
Python Frequently Asked Questions

リリース 3.13.1

Guido van Rossum and the Python development team

1 月 03, 2025

目次

第 1 章	一般 Python FAQ	1
第 2 章	プログラミング FAQ	9
第 3 章	デザインと歴史 FAQ	51
第 4 章	ライブラリと拡張 FAQ	65
第 5 章	拡張と埋め込み FAQ	79
第 6 章	Windows 上の Python FAQ	85
第 7 章	グラフィックユーザインターフェース FAQ	91
第 8 章	"なぜ Python が私のコンピュータにインストールされているのですか?" FAQ	93
付録 A 章	用語集	95
付録 B 章	About this documentation	119
付録 C 章	歴史とライセンス	121
付録 D 章	Copyright	147
索引		149
索引		149

一般 PYTHON FAQ

1.1 一般情報

1.1.1 Python とは何ですか？

Python は、インタプリタ形式の、対話的な、オブジェクト指向プログラミング言語です。この言語には、モジュール、例外、動的な型付け、超高水準の動的なデータ型、およびクラスが取り入れられています。Python は、オブジェクト指向プログラミングを越えて、手続き型プログラミングや関数型プログラミングなど複数のプログラミングパラダイムをサポートしています。Python は驚くべきパワーと非常に分かりやすい文法を持ち合わせています。そして、多くのシステムコールやライブラリへだけでなく、様々なウィンドウシステムへのインターフェースがあり、C や C++ で拡張することもできます。また、プログラム可能なインターフェースが必要なアプリケーションのための拡張言語としても利用できます。最後に、Python はポータブルです。Linux や macOS を含む多くの Unix の派生システムで動き、Windows でも動きます。

さらに知りたければ、[tutorial-index](#) から始めましょう。他にも、[Beginner's Guide to Python](#) から、Python 学習のための入門用チュートリアルやリソースを参照できます。

1.1.2 Python Software Foundation とは何ですか？

Python Software Foundation は、Python バージョン 2.1 以降の著作権を保持する独立の非営利組織です。PSF の任務は、Python プログラミング言語に関するオープンソース技術を進め、Python の使用を広めることです。PSF のホームページが <https://www.python.org/psf/> にあります。

PSF への寄付は米国で免税されています。Python を利用して役立ったと感じたら、[the PSF donation page](#) で貢献をお願いします。

1.1.3 Python を使うのに著作権の制限はありますか？

あなたが作成する Python に関するドキュメントのすべてに著作権を残し、それらの著作権を表示する限り、ソースコードをどのように扱ってもかまいません。この著作権規則を尊重する限り、商用に Python を利用しても、ソースあるいはバイナリ形式で (変更の有無にかかわらず) Python のコピーを販売しても、Python が何らかの形で組み込まれた製品を販売してかまいません。もちろん、Python のあらゆる商業用途についても同様です。

PSF License のより詳しい説明と全文へのリンクは [the license page](#) を参照してください。

Python のロゴは商標登録されていて、使用に許可が必要な場合があります。詳しい情報は [the Trademark Usage Policy](#) を参照してください。

1.1.4 そもそも Python は何故創られたのですか？

Guido van Rossum によると、すべての始まりについての **非常に** 簡潔な概要は以下のとおりです:

私は CWI の ABC グループでインタプリタ言語を実装する幅広い経験をしていて、そこで共に働くことで言語設計に関して大いに学びました。これは文のグループ化にインデントを使用することや超高水準のデータ型の包含など、(詳細は Python とは全く異なりますが) Python の多くの特徴のもととなっています。

私は ABC 言語に対して多くの不満を持っていましたが、同時に特徴の多くが好きでもありました。ABC 言語 (やその実装) を拡張して私の不満を解消することは不可能でした (実際、拡張性の欠如は大きな問題だったのです)。私は Modula-2+ を使用することでいくらかの経験を積み、Modula-3 のデザイナと話して Modula-3 のレポートを読みました。Modula-3 は例外処理に使う構文や語義、その他いくつかの Python の特徴の起源です。

私は CWI で Amoeba 分散オペレーティングシステムのグループで働いていました。Amoeba のシステムコールインターフェースには Bourne シェルからアクセスしにくかったので、C プログラムや Bourne シェルスクリプトを書くよりも良いシステム管理の方法が必要でした。Amoeba のエラー処理の経験から、プログラミング言語の機能としての例外の重要性を強く意識するようになりました。

ABC のような構文と Amoeba のようなシステムコールを合わせ持ったスクリプト言語が必要だろうと思いつきました。Amoeba 専用の言語を書くのは愚かであるだろうと気づき、一般に拡張できるような言語を求めることに決めました。

1989 年のクリスマス休暇の間、自由な時間がたくさんできたので、その言語を実際に作ってみることにしました。翌年の間、勤務時間以外はほとんどその開発に費やし、Python は Amoeba プロジェクトのなかで成果を重ね、同僚からのフィードバックは私の開発を大いに加速させてくれました。

1991 年 2 月、1 年間で少しの開発を経て、USENET に投稿することにしました。それは Misc/HISTORY ファイルに残っています。

1.1.5 Python は何をするのに向いていますか？

Python は、多岐にわたる問題に適用できる高水準な汎用プログラム言語です。

この言語は、文字列処理 (正規表現、Unicode、ファイル間の差分の計算)、インターネットプロトコル (HTTP、FTP、SMTP、XML-RPC、POP、IMAP)、ソフトウェアエンジニアリング (Python コードのユニットテスト、ロギング、プロファイリング、解析)、オペレーティングシステムインターフェース (システムコール、ファイルシステム、TCP/IP ソケット) のような領域をカバーする大規模な標準ライブラリから成り立っています。何ができるかを知るには [library-index](#) の一覧を参照してください。また、様々なサードパーティの拡張も使えます。[the Python Package Index](#) から興味のあるパッケージを探してみましょう。

1.1.6 Python のバージョン番号の仕組みはどうなっているのですか？

Python のバージョン番号は "A.B.C" または "A.B" が付けられます：

- *A* はメジャーバージョン番号です -- 言語に本当に大きな変更があった場合に増やされます。
- *B* はマイナーバージョン番号です -- 驚きの少ない変更があった場合に増やされます。
- *C* はマイクロバージョン番号です -- バグフィックスリリースごとに増やされます。

すべてのリリースがバグフィックスリリースであるというわけではありません。フィーチャーリリースへの準備段階では、一連の開発リリースが作られ、アルファ版、ベータ版、またはリリース候補と名付けられます。アルファ版はインターフェースが確定されないうちにリリースされる早期リリースで、2つのアルファリリース間でインターフェースが変わるかもしれません。ベータ版はもっと安定していて、現存のインターフェースは保存されますが新しいモジュールが追加されるかもしれません。リリース候補は固まったもので、致命的なバグを直すのでなければ変更されません。

アルファ、ベータとリリース候補のバージョンは、追加の接尾辞を持ちます：

- アルファバージョンの接尾辞は、ある小さな数 *N* に対して "a*N*" となります。
- ベータバージョンの接尾辞は、ある小さな数 *N* に対して "b*N*" となります。
- リリース候補バージョンの接尾辞は、ある小さな数 *N* に対して "rc*N*" となります。

つまり、*2.0a*N** と付けられたバージョンはすべて *2.0b*N** に先立ち、*2.0b*N** は *2.0rc*N** に先立ち、そして **これら** は 2.0 に先立ちます。

また、"2.2+" のように "+" 接尾語が付いたバージョン番号もあります。これは未発表のバージョンで、CPython 開発リポジトリから直接ビルドされています。実際、最後のマイナーリリースが行われた後、バージョンは "2.4a0" のように "a0" がつく次のマイナーバージョンになります。

開発サイクルの詳細は [Developer's Guide](#) を、Python の後方互換性ポリシーについては [PEP 387](#) を参照してください。ドキュメント `sys.version`、`sys.hexversion`、`sys.version_info` も参照してください。

1.1.7 Python のソースのコピーはどこで手に入りますか？

最新の Python ソースは [python.org](https://www.python.org/downloads/) (<https://www.python.org/downloads/>) からいつでも手に入れることができます。最新の開発版ソースは <https://github.com/python/cpython/> にアクセスして手に入れることができます。

ソースは gzip された tar ファイルで配布され、完全な C のソース、Sphinx によりフォーマットされたドキュメント、Python ライブラリモジュール、サンプルプログラム、そしていくつかの役立つ配布自由なソフトウェアを含んでいます。このソースはほとんどの UNIX プラットフォームでそのままコンパイルして動かします。

ソースコードの入手とコンパイルについて、より詳しい情報は [Getting Started section of the Python Developer's Guide](#) を参照してください。

1.1.8 Python のドキュメントはどこで手に入りますか？

Python の現行の安定バージョンの標準ドキュメントは <https://docs.python.org/3/> から利用できます。また、PDF、プレーンテキスト、ダウンロードできる HTML 版も <https://docs.python.org/3/download.html> から利用できます。

このドキュメントは reStructuredText で書かれ、the Sphinx documentation tool で構成されました。このドキュメントに使われた reStructuredText のソースは Python のソース配布に含まれます。

1.1.9 プログラミングをしたことがないのですが、Python のチュートリアルはありますか？

膨大な量の役に立つチュートリアルや書籍があります。標準のドキュメントには tutorial-index などがあります。

Python プログラム初心者のための情報のチュートリアルのリストは the Beginner's Guide を参照してください。

1.1.10 Python のためのニュースグループやメーリングリストはありますか？

ニュースグループ *comp.lang.python* やメーリングリスト *python-list* があります。ニュースグループとメーリングリストは互いに接続されていて、ニュースを購読すればメーリングリストに参加する必要はありません。*comp.lang.python* は一日に数百のポスティングを受ける高いトラフィックで、USENET 読者は多くの場合、このボリュームに 대응することができます。

新しいソフトウェアリリースとイベントの告知は *comp.lang.python.announce* で見つけられます。これは一日に 5 ポスティング程度を受ける低トラフィックの手頃なメーリングリストです。the *python-announce mailing list* から利用可能です。

その他のメーリングリストやニュースグループについての詳しい情報は <https://www.python.org/community/lists/> にあります。

1.1.11 Python のベータテスト版はどこで手に入りますか？

アルファ/ベータリリースは <https://www.python.org/downloads/> で手に入ります。リリースはすべて *comp.lang.python* や *comp.lang.python.announce* のニュースグループと Python ホームページ <https://www.python.org/> で告知され、RSS ニュースフィードが利用できます。

Git から開発版を手に入れることもできます。詳細は The Python Developer's guide を参照してください。

1.1.12 Python のバグ報告やパッチを上げるにはどうしたら良いですか？

バグの報告やパッチの提出には、issue tracker <https://github.com/python/CPython/issues> をご利用ください。

Python 開発の工程について、詳しくは the Python Developer's Guide を参照してください。

1.1.13 Python について発行された記事を何か参照できますか？

Python に関するあなたの愛読書を引用するのが一番でしょう。

Python に関する [一番初めの記事](#) はとても古く、1991 年に書かれています。

Guido van Rossum and Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language", CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283--303.

1.1.14 Python の本はありますか？

はい、たくさんあり、そのほとんどは現在も出版されています。リストは [python.org wiki https://wiki.python.org/moin/PythonBooks](https://wiki.python.org/moin/PythonBooks) にあります。

また、オンライン書店で "Python" で検索し、Monty Python をフィルタで除外してもいいです (または、"Python" と " 言語" で検索してください)。

1.1.15 www.python.org は世界のどこにあるのですか？

Python プロジェクトのインフラは世界中にあり、Python インフラストラクチャチームによって管理されています。詳細は [こちら](#)。

1.1.16 なぜ Python という名前なのですか？

Python の開発が始まった頃、Guido van Rossum は 1970 年代に始まった BBC のコメディシリーズ "Monty Python's Flying Circus" の台本を読んでいた。Van Rossum は、短くて、ユニークで、少しミステリアスな名前が欲しかったので、この言語の名前を Python と呼ぶことにしたのです。

1.1.17 『空飛ぶモンティ・パイソン』を好きでなくてはいけませんか？

いいえ。でも、好きになってくれるといいな。:)

1.2 現実世界での Python

1.2.1 Python はどれくらい安定していますか？

とても安定しています。1991 年以来新しい安定リリースはおおよそ 6 から 18 ヶ月毎に出されていて、このペースが続きそうです。バージョン 3.9 の時点では、Python は 12 ヶ月に一度のペースで新しいフィーチャーリリースを行っています ([PEP 602](#))。

開発者が旧バージョンのバグフィックスリリースを公開するので、既存リリースの安定性は徐々に向上していきます。バグフィックスリリースは、バージョン番号の 3 番目の要素によって示され (例: 3.5.3、3.6.2)、安定性のために管理されています。バグフィックスリリースには既知の問題への修正だけが含まれ、一連のバグフィックスリリースを通じて同じインターフェースが保たれることが保証されているのです。

The latest stable releases can always be found on the [Python download page](#). Python 3.x is the recommended version and supported by most widely used libraries. Python 2.x **is not maintained anymore**.

