)

## K

key function (*função chave*), [94](#)

## L

lambda, [94](#)  
LBYL, [94](#)  
list comprehension, [94](#)  
lista, [94](#)  
loader, [94](#)  
localizador, [92](#)

## M

- magic
  - method, [94](#)
- mapeamento, [94](#)
- máquina virtual, [97](#)
- metaclass, [94](#)
- method
  - magic, [94](#)
  - special, [97](#)
- method (*método*), [95](#)
- method resolution order (*ordem de resolução de método*), [95](#)
- método especial, [97](#)
- método mágico, [94](#)
- módulo, [95](#)
- MRO, [95](#)
- mutável, [95](#)

## N

- namespace, [95](#)
- nested scope (*escopo aninhado*), [95](#)
- new-style class (*novo estilo de classes*), [95](#)
- Novas linhas universais, [97](#)
- número complexo, [90](#)

## O

- object (*objeto*), [95](#)
- objeto arquivo, [91](#)
- objeto byte ou similar, [90](#)
- objeto como-arquivo, [92](#)

## P

- pacote, [95](#)
- parameter
  - difference from argument, [17](#)
- parâmetro, [95](#)
- PATH, [56](#)
- PEP, [96](#)
- positional argument (*argumento posicional*), [96](#)
- Propostas Estendidas Python
  - PEP 1, [96](#)
  - PEP 5, [6](#)
  - PEP 6, [3](#)
  - PEP 8, [10](#)
  - PEP 238, [24](#), [92](#)
  - PEP 275, [46](#)
  - PEP 278, [97](#)
  - PEP 302, [92](#), [94](#)
  - PEP 343, [90](#)
  - PEP 3116, [97](#)
- Python 3000, [96](#)
- Pythonic, [96](#)

## R

- reference count, [96](#)

## S

- sequência, [96](#)
- special
  - method, [97](#)
- struct sequence, [97](#)

## T

- TCL\_LIBRARY, [84](#)
- TK\_LIBRARY, [84](#)
- tupla nomeada, [95](#)
- type, [97](#)

## V

- váriavel de ambiente
  - PATH, [56](#)
  - TCL\_LIBRARY, [84](#)
  - TK\_LIBRARY, [84](#)
- visualização de dicionário, [91](#)

## Z

- Zen of Python, [97](#)