1.2.2 どれくらいの人が Python を使っていますか？

正確な人数を調べるのは難しいですが、おそらく何百万人ものユーザーがいるでしょう。

Python は自由にダウンロード可能なので、売上高がなく、多数のサイトから利用でき、多くの Linux ディストーションに同梱されているので、ダウンロード統計から全体の状況を知ることはできません。

comp.lang.python ニュースグループはとても活発ですが、すべての Python ユーザーが投稿するわけではなく、読みすらしない人もいます。

1.2.3 Python で行われた大きなプロジェクトはありますか？

Python を利用しているプロジェクトのリストは <https://www.python.org/about/success> を参照してください。past Python conferences から議事録を参照すると、多くの会社や組織の貢献がわかるでしょう。

注目されている Python のプロジェクトは the Mailman mailing list manager や the Zope application server などです。Red Hat をはじめとするいくつかの Linux ディストーションのインストーラやシステムアドミニストレーションソフトウェアは、一部や全部が Python で書かれています。内部で Python を利用している企業には、Google、Yahoo、Lucasfilm Ltd. などがあります。

1.2.4 将来 Python にどのような新しい開発が期待されますか？

Python Enhancement Proposals (PEP) <https://peps.python.org/> を参照してください。PEP は Python に提案された新機能について書かれた設計書で、簡潔な技術仕様と原理が提供されています。”Python X.Y Release Schedule” (X.Y はまだリリースされていないバージョン) を探してください。

新しい開発については the python-dev mailing list で議論されています。

1.2.5 Python の互換性を無くすような提案をしてもいいのですか？

一般的には、してはいけません。世界中にすでに何百万行もの Python コードがあるので、既存のプログラムのどんなに僅かな部分でも無効にしてしまうような言語仕様の変更も認められてはなりません。コンバートするプログラムが出来てさえ、すべてのドキュメントをアップデートしなければならないという問題があります。Python に関する多くの本が出版されているので、それらを一発で不適切にするようなことはしたくないです。

仕様を変えなければならないのなら、緩やかなアップグレード計画が組まれなくてはなりません。PEP 5 で、ユーザーの受ける分裂を最小限にしながら後方互換性のない変更を行うために従うべき手順について説明しています。

1.2.6 Python は初心者プログラマに向いている言語ですか？

はい。

未だにプログラミング初学者にとって一般的なのは、Pascal、C、C++ のサブセット、Java など、手続き型の静的型付けの言語です。生徒にとっては、第一の言語として Python を学ぶのが役に立つでしょう。Python には非常に簡潔で一貫した構文と大きな標準ライブラリがあります。そして一番重要なことに、初心者プログラミングのコースで Python を学ぶことで、生徒は問題の分析やデータ型の設計など、重要なプログラミングスキルに集中することができるのです。Python なら、生徒はループや手続きなどの基本概念をすぐに取り入

られます。最初の課程でいきなりユーザ定義のオブジェクトを操ることさえできるでしょう。

プログラミングをしたことがない初心者にとって、静的型付け言語を使うのは不自然に思われます。習得すべき内容はより複雑になり、学習のペースが遅くなってしまいます。生徒は、コンピュータのように思考し、問題を分析し、一貫したインターフェースを設計して、データをカプセル化することを学ぶことになります。長期的に見れば静的型付け言語を学ぶことは重要ですが、それが最初のプログラミングの授業で学ぶべき最高の話題とは限りません。

Python の良いところは他にもたくさんあります。Python には、Java のように大きな標準ライブラリがあり、生徒が何かを **する** 段階で非常に早くプログラミングプロジェクトに参加できるようになります。宿題は標準の四則演算機や平均を求めるプログラムに限定されません。標準ライブラリを使用することで、生徒はプログラミングの原理を学びながら現実的なアプリケーションに取り組む満足感を得ることができます。また、標準ライブラリの使用からコードの再利用を学ぶことができます。PyGame などのサードパーティモジュールもまた手が届く範囲を広げるのに役立ちます。

Python のインタラクティブインタプリタによって、プログラミングをしながら、言語機能を確認することができます。別のウィンドウでプログラムのソースに入っている間、ウィンドウでインタプリタを起動させたままにしておくことができます。リストのメソッドを思い出せないときは、例えばこのようにできます:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 → 'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

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

インタプリタがあれば、プログラミングをしている間にドキュメントは生徒のそばを離れません。

Python のための良い IDE もあります。IDLE は Python で Tkinter を使って書かれたクロスプラットフォーム IDE です。Emacs には、ユーザにとって幸運なことに、素晴らしい Python モードがあります。これらすべてのプログラミング環境から、シンタックスハイライト、オートインデント、コーディング中のインタラクティブインタプリタへのアクセスが使えます。[the Python wiki](#) から Python 編集環境の一覧を参照してください。

Python の教育における利用についての議論がしたいなら、[the edu-sig mailing list](#) に参加するとよいでしょう。

プログラミング FAQ

2.1 一般的な質問

2.1.1 ブレークポイントやシングルステップ実行などを備えたソースコードレベルデバッガはありますか？

はい。

Python 用のデバッガについては次に解説してあり、組み込みの `breakpoint()` 関数でそれらのデバッガに処理を移せます。

`pdb` モジュールは簡素にして十分な Python のコンソールモードデバッガです。これは Python の標準ライブラリに含まれているもので、[ライブラリリファレンスマニュアル](#)にドキュメントがあります。`pdb` のコードを手本にして自分用のデバッガを書くこともできます。

Python に同梱されている統合開発環境の IDLE は通常の Python の配布形態の一部 (普通は `Tools/scripts/idle3` から利用可能) であり、グラフィカルなデバッガを含んでいます。

PythonWin は、`pdb` をベースとした GUI デバッガを含む Python IDE です。Pythonwin デバッガは、ブレークポイントの色付けや非 PythonWin プログラムのデバッグなどの素敵な機能をたくさん持っています。PythonWin は `pywin32` プロジェクトの一部、あるいは [ActivePython](#) ディストリビューションの一部として利用可能です。

[Eric](#) は PyQt や Scintilla editing component をもとにした IDE です。

[trepan3k](#) は gdb ライクなデバッガです。

[Visual Studio Code](#) はバージョン管理ソフトと一緒にあったデバッグツールを備えた IDE です

商業のグラフィカルデバッガ付き Python IDE もあります。例えば:

- [Wing IDE](#)
- [Komodo IDE](#)
- [PyCharm](#)

2.1.2 バグの発見や静的解析に役立つツールはありますか？

はい。

`Pylint` と `Pyflakes` は、バグの早期発見に役立つ基本的なチェックを行います。

`Mypy`, `Pyre`, `Pytype` などの静的型チェッカーは、Python ソースコードにある型ヒントをチェックできます。

2.1.3 どうしたら Python スクリプトからスタンドアロンバイナリを作れますか？

ユーザがダウンロードでき、Python ディストリビューションをインストールせずに実行できるようなスタンドアロンプログラムだけでいいなら、Python を C コードにコンパイルできなくても構いません。プログラムに対して必要なモジュールを選び、そのモジュールを Python バイナリに束縛して一つの実行可能ファイルにまとめる多くのツールがあります。

一つは `freeze` ツールで、Python ソースツリーに `Tools/freeze` として含まれています。これは Python バイトコードを C 配列に変換します。すべてのモジュールを新しいプログラムに埋め込む C コンパイラで、そのプログラムは Python モジュールにリンクされます。

これはあなたのソースの (両方の形式の) `import` 文を再帰的にスキャンし、`import` されたモジュールを標準の Python パスと (組み込みモジュールのある) ソースディレクトリから探します。そして Python で書かれたモジュールのバイトコードを C コード (`marshal` モジュールでコードオブジェクトに変換できる配列) に変換し、実際にそのプログラム内で使われている組み込みモジュールだけが含まれたカスタムメイドの設定ファイルを作成します。そして生成された C コードをコンパイルして Python インタプリタの残りと同じようにリンクし、元のスクリプトと全く同じように動作する自己完結的なバイナリを形成します。

以下のパッケージはコンソールや GUI の実行ファイル作成に役立ちます。

- `Nuitka` (Cross-platform)
- `PyInstaller` (Cross-platform)
- `PyOxidizer` (Cross-platform)
- `cx_Freeze` (Cross-platform)
- `py2app` (macOS only)
- `py2exe` (Windows only)

2.1.4 Python プログラムのためのコーディングスタンダードやスタイルガイドはありますか？

はい。標準ライブラリモジュールに求められるコーディングスタイルは **PEP 8** として文書化されています。

2.2 コア言語

2.2.1 なぜ変数に値があるのに `UnboundLocalError` が出るのですか？

もともと動いていたコードが、関数の本体のどこかに代入文を加えるという変更をしたら `UnboundLocalError` を出すのには驚くかもしれません。

このコード:

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

は動きますが、このコード:

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

は `UnboundLocalError` になります:

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

これは、あるスコープの中で変数に代入を行うとき、その変数はそのスコープに対してローカルになり、外のスコープにある同じ名前の変数を隠すからです。foo の最後の文が x に新しい値を代入しているので、コンパイラはこれをローカル変数であると認識します。その結果、先の `print(x)` が初期化されていないローカル変数を表示しようとして結果はエラーとなります。

上の例では、グローバルであると宣言することで外のスコープにアクセスできます:

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

この明示的な宣言は (表面的には似ているクラスとインスタンス変数の例とは違って) あなたは実際は他のスコープの変数の値を変えようとしているのだ、ということを知らせるのに必要です:

```
>>> print(x)
11
```

同様のことを、ネストされたスコープで `nonlocal` 予約語を使うことでもできます:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
...
>>> foo()
10
11
```

2.2.2 Python のローカルとグローバル変数のルールは何ですか？

Python では、関数内で参照されるだけの変数は暗黙的にグローバルとなります。関数の本体のどこかで値が変数に代入されたなら、それは明示的にグローバルであると宣言されない限り、ローカルであるとみなされます。

最初はちょっと驚くでしょうが、少し考えると納得できます。一方では、代入された変数に `global` を要求することで、意図しない副作用を防げます。他方では、グローバルな参照の度に `global` が要求されてしまうと、`global` を使っただけになってしまいます。ビルトイン関数やインポートされたモジュールの内容を参照するたびにグローバル宣言をしなければならないのです。その乱雑さは副作用を特定するための `global` 宣言の便利さよりも重大です。

2.2.3 ループの中で異なる値で定義されたラムダ式が、同じ値を返すのはなぜですか？

for ループを使って、少しずつ異なるラムダを定義 (もしくは簡単な関数) するとします。例えば:

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

これで `x**2` を計算する 5 つのラムダのリストが得られます。それら呼び出したとき、それぞれ 0、1、4、9、16 を返すと予想するかもしれませんが。しかし実際にやってみると、全て 16 が返ってくるのを目にします。

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

これは、`x` がラムダにとってのローカル変数ではなく外側のスコープで定義されていて、ラムダが定義されたときでなく呼び出されたときにアクセスされるために起こります。ループが終わった時点では `x` は 4 であり、

従って、全ての関数は $4**2$ つまり 16 を返します。このことは x の値を変えてみることで検証でき、ラムダの返り値がどのように変わるのか観察できます:

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

これを避けるためには、グローバルの x の値に依存しないために、ラムダにとってのローカル変数に値を保存する必要があります:

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

ここで、 $n=x$ は新しいラムダにとってのローカル変数 n を作成し、ラムダが定義されるときに計算されるので、ループのその時点での x と同じ値を持っています。これは、1 つ目のラムダでは n の値は 0 になり、2 つ目では 1、3 つ目では 2 以下同様、となることを意味します。従って、それぞれのラムダは今や正しい値を返すようになりました:

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

この動作はラムダに特有なものではなく、通常の関数にも適用されることに注意してください。

2.2.4 グローバル変数をモジュール間で共有するにはどうしたらいいですか？

一つのプログラムのモジュール間で情報を共有する正準な方法は、特別なモジュール (しばしば `config` や `cfg` と呼ばれる) を作ることです。単に設定モジュールをアプリケーションのすべてのモジュールにインポートしてください。このモジュールはグローバルな名前として使えます。それぞれのモジュールのただ一つのインスタンスがあるので、設定モジュールオブジェクトに対するいかなる変更も全体に反映されます。例えば:

`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)
```

なお、同じ理由から、モジュールを使うということは、シングルトンデザインパターンを実装することの基礎でもあります。

2.2.5 モジュールで `import` を使う際の「ベストプラクティス」は何ですか？

一般的に `from modulename import *` を使ってはいけません。そのようにするとインポータの名前空間は汚染され、`linter` が未定義の名前を発見することが難しくなります。

モジュールはファイルの先頭でインポートしてください。これによってコードが必要とする他のモジュールが明確になり、モジュール名がスコープに含まれるかどうか迷わなくなります。行に一つのインポートにすると、モジュールのインポートの追加と削除が容易になりますが、行に複数のインポートにすると画面の領域が少なく済みます。

次の手順でモジュールをインポートするのが、良い習慣になります：

1. 標準ライブラリモジュール -- 例 `sys`、`os`、`argparse`、`re`
2. サードパーティのライブラリモジュール (Python の `site-packages` ディレクトリにあるもの) -- 例 `dateutil`、`requests`、`PIL.Image`、など
3. 自前で開発したモジュール

循環参照の問題を避けるために、インポートを関数やクラスに移すことが必要なときもあります。Gordon McMillan によれば：

循環参照は両方のモジュールが `"import <module>"` 形式のインポートを使っていれば大丈夫です。二つ目のモジュールが最初のモジュールから名前を確保しようとして (`"from module import name"`)、そのインポートがトップレベルにあると駄目です。最初のモジュールが二つ目のモジュールをインポートするのに忙しくて、最初のモジュールの名前が利用可能になっていないからです。

この状況では、二つ目のモジュールが一つの関数の中でのみ使われているならば、そのインポートは簡単に関数の中に移せます。インポートが呼ばれたとき、最初のモジュールは初期化を完了していて、二つ目のモジュールは自分のインポートをできます。

プラットフォーム依存のモジュールがあるときには、インポートをトップレベルの外に動かすことも必要です。この場合、ファイルの先頭ではすべてのモジュールをインポートすることさえできないかもしれません。この場合は、対応するプラットフォームに合わせたコードで正しいモジュールをインポートすることを選ぶと良いです。

循環参照の問題を避けたりモジュールの初期化にかかる時間を減らしたりしたいなら、単にインポートを関数定義の中などのローカルなスコープに移してください。この手法は多くのインポートがプログラムがどのように実行されるかに依存しなくてよいときに特に有効です。ある関数の中でのみモジュールが使われるのなら、インポートをその関数の中に移すことを考えてもいいでしょう。なお、モジュールを読み込む最初の回はモジュールの初期化の時間のために高価になりえますが、複数回目にモジュールを読み込むのは事実上無料、辞

書探索の数回のコストだけで済みます。モジュール名がスコープから外れてさえ、そのモジュールはおそらく `sys.modules` から利用できるでしょう。

2.2.6 なぜオブジェクト間でデフォルト値が共有されるのですか？

この種のバグがよく初心者プログラマに噛み付きます。この関数を考えてみてください:

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

初めてこの関数を呼び出した時、`mydict` には一つの要素があります。二回目には、`foo()` が実行されるときに `mydict` には初めから一つの要素をすでに持っているので、`mydict` には二つの要素があります。

関数の呼び出しによって、デフォルトの値に対する新しいオブジェクトが作られるのだと予想しがちです。実はそうなりません。デフォルト値は、関数が定義されたときに一度だけ生成されます。この例の辞書のよう、そのオブジェクトが変更されたとき、その後の関数の呼び出しは変更後のオブジェクトを参照します。

定義の時に、数、文字列、タプル、`None` など、イミュータブルなオブジェクトを使うと変更される危険がありません。辞書、リスト、クラスインスタンスなどのミュータブルなオブジェクトは混乱のもとです。

この性質から、ミュータブルなオブジェクトをデフォルト値として使わないプログラミング手法がいいです。代わりに、`None` をデフォルト値に使い、そのパラメタが `None` である時にだけ、関数の内部で新しいリスト/辞書/その他をつくるようにしてください。例えば、こう書かずに:

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

代わりに:

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

この性質が便利なこともあります。時間のかかる計算を行う関数があるときに使われる一般的な技法は、関数が呼び出されるごとにパラメタと結果の値をキャッシュし、再び同じ値が要求されたらキャッシュされた値を返すというものです。これは "memoizing" と呼ばれ、このように実装されます:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
```

(次のページに続く)

(前のページからの続き)

```
_cache[(arg1, arg2)] = result          # Store result in the cache
return result
```

デフォルト値の代わりに、辞書を含むグローバル変数も使えます。これは好みの問題です。

2.2.7 オプションパラメータやキーワードパラメータを関数から関数へ渡すにはどうしたらいいですか？

関数のパラメータリストに引数を `*` と `**` 指定子 (specifier) で集めてください。そうすれば、位置引数をタプルとして、キーワード引数を辞書として得られます。これで、他の関数を呼び出すときに `*` と `**` を使ってそれらの引数を渡せます:

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

2.2.8 実引数と仮引数の違いは何ですか？

仮引数 (*parameter*) は関数定義に表れる名前で定義されるのに対し、**実引数** (*argument*) は関数を呼び出すときに実際に渡す値のことです。仮引数は関数が受け取ることの出来る **実引数の型** を定義します。例えば、以下のような関数定義があったとして:

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

`foo`、`bar`、`kwargs` は `func` の仮引数です。一方、`func` を呼び出すときには、例えば:

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

42、314、`somevar` という値は実引数です。

2.2.9 なぜ list 'y' を変更すると list 'x' も変更されるのですか？

次のようなコードを書いたとします:

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

どうして `y` への要素の追加が `x` も変更してしまうのか疑問に思うかもしれません。

このような結果になる 2 つの要因があります:

- 1) 変数とは、単にオブジェクトを参照するための名前に過ぎません。`y = x` とすることは、リストのコピーを作りません -- それは `x` が参照するのと同じオブジェクトを参照する新しい変数 `y` を作ります。つまり、あるのは一つのオブジェクト (この場合リスト) だけであって、`x` と `y` の両方がそれを参照しているのです。
- 2) リストは *mutable* です。内容を変更出来る、ということです。

`append()` 呼び出しの後、ミュータブルオブジェクトの内容が `[]` から `[10]` に変わります。変数が同じオブジェクトを参照しているので、どちらの名前であっても変更された値 `[10]` にアクセスします。

代わりに `x` にイミュータブルを代入すると:

```
>>> 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
```

この場合ご覧の通り `x` と `y` はまったく同じではありませんね。これは整数が *immutable* だからで、`x = x + 1` は整数の 5 の値を変更しているわけではありません; 代わりに新しいオブジェクト (整数 6) を作って `x` に代入しています (つまり `x` が参照するオブジェクトが変わります)。この代入の後では私たちは 2 つのオブジェクト (整数の 6 と 5) を持っていて、2 つの変数はそれらを参照しています (`x` はいまや 6 を参照していますが `y` は 5 を参照したままです)。

ある演算 (たとえば `y.append(10)`, `y.sort()`) がオブジェクトを変更する一方で、外見上は似た演算 (たとえば `y = y + [10]`, `sorted(y)`) は新しいオブジェクトを作ります。Python では一般に (そして標準ライブラリの全てのケースで)、このような 2 つのタイプの演算にまつわる混乱を避けるために、オブジェクトを変更するメソッドは `None` を返します。ですからもしあなたが誤って `y` の複製の並び替えをするつもりで `y.sort()` と書いた場合に結果手にするのは `None` でしょうから、あなたのプログラムは簡単に診断出来るエラーを起こすでしょう。

しかしながら、同じ操作が型ごとに異なる振る舞いをする演算の種類が一つあります: 累算代入演算です。例えば `+=` はリストを変更しますが、タプルや整数は変更しません (`a_list += [1, 2, 3]` は `a_list.extend([1, 2, 3])` と同じ意味で、そして `a_list` を変更しますが、`some_tuple += (1, 2, 3)` と `some_int += 1` は新しいオブジェクトを作ります)。

言い換えると:

- ミュータブルなオブジェクト (`list`, `dict`, `set`, 等) を持っている場合、私たちはその内容を変更するある種の演算を使うことが出来、それを参照している全ての変数はその変化を見ることになるでしょう。
- イミュータブルなオブジェクト (`str`, `int`, `tuple`, 等) を持っている場合、それを参照している全ての変数は同じ値を参照しているでしょうが、持っている値を新しい値に変換する演算はいつでも新しいオ

プロジェクトを返します。

2つの変数が同じオブジェクトを参照しているかどうかを知りたいければ、`is` 演算子や組み込み関数 `id()` が使えます。

2.2.10 出力引数のある関数 (参照渡し) はどのように書きますか？

前提として、Python では引数は代入によって渡されます。代入はオブジェクトへの参照を作るだけなので、呼び出し元と呼び出し先にある引数名の間にエイリアスはありませし、参照渡しそれ自体はありません。望む効果を得るためには幾つかの方法があります。

- 1) 結果のタプルを返すことによって:

```
>>> def func1(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
>>> func1(x, y)
('new-value', 100)
```

これはたいてい一番明確な方法です。

- 2) グローバル変数を使って。これはスレッドセーフでないので、推奨されません。
- 3) ミュータブルな (インプレースに変更可能な) オブジェクトを渡すことによって:

```
>>> def func2(a):
...     a[0] = 'new-value'        # 'a' references a mutable list
...     a[1] = a[1] + 1           # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

- 4) 変更される辞書に渡すことによって:

```
>>> 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)
>>> args
```

(次のページに続く)

(前のページからの続き)

```
{'a': 'new-value', 'b': 100}
```

5) または、クラスインスタンスに値を同梱することによって:

```
>>> class Namespace:
...     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 Namespace
...     args.b = args.b + 1           # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

このような複雑なことをする理由はめったに無いでしょう。

一番の選択は、複数の結果を含むタプルを返すことです。

2.2.11 Python で高次関数はどのようにつくりますか？

二つの方法があります: ネストされたスコープを使う方法と、呼び出し可能オブジェクトを使う方法です。例えば、 $a \cdot x + b$ の値を計算する $f(x)$ 関数を返す `linear(a,b)` を定義したいとします。ネストされたスコープを使うと:

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

また、呼び出し可能オブジェクトを使うと:

```
class linear:

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

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

どちらの場合でも、

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

とすれば、`taxes(10e6) == 0.3 * 10e6 + 2` となるような呼び出し可能オブジェクトを得られます。

呼び出し可能オブジェクトを使う方法は、少し遅くなり、わずかにコードが長くなるという短所があります。ですが、継承を使って呼び出し可能オブジェクト同士で記号を共有することもできます：

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

オブジェクトはいくつかのメソッドに状態をカプセル化できます：

```
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
```

ここで、`inc()`、`dec()`、`reset()` は同じカウント変数を共有する関数のようにふるまいます。

2.2.12 Python のオブジェクトはどのようにコピーしますか？

一般的に、普通は `copy.copy()` や `copy.deepcopy()` を試してください。何でもコピーできるとは限りませんが、たいていはできます。

もっと簡単にコピーできるオブジェクトもあります。辞書には `copy()` メソッドがあります：

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

シーケンスはスライシングでコピーできます：

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


2.2.13 オブジェクトのメソッドや属性はどのように見つけますか？

ユーザー定義クラスのインスタンス `x` で、`dir(x)` はインスタンス属性とそのクラスで定義されたメソッドや属性を含む名前のアルファベット順リストを返します。

2.2.14 コードはどのようにオブジェクトの名前を見つけるのですか？

概して、オブジェクトは本当は名前を持たないので、見つけることはできません。本質的には、代入といつも値に名前を束縛することです。 `def` と `class` 文も同じですが、この場合は値はコーラブルです。以下のコードを考えてみましょう：

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

おそらく、このクラスには名前があります。このクラスは二つの名前に縛られて、名前 `B` を通して呼び出されますが、それでもクラス `A` のインスタンスとして報告されるのです。しかし、両方の名前が同じ値に束縛されている以上、このインスタンスの名前が `a` か `b` か決めることはできないのです。

概して、コードにとってある値の「名前を知っている」事は重要ではありません。あなたがわざと内省的なコードを書いているのでない限り、方針を変えた方がいいかもしれないということになるでしょう。

`comp.lang.python` で、Fredrik Lundh はこの問題の答えとして素晴らしい喩えをしてくれました：

玄関にいた猫の名前を知るのと同じ方法です：その猫（オブジェクト）自体はその名前を言うことができないし、それは実は問題ではありません -- その猫が何と呼ばれているかを知る唯一の方法は、すべての隣人（名前空間）にその猫（オブジェクト）が何と呼ばれているかを聞くことです。

……そして、その猫が沢山の名前で知られていたり、逆に全く名前が無かったりしても驚かないでください！

2.2.15 カンマ演算子はなぜ優先されるのですか？

カンマは Python では演算子ではありません。このセッションを考えてください：

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

カンマは演算子ではなく、式の分離子なので、上の式は次の式と同じように評価されます：

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

こうではありません:

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

他のさまざまな演算子 (=, += など) も同じです。これらは真の演算子ではありませんが、代入文の構文上のデリミタです。

2.2.16 C の "?:" 三項演算子と等価なものがありますか？

はい、あります。構文は以下のようになります:

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

x, y = 50, 25
small = x if x < y else y
```

この構文が導入された 2.5 以前のバージョンに関しては、論理演算子を使ったこのイディオムが一般的でした:

```
[expression] and [on_true] or [on_false]
```

しかし、このイディオムは安全ではありません。`on_true` のブール値が偽であるときに間違った結果を与えることがあります。ですから、いつでも `... if ... else ...` 形式を使ったほうが良いです。

2.2.17 Python で解し難いワンライナーを書くことはできますか？

はい。そういうものはたいてい、`lambda` の中に `lambda` がネストされています。Ulf Bartelt によるものを少しアレンジした下の 3 つの例を見てください:

```
from functools import reduce

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

# First 10 Fibonacci numbers
print(list(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+'\n'+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,
```

(次のページに続く)

(前のページからの続き)

```
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"
#          /          / \_ _ _ _ _ range on y axis
#          / \_ _ _ _ _ range on x axis
```

よい子はまねしないでね！

2.2.18 関数の引数リストにあるスラッシュ (/) は何を意味しますか？

パラメータ関数の仮引数にあるスラッシュは、それより前にある仮引数は位置専用引数であることを示します。位置専用引数は、外で使える名前を持たない仮引数です。位置専用引数を受け付ける関数の呼び出しにおいて、実引数はその位置だけに基づいて仮引数に対応付けられます。例えば、`divmod()` は位置専用引数を受け付けます。そのドキュメントは次のようになります：

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

この引数リストの末尾にスラッシュは、この2つの仮引数が両方とも位置専用引数であることを意味します。したがって、`divmod()` をキーワード引数を使って呼び出すとエラーになります：

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

2.3 数と文字列

2.3.1 十六進数や八進数を指定するにはどうしたらいいですか？

八進数を指定するには、八進数での値の先頭に0と"o" (小文字または大文字) を加えてください。たとえば、変数"a"に八進数での"10" (十進数での"8") を代入するには、こう打ってください：

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

十六進数も簡単です。ただ十六進数での値の先頭に 0 と "x" (小文字または大文字) を加えてください。十六進数は小文字でも大文字でも指定できます。たとえば、Python インタプリタで:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.3.2 なぜ `-22 // 10` は `-3` を返すのですか？

`i % j` が `j` と同じ符号であってほしいに基づいています。それに加えて以下のようにもしたいとすると:

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

整数除算は床を返すことになります。C にも C の一貫性があって、`i % j` が `i` と同じ符号を持つように `i // j` を丸めています。

`i % j` は、`j` が負の時には実際にはほとんど使いません。`j` が正なら、たくさん使います。その事実上すべての場合、`i % j` は `>= 0` となる方が便利です。時計が 10 時を指している時、その 200 時間前は何時でしょうか。`-190 % 12 == 2` となるのが便利です。`-190 % 12 == -10` は噛み付きかねないバグです。

2.3.3 どうすれば `SyntaxError` を起こさずに整数リテラルの属性を得られますか？

ふつうの方法で `int` リテラルの属性を探そうとすると、ピリオドが小数点とみなされ、`SyntaxError` となります:

```
>>> 1.__class__
File "<stdin>", line 1
  1.__class__
    ^
SyntaxError: invalid decimal literal
```

解決策は、リテラルとピリオドをスペースや括弧で分けることです。

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

2.3.4 文字列を数に変換するにはどうしたらいいですか？

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to a floating-point number, e.g. `float('144') == 144.0`.

デフォルトでは、これらは数を十進数として解釈するので、`int('0144') == 144` は成立しますが `int('0x144')` は `ValueError` を送出します。`int(string, base)` はオプションの第二引数をとって変換元の基数にします。つまり `int('0x144', 16) == 324` です。基数が 0 と指定された場合、その数は Python の基準によって解釈されます。先頭が '0o' なら八進数で、'0x' なら十六進数を表します。

文字列を数に変換するだけのために `eval()` を使わないでください。`eval()` は特に遅いですし、セキュリティ上のリスクもあります。求められない副作用を持つような Python の式を渡そうとする人がいるかも知れません。例えば、あなたのホームディレクトリを消去する `__import__('os').system("rm -rf $HOME")` を渡そうとする人がいるかも知れません。

`eval()` にも数を Python の式として解釈する機能があります。だから例えば、`eval('09')` は構文エラー起こします。Python は ('0' 以外の) 十進数を '0' で始めてはならないからです。

2.3.5 数を文字列に変換するにはどうしたらいいですか？

例えば、144 という数を '144' という文字列に変換したいなら、組み込み型のコンストラクタ `str()` を使ってください。十六進数や八進数にしたければ、組み込み関数の `hex()` や `oct()` を使ってください。手の込んだフォーマット形式を使うなら、f-strings と formatstrings の節を参照してください。例えば、`"{:04d}".format(144)` は '0144' になり、`"{: .3f}".format(1.0/3.0)` は '0.333' になります。

2.3.6 文字列をインプレースに変更するにはどうしたらいいですか？

文字列はイミュータブルなので、それはできません。殆どの場合、組み立てたい個別の部品から単純に新しい文字列を構成するべきです。しかし、Unicode データをインプレースに変更できるオブジェクトが必要なら、`array` モジュールの `io.StringIO` オブジェクトを試してください:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('w', s)
>>> print(a)
```

(次のページに続く)

(前のページからの続き)

```
array('w', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('w', 'yello, world')
>>> a.tounicode()
'yello, world'
```

2.3.7 関数やメソッドを呼ぶのに文字列を使うにはどうしたらいいですか？

様々なテクニックがあります。

- 一番いいのは、文字列に関数に対応させる辞書を使うことです。このテクニックの一番の利点は、文字列が関数の名前と同じ必要がないことです。この方法は case 構造をエミュレートするための一番のテクニックでもあります:

```
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
```

- 組み込み関数の `getattr()` を使う方法:

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

なお、`getattr()` はクラス、クラスインスタンス、モジュールなど、どんなオブジェクトにも使えます。

これは標準ライブラリでも何箇所か使われています。このように:

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

    def do_bar(self):
        ...

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

- `locals()` を使って関数名を決める方法:

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

fname = "myFunc"

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

2.3.8 文字列から後端の改行を取り除く Perl の `chomp()` に相当するものはありますか？

`S.rstrip("\r\n")` を使って文字列 `S` の終端から他の空白文字を取り除くことなくすべての行末記号を取り除くことができます。文字列 `S` が複数行を表し、終端に空行があるとき、そのすべての空行も取り除かれます:

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

これは典型的に一度に一行ずつテキストを読みたい時にのみ使われるので、`S.rstrip()` をこの方法で使うとうまくいきます。

2.3.9 `scanf()` や `sscanf()` と同等なものがありますか？

そのようなものはありません。

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional "sep" parameter which is useful if the line uses something other than whitespace as a separator.

もっと複雑な入力解析をしたいなら、C の `sscanf` よりも正規表現の方が便利ですし、この処理に向いています。

2.3.10 'UnicodeDecodeError' や 'UnicodeEncodeError' はどういう意味ですか？

`unicode-howto` を参照して下さい。

2.3.11 raw string を奇数個のバックスラッシュで終えることはできますか？

奇数個のバックスラッシュで終わる raw string は文字列のクォートをエスケープします:

```
>>> r'C:\this\will\not\work\'
File "<stdin>", line 1
```

(次のページに続く)

(前のページからの続き)

```
r'C:\this\will\not\work\'
~
SyntaxError: unterminated string literal (detected at line 1)
```

これにはいくつかの回避策があります。ひとつは通常文字列と二重バックスラッシュを使うことです:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

もうひとつは、エスケープされたバックスラッシュを含む通常の文字列を、raw string に連結します:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

Windows では、`os.path.join()` を使ってバックスラッシュを追加することもできます:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

注意点として、バックスラッシュは raw string 終端のクォートを「エスケープ」しますが、raw string の値を解釈する際にはエスケープが生じません。つまり、バックスラッシュは raw string 値の中に残ったままになります:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

言語リファレンス の仕様も参照してください。

2.4 性能

2.4.1 プログラムが遅すぎます。どうしたら速くなりますか？

これは、一般的に難しい問題です。まず、先に進む前に覚えておいて欲しいことをここに挙げます:

- 性能の傾向は Python 実装によって変わります。この FAQ では *CPython* に焦点を当てます。
- 振る舞いはオペレーティングシステムによって変わります。特に、I/O やマルチスレッドに関しては顕著です。
- 常に、コードの最適化を始める **前に** プログラムのホットスポットを見つけるべきです (profile モジュールを参照してください)。
- ベンチマークスクリプトを書くことで、改善箇所の検索を素早く繰り返せます (timeit モジュールを参照してください)。
- 洗練された最適化に隠れたリグレッションの可能性を生む前に、(ユニットテストやその他の技法で) コードカバレッジを上げることを強く推奨します。

とは言っても、Python コードを高速化する技法はたくさんあります。ここでは、満足な性能のレベルにたどり着くまでの長い道のりを進む、一般的な方針を示します:

- コード中に細かい最適化の技法をばらまこうとするよりも、アルゴリズムを高速化 (または高速なアルゴリズムに変更) するほうが、大きな利益を生むことがあります。
- 適切なデータ構造を使ってください。bltin-types や collections を調べてください。
- 何かをするための基本要素が標準ライブラリにあるなら、自分で発明した代用品よりもそちらのほうが、(絶対には言えませんが) おそらく速いです。それが組み込み型やある種の拡張型のように C で書かれたものならなおさらです。たとえば、ソートするには、必ず `list.sort()` 組み込みメソッドか `sorted()` 関数を使ってください (また、中程度に高度な例は、`sortinghowto` を参照してください)。
- 抽象化は、遠回りにしがちで、インタプリタの作業を増やすことになります。この遠回りさが、なされる作業の量より重大になると、プログラムが遅くなってしまいます。過度な抽象化、特に細かい関数やメソッドの形で現れるもの (これは読みにくさも落とします) は防ぐべきです。

pure Python にできる限界に達したなら、更に進むためのツールがあります。例えば、[Cython](#) は、Python コードのわずかに変形した版を C 拡張にコンパイルし、多種のプラットフォームで使えます。Cython は、コンパイル (と任意の型アノテーション) を利用し、コードの解釈を大幅に速くします。C プログラミングに自信があるなら、自分で write a C extension module こともできます。

➡ 参考

パフォーマンス tips が載っている wiki のページ。

2.4.2 多くの文字列を結合するのに最も効率的な方法は何ですか？

`str` および `bytes` オブジェクトはイミュータブルなので、多くの文字列の結合は結合ごとに新しいオブジェクトを作成し、効率が悪いです。一般的に、全体の実行時間のコストは文字列の長さの二乗に比例します。

多くの `str` オブジェクトを累積するのにおすすめのイディオムは、すべてをリストに配置してから最後に `str.join()` を呼び出すことです:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(他の割と効率的なイディオムは、`io.StringIO` を使うことです)

多くの `bytes` オブジェクトを累積するのにおすすめのイディオムは、`bytearray` オブジェクトをインプレース結合 (`+=` 演算子) で拡張することです:

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 シーケンス (タプル/リスト)

2.5.1 タプル、リスト間の変更はどのようにするのですか？

型コンストラクタ `tuple(seq)` はすべてのシーケンス (実際には、すべてのイテラブル) を同じ要素、同じ順序のタプルに変換します。

例えば、`tuple([1, 2, 3])` は `(1, 2, 3)` を与え、`tuple('abc')` は `('a', 'b', 'c')` を与えます。引数がタプルなら、コピーを作らずに引数のオブジェクトそのものを返すので、あるオブジェクトが既にタプルになっているか確信が持てないのなら、`tuple()` を呼ぶのが手軽です。

型コンストラクタ `list(seq)` はすべてのシーケンスあるいはイテラブルを同じ要素、同じ順序のリストに変換します。例えば、`list((1, 2, 3))` は `[1, 2, 3]` を与え、`list('abc')` は `['a', 'b', 'c']` を与えます。引数がリストなら、`seq[:]` と同様にコピーを作ります。

2.5.2 負の添え字は何ですか？

Python のシーケンスは正の数と負の数でインデックスされます。正の数では、0 が最初のインデックス、1 が 2 番目のインデックス、以下も同様です。負のインデックスでは、-1 が最後のインデックス、-2 が最後から 2 番目のインデックス、以下も同様です。`seq[-n]` は `seq[len(seq)-n]` と同じだと考えてください。

負のインデックスを使うと便利ことがあります。例えば、`S[:-1]` は文字列の最後以外のすべての文字を表すので、文字列の末尾の改行を取り除くときに便利です。

2.5.3 シーケンスを逆順にイテレートするにはどうしたらいいですか？

組み込み関数 `reversed()` を使ってください。

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

これは元のシーケンスをいじるのではなく、逆順の新しいコピーを作ってイテレートさせます。

2.5.4 リストから重複を取り除くにはどうしますか？

Python Cookbook の長い議論に多くの方法があるので参照してください:

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

リストを並び替えて構わないのなら、ソートした上でリストの最初から最後までを調べ、次のように重複を削除してください:

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

(次のページに続く)

(前のページからの続き)

```
else:
    last = mylist[i]
```

リストのすべての要素が集合のキーとして使える (つまり、すべての要素が *hashable*) なら、おそらくこのほうが速いです

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

リストを集合に変換するときに重複は取り除かれるので、それをリストに戻せばいいのです。

2.5.5 リストから複数のアイテムを取り除く方法

重複除去と同様に、削除条件を付けて明示的に逆回すのも一つの方法です。ですが、明示的/暗黙的な反復でスライス置換する方が速くて簡単です。こちらは3つのバリエーションです。:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

リスト内包表記がおそらく最も高速です。

2.5.6 Python で配列を作るにはどうしますか？

リストを使ってください:

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

リストの時間計算量は C や Pascal の配列と同じです。大きな違いは、Python のリストは多くの異なる型のオブジェクトを含めることです。

`array` モジュールにも固定された型を簡潔に表現する配列を作るためのメソッドがありますが、リストよりもインデックスが遅いです。また、`NumPy` やその他のサードパーティー拡張でも、様々な特徴をもつ配列的な構造体が定義されています。

Lisp 方式の連結リストを得るのに、タプルを使って *cons cells* をエミュレートできます:

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

ミュータブルな必要があるなら、タプルではなくリストを使いましょう。Lisp の *car* にあたるものが `lisp_list[0]` で、*cdr* にあたるものが `lisp_list[1]` です。本当に必要だと確信できるとき以外はこれはいしないでください。たいてい、これは Python のリストを使うよりも非常に遅いからです。

2.5.7 多次元のリストを作るにはどうしますか？

このようにして多次元の配列を作ろうとしてしまったことがあるでしょう：

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

これを表示したときには問題なさそうに見えます：

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

しかし値を代入すると、その値が複数の場所に現れてしまいます：

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

これは、`*` を使ったリストの複製がコピーを作らず、存在するオブジェクトへの参照を作るだけだからです。この `*3` は長さ 2 の同じリストへの参照を含むリストを作ります。一つの列に対する変更はすべての列に現れますが、これが望んだ結果であることはまずないでしょう。

おすすめの方法は、最初に望んだ長さのリストを作り、それから新しく作ったリストでそれぞれの要素を埋めていくことです：

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

これは長さ 2 の異なるリスト 3 つを含むリストを生成します。リスト内包表記も使えます：

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

あるいは、行列データ型を提供している拡張を使用することもできます；`NumPy` が最もよく知られています。

2.5.8 オブジェクトのシーケンスにメソッドや関数を適用するにはどうしますか？

メソッドや関数を呼び出して結果を蓄積するには、*list comprehension*（リスト内包表記）がエレガントな解決策です：

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

result = [function(obj) for obj in mylist]
```

戻り値を保存せずにメソッドや関数を実行するだけなら、ただの `for` ループで十分です：

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

2.5.9 なぜ加算はされるのに `a_tuple[i] += ['item']` は例外を送出するのですか？

これは、累算代入演算子は **代入** 演算子だ、という事実と、Python での可変オブジェクトと不変オブジェクトの違いが組み合わさって起きるのです。

この議論は一般的に、可変オブジェクトを指すタプルの要素に、累算代入演算子が適用されたときにも適用できますが、例として `list` と `+=` を使います。

次のように書いたとします:

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

例外が送出された理由は明らかです: 1 が (1) を指すオブジェクト `a_tuple[0]` に加えられ、結果のオブジェクト 2 が生成されますが、計算結果 2 をタプルの第 0 要素に代入しようとしたときに、エラーが発生します。なぜならば、タプルの要素が何を指すかは変えられないからです。

このような裏事情の元、累算代入文はだいたい次のようなことをしています:

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

タプルは不変なので、例外を生み出しているのは操作の代入部分なのです。

次のように書いたとします:

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

この例外にはちょっと驚きますが、もっと驚くべきことは、エラーがあったとしても追記はきちんと動いている、という事実です:

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

なぜこれが起きるかを調べるためには、次の 2 点を知っている必要があります。(a) オブジェクトに `__iadd__()` 特殊メソッドが実装されている場合、拡張代入 `+=` が実行されるときにそれが呼び出され、その返り値が代入文で使われます; (b) リストでは、`__iadd__()` は `extend()` の呼び出しと等価で、リストを返します。こんな理由で、リストでは `+=` は `list.extend()` の ”略記” だと言ったのでした:

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

これは次と等価です:

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

`a_list` が指していたオブジェクトは更新され、更新されたオブジェクトへのポインタは再度 `a_list` に代入されます。代入しているのは、`a_list` が更新前まで指していた同じオブジェクトへのポインタなので、代入は最終的には何もしていないのですが、代入処理自体は起きています。

従って、今のタプルの例では、次のと同じことが起きています:

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

`__iadd__()` は成功し、リストは拡張 (`extend`) されますが、`result` が `a_tuple[0]` が既に指しているオブジェクトと同じオブジェクトを指していたとしても、タプルは不変なので、その最後の代入はやはりエラーとなります。

2.5.10 複雑なソートがしたいのですが、Python でシュワルツ変換はできますか？

Perl コミュニティの Randal Schwartz の作とされるこのテクニックは、リストの要素を、それぞれの要素をその「ソート値」に対応付けるメトリックによってソートします。Python では、`list.sort()` メソッドに `key` 引数を使ってください:

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

2.5.11 リストを別のリストの値によってソートするにはどうしますか？

二つのイテレータを混ぜあわせてタプルのイテレータにしてから、必要な要素を選んでください。

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

2.6 オブジェクト

2.6.1 クラスとは何ですか？

クラスは、class 文の実行で生成される特殊なオブジェクトです。クラスオブジェクトはインスタンスオブジェクトを生成するためのテンプレートとして使われ、あるデータ型に特有のデータ (attribute/属性) とコード (メソッド) の両方を内蔵しています。

新しいクラスを一つ以上の他のクラス (新しいクラスの基底クラスと呼ばれます) に基づいて作ることもできます。この新しいクラスは、基底クラスから属性とメソッドを継承します。これにより、オブジェクトモデルを継承で連続的に洗練できます。メールボックスへの基本的なアクセサを提供する一般的な Mailbox クラスを作って、それからいろいろな特定のメールボックスの形式を扱う MboxMailbox、MaildirMailbox、OutlookMailbox のようなサブクラスを作れるのです。

2.6.2 メソッドとは何ですか？

メソッドは、オブジェクト x が持つ関数で、通常 x.name(arguments...) として呼び出されるものです。メソッドはクラス定義の中で関数として定義されます:

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

2.6.3 self とは何ですか？

self はメソッドの第一引数に慣習的につけられる名前にすぎません。meth(self, a, b, c) として定義されたメソッドは、その定義がなされたクラスのインスタンス x に対して x.meth(a, b, c) として呼び出されます。呼び出されたメソッドは、meth(x, a, b, c) が呼ばれたものと考えます。

なぜメソッドの定義や呼び出しにおいて 'self' を明示しなければならないのですか？ も参照してください。

2.6.4 あるオブジェクトが、与えられたクラスやそのサブクラスのインスタンスであるかを調べるにはどうしますか？

ビルトイン関数 `isinstance(obj, cls)` を使ってください。クラスのタプルを与えて `isinstance(obj, (class1, class2, ...))` のようにすれば、あるオブジェクトが任意の数のクラスのオブジェクトであるかを調べられますし、`isinstance(obj, str)` や `isinstance(obj, (int, float, complex))` のようにすれば、Python のビルトイン型のオブジェクトであるかも調べられます。

注意：`isinstance()` は *abstract base class* (抽象基底クラス) からの仮想継承もチェックします。そのため、登録 (register) されたクラスが直接/間接的にそれを継承していなくても `True` を返します。「真のインスタンス」をテストする場合は、クラスの *MRO* を調べます：

```
from collections.abc import Mapping

class P:
    pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)           # direct
True
>>> isinstance(c, P)           # indirect
True
>>> isinstance(c, Mapping)     # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

なお、大部分のプログラムでは、`isinstance()` をユーザー定義のクラスに何度も使うべきではありません。クラスを自分で開発するときに、適切なオブジェクト指向スタイルは、特定の振る舞いをカプセル化するクラスのメソッドを定義するものであって、オブジェクトのクラスを調べてそのクラスに応じて違うことをするものではありません。例えば、何かをする関数があったとして：

```
def search(obj):
```

(次のページに続く)

(前のページからの続き)

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

よりよいアプローチは、`search()` メソッドをすべてのクラスに定義して、それをただ呼び出すことです:

```
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 委譲とは何ですか？

委譲 (delegation) とは、オブジェクト指向のテクニック (デザインパターンとも呼ばれる) の一つです。オブジェクト `x` があって、そのメソッドのうちただ一つの振る舞いを変えたいとしましょう。新しいクラスを作成し、変えたいメソッドだけを新しく実装し、他のすべてのメソッドを `x` の対応するメソッドに委譲する新しいクラスを作れます。

Python プログラマは簡単に委譲を実装できます。例えば、以下のクラスは、ファイルのように振る舞いながらすべての文字を大文字に変換するクラスを実装します:

```
class UpperOut:

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

    def write(self, s):
        self._outfile.write(s.upper())

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

ここで `UpperOut` クラスは `write()` メソッドを定義しなおして、引数の文字列を大文字に変換してから基礎となる `self._outfile.write()` メソッドを呼び出すようにします。その他すべてのメソッドは基礎となる `self._outfile` オブジェクトに移譲されます。この委譲は `__getattr__()` メソッドを通してなされます。属性の制御の詳細は [言語リファレンス](#) を参照してください。

なお、一般的に委譲はトリッキーになりがちです。属性が設定される時には読み出される時と同様に、そのクラスに `__setattr__()` メソッドを定義する必要があり、それには細心の注意が必要です。`__setattr__()` の基本的な実装はおおよそ以下のようになります:

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

Many `__setattr__()` implementations call `object.__setattr__()` to set an attribute on self without causing infinite recursion:

```
class X:
    def __setattr__(self, name, value):
        # Custom logic here...
        object.__setattr__(self, name, value)
```

Alternatively, it is possible to set attributes by inserting entries into `self.__dict__` directly.

2.6.6 基底クラスで定義されたメソッドを、そのクラスを継承した派生クラスから呼び出すにはどうしますか？

組み込みの `super()` 関数を使ってください:

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

この例では、`super()` は呼ばれたインスタンス (`self` の値) を自動判断し、`type(self).__mro__` で *method resolution order* (MRO メソッド解決順) を検索し、そして MRO 内で `Derived` の次行を返します: `Base`。

2.6.7 基底クラスの名前を変えやすいコードを書くにはどうしますか？

基底クラスをエイリアス (alias) に代入しておいてから、そのエイリアスを継承するといいかもかもしれません。そうすればエイリアスに代入する値を変えるだけで済みます。ちなみに、この手法は使用する基底クラスを動的に選びたいとき、例えば使えるリソースによって選びたいときなどにも便利です。例:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

2.6.8 静的なクラスデータや静的なクラスメソッドを作るにはどうしますか？

(C++ や Java の意味で) 静的なデータも静的なメソッドも Python でサポートされています。

静的なデータを作るには、単純にクラス属性を定義してください。その属性に新しい値を代入するには、代入するクラス名を明示する必要があります:

```
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 そのものや c.__class__ から C にいたるパス探索経路上のクラスによってオーバーライドされない限り、c.count も isinstance(c, C) であるすべての c に対する C.count を参照します。

注意: C のメソッド内では、self.count = 42 のような代入は self 自身の辞書に "count" という名前の新しく関係ないインスタンスを作ります。クラスの静的なデータの再束縛には、メソッド内であるか否かにかかわらず、いつもクラスを指定しなければなりません:

```
C.count = 314
```

静的メソッドが使えます:

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

しかし、静的メソッドの効果を得るもっと簡単な方法は、単にモジュールレベル関数を使うことです:

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

モジュールあたりに一つのクラスを定義するように (あるいはクラス組織を厳密に関連させるように) コードが構成されているなら、これで必要なカプセル化ができます。

2.6.9 Python でコンストラクタ (やメソッド) をオーバーロードするにはどうしたらいいですか？

この質問の答えはすべてのメソッドについて言えることですが、この質問はだいたい以下の構造の文脈から出てきます。

C++ では、このように書けます

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

Python では、一つのコンストラクタでデフォルトの引数を使ってすべての場合に対応するように書かなければなりません。例えば:

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

これで完全に等価とは言えませんが、実用上は十分に近いです。

長さが変えられる引数のリストを試すには、例えば

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

これと同じやり方がすべてのメソッド定義で使えます。

2.6.10 `__spam` を使おうとしたら `_SomeClassName__spam` からエラーがでました。

先頭にアンダースコアが二つ付いた変数名は、クラスのプライベートな変数を、“マングル化” という単純かつ効率のいい方法で定義します。`__spam` のような形式 (先頭に二つ以上、末尾にもしあっても一つのアンダースコアがある) のすべての識別子は、`classname` が先頭のアンダースコアをすべて削除した現在のクラス名とすれば、`_classname__spam` のように文字上で置換えられます。

The identifier can be used unchanged within the class, but to access it outside the class, the mangled name must be used:

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()

class B(A):
    def three(self):
        return 3 * self._A__one()
```

(次のページに続く)

(前のページからの続き)

```
four = 4 * A()._A_one()
```

In particular, this does not guarantee privacy since an outside user can still deliberately access the private attribute; many Python programmers never bother to use private variable names at all.

➡ 参考

詳細と特例は、[private name mangling specifications](#)。

2.6.11 クラスに `__del__` メソッドを定義しているのですが、オブジェクトを削除したときに呼ばれません。

いくつかの可能性があります。

`del` 文は必ずしも `__del__()` を呼び出すとは限りません -- これは単純にオブジェクトの参照カウントを減らすもので、カウントがゼロになったときに `__del__()` が呼び出されます。

データ構造が循環リンク (子のそれぞれが親の参照を持ち、親のそれぞれが子のリストを持つツリーなど) を含む場合、その参照カウントは決して 0 にはなりません。時々、Python はこのようなサイクルを検出するアルゴリズムを実行しますが、データ構造への参照がなくなってからこのガベージコレクタが実行されるまでいくらか時間が掛かるかもしれないので、`__del__()` メソッドは不都合な予期できないタイミングで呼び出されるかもしれません。これは問題を再現しようとするときに不便です。さらに悪いことに、オブジェクトの `__del__()` メソッドが実行される順序は任意です。`gc.collect()` を起動して収集を強制することができますが、オブジェクトが決して回収されないような本当に病的な場合も **あります**。

循環参照コレクタがあるとはいえ、オブジェクトに `close()` メソッドを明示的に定義し、使い終わったらいつでも呼び出せるようにするのはいいことです。`close()` メソッドを使うと、サブオブジェクトを参照している属性を取り除けます。`__del__()` を直接呼び出さないでください -- `__del__()` は `close()` を呼び出すでしょうし、`close()` なら同じオブジェクトに対して複数回呼ばれてもいいことが保証されているでしょう。

循環参照を避ける他の方法は、`weakref` モジュールを使って、参照カウントを増やすことなくオブジェクトを示すことです。例えばツリー構造は、親と (必要なら!) 兄弟に弱参照を使うべきです。

最後に、`__del__()` メソッドが例外を発生させた場合、警告のメッセージが `sys.stderr` に書きこまれます。

2.6.12 与えられたクラスのすべてのインスタンスのリストを得るにはどうしますか？

Python はクラス (やビルトイン型) のすべてのインスタンスをたどりません。クラスのコンストラクタにそれぞれのインスタンスへの弱参照のリストを作らせることですべてのインスタンスをたどらせられます。

2.6.13 なぜ `id()` の結果は一意でないように見えるのですか？

組み込みの `id()` は、オブジェクトが生存している間は一意なことが保証されている整数値を返します。CPython では、それはオブジェクトのメモリアドレスなので、オブジェクトがメモリから削除された後に、次に新しく生成されたオブジェクトはメモリの同じ場所にメモリ領域を確保されていることが、しばしば起きます。この現象を次の例で示しましょう：

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

2 つの同じ値を持つ `id` は `id()` の実行の前に作られてすぐさま削除された異なる整数オブジェクトによるものです。`id` を調べたいオブジェクトがまだ生きてることを保証したいなら、オブジェクトへの別の参照を作ってください：

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

2.6.14 いつ `is` 演算子での同一性テストが頼れますか？

`is` 演算子はオブジェクトの同一性をテストします。テスト `a is b` は `id(a) == id(b)` と同等です。

同一性テストの最も重要な特性は、オブジェクトは常にそれ自身と同一であり、`a is a` は常に `True` を返すということです。同一性テストは通常、等価性テストよりも高速です。また、等価性テストとは異なり、同一性テストは真偽値 `True` または “False” を返すことが保証されています。

ただし、同一性テストを等価性テストの代用とできるのは、オブジェクトの同一性が保証されている場合のみです。一般的に、同一性が保証される状況は3つあります：

- 1) 代入は新しい名前を作りますが、オブジェクト ID は変えません。`new = old` 代入の後、`new is old` が保証されます。
- 2) オブジェクト参照を格納するコンテナにオブジェクトを入れても、オブジェクト ID は変わりません。`s[0] = x` リスト代入の後、`s[0] is x` が保証されます。
- 3) オブジェクトがシングルトンなら、それはそのオブジェクトのインスタンスは1つだけ存在できることを意味します。`a = None` と `b = None` 代入の後、`a is b` が保証されます。`None` がシングルトンのためです。

多くの他の状況では、同一性テストは賢明でなく、等価性テストをお勧めします。特に、シングルトンである

ことが保証されていない `int` や `str` などの定数をチェックするために同一性テストを使わないでください。

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

同様に、ミュータブルなコンテナの新しいインスタンスは同一ではありません:

```
>>> a = []
>>> b = []
>>> a is b
False
```

標準ライブラリのコードには、同一性テストを正しく使うための一般的なパターンがあります:

- 1) **PEP 8** で推奨されるように、同一性テストは `None` のチェックの良い方法です。コードの中で平易な英語のように読めますし、`false` と評価される真偽値を持ちうる他のオブジェクトとの混同を避けます。
- 2) `None` が有効な入力値である場合、省略された引数を検出にはコストがいらいます。そのような状況では、他のオブジェクトと区別されることが保証されたシングルトンの番兵オブジェクトを作れます。例えば、これは `dict.pop()` のように振る舞うメソッドを実装する方法です:

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

- 3) コンテナの実装では、等価性テストを同一性テストで補強しないといけない場合があります。これは、`float('NaN')` のような自分自身と等価でないオブジェクトによってコードが混乱するのを防ぐためです。

例えば、これは `collections.abc.Sequence.__contains__()` の実装です:

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

2.6.15 どうすればサブクラスはイミュータブルなインスタンスに格納されたデータを制御できますか？

イミュータブル型をサブクラス化する場合、`__init__()` ではなく `__new__()` メソッドを継承します。前者はインスタンスが生成された 後 に動くため、イミュータブルなインスタンスのデータを変えるには遅すぎます。

これらのイミュータブルクラスはすべて、親クラスと異なるシグネチャを持ちます：

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

クラスはこのように使えます：

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
```

(次のページに続く)

(前のページからの続き)

20

```
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

2.6.16 メソッド呼び出しをキャッシュするには どうしたらいいですか？

メソッドキャッシュの主な道具は2つあり、`functools.cached_property()` と `functools.lru_cache()` です。前者はインスタンスレベル、後者はクラスレベルで結果を保存します。

`cached_property` アプローチは、引数を取らないメソッドでのみ働きます。これはインスタンスへの参照を作りません。キャッシュされたメソッドの結果は、インスタンスが活着ている間だけ保持されます。

利点は、インスタンスが使われなくなった場合、キャッシュされたメソッドの結果がすぐに解放されることです。欠点は、インスタンスが溜まると、溜められたメソッドの結果も増えてしまうことです。それらは際限なく増える恐れがあります。

`lru_cache` アプローチは、*hashable* な引数を持つメソッドで働きます。weak references を渡すための特別な努力がない限り、インスタンスへの参照を作ります。

least recently used アルゴリズム（訳注：LRU 直近に使われたものを残す）の利点は、キャッシュが指定された *maxsize* で制限されることです。欠点は、キャッシュから期限切れで追い出されるかクリアされるまで、インスタンスが生き続けることです。

この例は各種テクニックを示します：

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
```

(次のページに続く)

(前のページからの続き)

```
"Rainfall on a given date"
# Depends on the station_id, date, and units.
```

上の例では、`station_id` が常に変わらないことを前提としています。関連するインスタンスの属性が可変である場合、`cached_property` アプローチは属性の変更を検出できないため、機能しなくなります。

`station_id` が可変の場合に `lru_cache` アプローチを機能させるには、クラスで `__eq__()` と `__hash__()` メソッドを定義する必要があります。これにより、キャッシュは関連する属性の更新を検出できます:

```
class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

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

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.
```

2.7 モジュール

2.7.1 .pyc ファイルを作るにはどうしますか？

モジュールが初めてインポートされたとき (もしくは、現在のコンパイルされたファイルが作られてから、ソースファイルが変更されたとき)、コンパイルされたコードが入っている `.pyc` ファイルが、`.py` ファイルのあるディレクトリのサブディレクトリ `__pycache__` に作成されます。`.pyc` ファイルのファイル名は、`.py` ファイルの名前で始まり、`.pyc` で終わり、中間部分はこのファイルを作った `python` バイナリに依存した文字列になります。(詳細は [PEP 3147](#) を参照してください。)

`.pyc` が作られない理由の 1 つは、ソースファイルがあるディレクトリの権限の問題、つまり `__pycache__` サブディレクトリが作れない問題です。これは、例えば、ウェブサーバーでテストを行っているときのような、開発者のユーザと実行者のユーザが別な場合に、起こり得ます。

`PYTHONDONTWRITEBYTECODE` 環境変数がセットされない限り、モジュールをインポートしていて、Python に

`__pycache__` サブディレクトリを作り、そこにコンパイルされたモジュールが置ける能力 (権限、ディスクの空きスペース、など) がある場合は、`.pyc` ファイルは自動的に作られます。

最上位のスクリプトを Python で実行するのはインポートとはみなされず、`.pyc` は作成されません。例えば、最上位のモジュール `foo.py` が別のモジュール `xyz.py` をインポートしている場合、(シェルコマンドとして `python foo.py` と打ち込んで) `foo` を実行すると、`xyz` はインポートされるので `xyz` の `.pyc` は作成されますが、`foo.py` はインポートされたわけではないので `foo` の `.pyc` は作られません。

`foo` の `.pyc` ファイルを作成する -- つまり、インポートされていないモジュールの `.pyc` ファイルを作成する -- 必要がある場合、`py_compile` モジュールと `compileall` モジュールを使えば可能です。

`py_compile` モジュールは手動で任意のモジュールをコンパイルできます。やり方の一つは、このモジュールの `compile()` 関数をインタラクティブに実行することです:

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

このように実行すると、`foo.py` と同じ場所の `__pycache__` サブディレクトリに `.pyc` が書き出されます (出力ファイルの位置は、オプション引数 `cfile` で上書きすることもできます)。

`compileall` モジュールを使えば自動的に一つや複数のディレクトリのすべてのファイルをコンパイルできます。シェルプロンプトから `compileall.py` を起動して、コンパイルしたいファイルを含むディレクトリのパスを指定してください:

```
python -m compileall .
```

2.7.2 現在のモジュール名を知るにはどうしますか？

モジュールは前もって定義されたグローバル変数 `__name__` を検索することで自身の名前を決定できます。この値が `'__main__'` であるとき、そのプログラムはスクリプトとして実行されています。インポートされることによって使われる大抵のモジュールはコマンドラインインターフェースや自己テストも提供していて、`__name__` をチェックしてからそのコードだけを実行します:

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

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

2.7.3 相互にインポートしあうモジュールを作るにはどうしたらいいですか？

以下のモジュールがあったとしましょう:

`foo.py`:

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

bar.py:

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

問題はインタプリタが以下の段階を実行することです:

- main が foo をインポートする
- foo の空のグローバルが生成される
- foo がコンパイルされ実行を始める
- foo が bar をインポートする
- bar の空のグローバルが生成される
- bar がコンパイルされ実行を始める
- bar が foo をインポートする (すでに foo という名前のモジュールがあるので no-op となる)
- インポートメカニズムは foo のグローバルから foo_var を読んで、bar.foo_var = foo.foo_var に設定しようとします。

この最後の段階は失敗します。Python が foo を解釈し終わっていないで、foo のグローバルなシンボルの辞書はまだ空ですから。

import foo を使って、グローバルコードの foo.foo_var にアクセスしようとしたときにも、これと同じことが起こります。

この問題には (少なくとも) 三つの解決策があります。

Guido van Rossum は from <module> import ... を全く使わないで、すべてのコードを関数の中に入れることを勧めています。グローバル変数とクラス変数の初期化は定数とビルトイン関数のみで行われるべきです。これでインポートされたすべてのモジュールは <module>.<name> として参照されることになります。

Jim Roskind はそれぞれのモジュールに対して以下の順に進めることを提案しています:

- エクスポート (インポートされた基底クラスを必要としないグローバル、関数、クラス)
- import 文
- アクティブなコード (インポートされた値によって初期化されるグローバルを含む)。

インポートが奇妙な場所に現れることから Van Rossum はこの方法をそれほど好みませんが、これは有効です。

Matthias Urlichs は第一に再帰インポートが必要ないようにコードを構築しなすことを推奨しています。

これらの解決策はそれぞれ両立させることもできます。

2.7.4 `__import__('x.y.z')` は `<module 'x'>` を返しますが、`z` を得るためにはどうしますか？

`importlib` に `import_module()` という便利な関数があるので、代わりにそちらを使用することを検討してください。

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

2.7.5 インポートされたモジュールを編集してから再インポートしましたが、変化が現れません。なぜですか？

効率と一貫性上の理由から、Python はモジュールが最初にインポートされた時にのみモジュールファイルを読み込みます。そうしないと、たくさんのモジュールでできていて、それぞれが同じ基本モジュールをインポートしているようなプログラムでは、その基本モジュールの解析と再解析が繰り返されることになります。変更されたモジュールの再読込を強制するには、こうしてください：

```
import importlib
import modname
importlib.reload(modname)
```

注意: この手法は 100% 安全とは言えません。とりわけ

```
from modname import some_objects
```

のような文を含むモジュールは、インポートされたオブジェクトの古いバージョンを使い続けます。そのモジュールにクラス定義が含まれていたら、存在するクラスインスタンスは新しいクラス定義を使うようにアップデート **されません**。これによって以下の矛盾した振舞いがなされえます：

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

この問題の本質は、クラスオブジェクトの ”固有价值” を印字することで明らかになります：

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```


デザインと歴史 FAQ

3.1 Python はなぜ文のグループ化にインデントを使うのですか？

Guido van Rossum の信じるところによれば、インデントによるグループ化は非常にエレガントで、普通の Python プログラムを大いに読みやすくします。しばらくすればほとんどの人はこの仕様を気に入るようになります。

開始/終了の括弧がないので、構文解析器と人間の読者の間にグループ化の解釈の違いは起こりえません。時折、C のプログラマはこのようなコード片に出くわします：

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

条件式が真のとき、`x++` 行のみが実行されます。しかしインデントによって、多くの人が別のことを考えてしまいます。経験豊富な C プログラマでさえ、`x > y` なのに `y` がデクリメントされるのはなぜだろうと、このコードをしばらく凝視することがあります。

Python は開始/終了の括弧がないので、コーディングスタイルの争いに余り影響されません。C 言語では中括弧の置き方についてさまざまな流儀があります。特定のスタイルを使ってコードを読み書きするのに慣れたあと、別のスタイルでコードを読んだり (あるいは書く必要に迫られたり) するときに、何となく心配になるのはよくあることです。

多くのコーディングスタイルは `begin/end` の括弧にそれぞれ一行を使います。これではプログラムは冗長になって画面を浪費し、プログラムの見通しが悪くなります。一つの関数は一画面 (例えば 20 から 30 行) に収めるのが理想です。20 行の Python は 20 行の C よりもはるかに多くの作業ができます。これは `begin/end` の括弧がないからだけではなく -- 宣言が不要なことや高レベルなデータ型もその理由です -- が、インデントに基づく構文は確かに役に立っています。

3.2 なぜ単純な算術演算が奇妙な結果になるのですか？

次の質問を参照してください。

3.3 なぜ浮動小数点数の計算はこんなに不正確なんですか？

ユーザーはよく次のような結果に驚きます:

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

そしてこれが Python のバグだと考えます。が、これはバグではありません。この結果に Python はほとんど関与しておらず、むしろ基底のプラットフォームによる浮動小数点数の扱い方が関与しています。

CPython における `float` 型は記憶に C 言語の `double` 型を使います。`float` オブジェクトの値は固定精度 (典型的には 53 bit) の 2 進浮動小数点数として格納され、Python はプロセッサのハードウェアが実装している C 言語上の演算を使います。つまり、浮動小数点数に関して Python は C 言語や Java のような多くの一般的な言語と同じように振る舞います。

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating point. For example, after:

```
>>> x = 1.2
```

`x` に保存された値は 10 進数の 1.2 の (とても高い精度の) 近似値であって、厳密な 1.2 ではありません。一般的なコンピューターでは、実際に格納される値は:

```
1.0011001100110011001100110011001100110011001100110011001100110011 (binary)
```

で、正確には次の値です:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

53bit の典型的な精度は、Python の `float` に 10 進数で 15~16 桁の精度を与えます。

For a fuller explanation, please see the floating-point arithmetic chapter in the Python tutorial.

3.4 なぜ Python の文字列はイミュータブルなんですか？

これにはいくつかの利点があります。

一つはパフォーマンスです。文字列がイミュータブルなら、生成時に領域を割り当てることができるので、必要な記憶域は固定されて、変更されません。これはタプルとリストを区別する理由の一つでもあります。

他の利点は、Python の文字列は数と同じくらい "基本的" なものと考えられることです。8 という値を他の何かに変える手段が無いように、文字列 "eight" を他の何かに変える手段も無いのです。

3.5 なぜメソッドの定義や呼び出しにおいて 'self' を明示しなければならないのですか？

このアイデアは Modula-3 から取り入れられました。これは様々な理由からとても便利だと言えます。

まず、ローカル変数ではなく、メソッドやインスタンス属性を扱っていることがより明確になります。`self.x` や `self.meth()` と書いてあれば、そのクラスの定義を憶えていなくても、それがインスタンス変数やメソッドであることは明らかです。C++ では、(グローバルが滅多になかったり、簡単に見分けがつかずなら) ローカル変数宣言がないことからある程度わかるでしょう。-- しかし Python にはローカル変数宣言がないので、クラス定義を調べて確かめなくてはなりません。C++ や Java のコーディングスタンダードに、インスタンス属性に `m_` 接頭辞をつけるものがあるので、この明示性はそれらの言語においても有用です。

第二に、特定のクラスからメソッドを明示的に参照または呼び出したい時に、特別な構文が必要なくなります。C++ では、派生クラスでオーバーライドされた基底クラスからメソッドを使うには、`::` 演算子を使わなければなりません。-- Python では、`baseclass.methodname(self, <argument list>)` と書けます。これは特に、`__init__()` メソッドに便利です。派生クラスのメソッドが、基底クラスにある同じ名前のメソッドを拡張するために、基底クラスのメソッドをどうにかして呼び出したい時にも便利です。

最後に、インスタンス変数に対する、代入の構文の問題を解決できます。Python のローカル変数は、関数の中で (global が明示的に宣言されることなく) 値が代入された変数 (と定義されています!) です。なので、ある代入が意図するのが、ローカル変数へではなくインスタンス変数への代入であると、インタプリタが判断する手段が必要です。そしてそれは構文を見るだけで分かる方が (効率が) 良いのです。C++ ではその区別を宣言時に行いますが、Python では宣言がないので、この方法でしか区別できなかつたら残念です。`self.var` を明示すればうまく解決できます。同様に、インスタンス変数を使うのにも `self.var` と書かなければならないので、メソッドの中の `self` が付いていない名前への参照は、そのインスタンスのディレクトリを検索するまでもなくローカル変数とわかります。別の言い方をすれば、ローカル変数とインスタンス変数は二つの異なる名前空間に存在し、Python にどちらの名前空間を使うかを伝えなくてはならないのです。

3.6 式中で代入ができないのはなぜですか？

Python 3.8 以降ならできるよ!

セイウチ演算子 `:=` を使った代入式は、式の中で変数に代入します:

```
while chunk := fp.read(200):
    print(chunk)
```

より詳しくは [PEP 572](#) を参照してください。

3.7 Python にメソッドを使う機能 (`list.index()` 等) と関数を使う機能 (`len(list)` 等) があるのはなぜですか？

Guido いわく:

(a) 幾つかの演算では、接頭辞は接尾辞よりも単純に読みやすいからです。接頭辞 (そして接中辞!) による演算は数学において長い歴史があり、そこでは課題に対する数学者の思考を視覚的に助けるよ

うな記法が好まれます。x*(a+b) を x*a + x*b に書き換える容易さと、それと同じことを純粋なオブジェクト指向の記法で行う煩わしさを比較してみてください。

(b) len(x) というコードを読んだ時、私はそれが何かの長さを問うているのだなと知ることができません。これは私に 2 つの事を知らせています。一つは結果が整数であること、そして引数は何らかのコンテナであることです。対して、x.len() を目にした場合、私はその時点で x が何らかのコンテナであり、それが標準の len() を持っているクラスを継承しているか、インターフェースを実装していることを知っている必要があります。mapping を実装していないクラスが get() や keys() メソッドを持っていたり、file でない何か write() メソッドを持っているような混乱は時折見かけます。

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

3.8 join() がリストやタプルのメソッドではなく文字列のメソッドなのはなぜですか？

文字列は Python 1.6 から他の標準型に大きく近づきました。それ以前は常に string モジュールの関数を使ってできていたことと同等の機能を持つメソッドがこの時に追加されました。その新しいメソッドの多くは広く受け入れられましたが、一部のプログラマに不快を感じさせていると思われるものがこれで:

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

結果はこうなります:

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

この使い方には二つの議論があります。

一つ目は、「文字列リテラル (文字列定数) のメソッドを使うのは醜すぎる」というようなものです。確かにそうかも知れませんが、文字列リテラルは単なる固定された値に過ぎないというのが答えです。文字列に束縛された名前にメソッドが許されるなら、リテラルに使えないようにする論理的な理由はないでしょう。

二つ目の反対理由は、典型的には「私は実際、要素を文字列定数とともに結合させるよう、シーケンスに命じているのだ」というものです。残念ながら、そうではないのです。いくつかの理由から split() を文字列のメソッドとしておいた方がはるかに簡単です。これを見ると分かりやすいでしょう

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

これは文字列リテラルに対する、与えられたセパレータ (または、デフォルトでは任意の空白文字の連続) で区切られた部分文字列を返せという指示です。

join() は、セパレータ文字列に、文字列のシーケンスをイテレートして隣り合う要素の間に自身を挿入するように指示しているので、文字列のメソッドです。このメソッドは、独自に定義された新しいクラスを含め、シーケンスの規則を満たすいかなる引数にも使えます。バイト列やバイト配列にも同様のメソッドがあります。

3.9 例外はどれくらい速いのですか？

`try/except` ブロックは例外が送出されなければ極端に効率的です。実際に例外を捕捉するのは高価です。Python 2.0 より前のバージョンでは、このイディオムを使うのが一般的でした:

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

これは、辞書がほとんどの場合にキーを持っていると予想できるときにのみ意味をなします。そうでなければ、このように書きます:

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

この特殊な場合では `value = dict.setdefault(key, getvalue(key))` も使えますが、これは `getvalue()` 呼び出しが十分安価な場合に限りです。なぜならそれが全ての場合に評価されるからです。

3.10 Python に switch や case 文がないのはなぜですか？

一般的に、構造化された switch 文は、式が特定の値または値の集合を持つとき、1つのコードブロックを実行します。Python 3.10 以降では、リテラル値や名前空間内の定数を、`match ... case` 文で簡単にマッチさせることができます。より古い手段は一連の `if... elif... elif... else` です。

非常に大きな数の選択肢から選ぶとき、値を呼び出す関数に対応づける辞書を作れます。例えば:

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

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

オブジェクトのメソッドを呼び出すには、さらに単純に `getattr()` 組み込み関数で特定の名前のメソッドを検索できます:

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
```

(次のページに続く)

(前のページからの続き)

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

メソッドの名前にこの例の `visit_` のような接頭辞を使うことを勧めます。このような接頭辞がないと、信頼できないソースから値が与えられたときに、オブジェクトの任意のメソッドを呼び出す攻撃をされる可能性があります。

C の switch-case-default のような、フォールスルーのある switch を模倣することもできますが、はるかに難しいうえに必要性も少ないでしょう。

3.11 OS 特有のスレッド実装に依らずにインタプリタでスレッドをエミュレートすることはできないのですか？

答 1: 残念なことに、インタプリタは Python のスタックフレームごとに少なくとも一つの C のスタックフレームを push します。同様に、拡張もほとんどランダムなときに Python にコールバックすることがあります。よって、完全なスレッド実装には C のスレッドサポートが必要です。

答 2: 幸運なことに、[Stackless Python](#) があります。これは完全に再デザインされたインタープリタで、C のスタックを回避しています。

3.12 なぜラムダ式は文を含むことができないのですか？

Python のラムダ式が文を含むことができないのは、Python の文法的な枠組みが式の中にネストされた文を扱うことができないからです。しかし、Python では、これは深刻な問題ではありません。他の言語のラムダに機能が追加されているのとは違い、Python のラムダは単なる、関数を定義するのが面倒すぎる場合のための簡略な記法に過ぎないのです。

関数は既に Python の第一級オブジェクトで、ローカルスコープ内で宣言できます。従って、ローカルで定義された関数ではなくラムダを使う利点は、関数の名前を考える必要が無いことだけです -- しかし、(ラムダ式が生み出すオブジェクトと厳密に同じ型の) 関数オブジェクトが代入される先はただのローカル変数です！

3.13 Python は C やその他の言語のように機械語にコンパイルできますか？

[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.

3.14 Python はメモリをどのように管理するのですか？

Python のメモリ管理の詳細は実装に依ります。Python の標準の C 実装 [CPython](#) は参照カウントを使って、アクセスできないオブジェクトを探します。また別のメカニズムを使って参照サイクルを集めます。これはサイクル検出アルゴリズムを定期的に実行し、アクセスできないサイクルを探し、巻き込まれたオブジェクトを削除します。gc モジュールの関数で、ガベージコレクションを実行し、デバッグ統計を取得し、コレクタ

のパラメタを変更できます。

Other implementations (such as [Jython](#) or [PyPy](#)), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

Python の実装によっては、以下の (CPython では通る) コードはおそらく、ファイルディスクリプタを使い果たすでしょう:

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

実際、CPython の参照カウントとデストラクタのスキームを使えば `f` への新しい代入ごとにファイルは閉じられます。しかし、伝統的な GC を使うと、これらのファイルオブジェクトが回収され (て閉じられる) までに不定な、場合によっては長い、間隔が空くことがあります。

Python の実装に依らずに動くコードを書くには、ファイルを明示的に閉じるか、`with` 文を使ってください。これでメモリ管理のスキームに関係なく動きます:

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

3.15 CPython はなぜ伝統的なガベージコレクションスキームを使わないのですか？

まず、それは C の標準的な機能ではないのでポータブルではありません。(確かに Boehm GC ライブラリはあります。しかし、これにはアセンブリコードが含まれ、**ほとんどの** 有名なプラットフォームに対応していますが全てではありません。また、ほとんど透過的ですが、完全に透過的ではありません。Python を対応させるにはパッチが必要です。)

伝統的な GC は Python が他のアプリケーションに組み込まれるときにも問題となります。スタンドアロンの Python で動く限りでは、標準の `malloc()` と `free()` を GC ライブラリから提供されるものに置き換えても問題ありませんが、Python を実装したアプリケーションは Python のものではない **独自の** 代替品を使おうとするかもしれません。現在のようにすることで、CPython は `malloc()` と `free()` が適切に実装されている限りどんなものにも対応させられます。

3.16 なぜ CPython の終了時にすべてのメモリが解放されるわけではないのですか？

Python モジュールのグローバルな名前空間から参照されるオブジェクトは、Python の終了時にメモリの割り当てを解除されるとは限りません。これは、循環参照があるときに起こりえます。解放できない C ライブラリ (例えば、Purify のようなツールなどが当てはまります) によって割り当てられたいくらかのメモリも含まれます。しかし、Python は終了時にメモリをクリーンアップすることには積極的で、全ての単一のオブジェ

クトを破棄しようとしします。

再割り当て時に Python が特定のものを削除するように強制したいときは、`atexit` モジュールを使って削除を強制する関数を実行してください。

3.17 なぜタプルとリストという別のデータ型が用意されているのですか？

リストとタプルは、多くの点で似ていますが、一般には本質的に異なる方法で使われます。タプルは、Pascal の **レコード** や C の **構造体** と同様なものと考えられます。型が異なっても良い関連するデータの小さな集合で、グループとして演算されます。例えば、デカルト座標は 2 つや 3 つの数のタプルとして適切に表されます。

一方、リストは、もっと他の言語の配列に近いものです。全て同じ型の可変数のオブジェクトを持ち、それらが一つ一つ演算される傾向にあります。例えば、`os.listdir('.')` はカレントディレクトリ内にあるファイルの文字列表現のリストを返します。この出力を演算する関数は一般に、ディレクトリに一つや二つの別のファイルを加えても壊れません。

タプルはイミュータブルなので、一度タプルが生成されたら、そのどの要素も新しい値に置き換えられません。リストはミュータブルなので、リストの要素はいつでも変更できます。イミュータブルな要素だけが辞書のキーとして使えるので、リストではなくタプルだけがキーとして使えます。

3.18 CPython でリストはどのように実装されているのですか？

CPython のリストは実際に変数分の長さの配列で、Lisp スタイルの連結リストではありません。この実装は他のオブジェクトへの参照の連続した配列を使用していて、この配列へのポインタおよび配列長はリストの先頭の構造体に保存されています。

これにより、リストのインデクシング `a[i]` は、リストの大きさやインデクスの値に依存しないコストで演算できます。

要素が追加または挿入されるとき、この参照の配列は大きさが変更されます。要素追加の繰り返しのパフォーマンスを上げるために、少し工夫されています。配列が大きくなるとき、次の何回かは実際に大きさを変更する必要がないように、いくつかの追加の領域が割り当てられます。

3.19 CPython で辞書はどのように実装されていますか？

CPython の辞書は大きさを変更できるハッシュテーブルとして実装されています。B 木と比べて、ほとんどの条件下で (特に一般的な演算である) 探索のパフォーマンスが良いですし、実装も単純です。

辞書は、辞書に保存されているそれぞれのキーに対応するハッシュコードを `hash()` ビルトイン関数で計算することで機能します。このハッシュコードはキーやプロセスごとのシードによって大きく変化します。例えば、`'Python'` のハッシュ値は `-539294296` ですが、ビットが一つ違うだけの文字列 `'python'` のハッシュ値は `1142331976` です。そしてこのハッシュコードは、値が保存される内部配列での位置を計算するために使われます。保存しているキーのハッシュ値が全て異なるとすれば、一定の時間 - Big-O 記法では $O(1)$ - でキーを検索できることになります。

3.20 なぜ辞書のキーはイミュータブルでなくてはならないのですか？

辞書のハッシュテーブルの実装は、キーを見つけるために、キー値から計算されたハッシュ値を使います。もしキーがミュータブルなオブジェクトだったら、その値は変えられ、それによりハッシュ値も変わってしまいます。しかし、キーオブジェクトを変更したのが何者であれ、値が辞書のキーとして使われていたと気付けないので、辞書の中のエントリを適切な場所に動かさせません。そして、同じオブジェクトを探そうとしても、ハッシュ値が違うため見つかりません。古い値を探そうとしても、そのハッシュバイナリから見つかるオブジェクトの値は異なるでしょうから、これも見つかりません。

リストでインデクシングされた辞書が必要ななら、まず単純にリストをタプルに変換してください。関数 `tuple(L)` は、リスト `L` と同じエントリのタプルを生成します。タプルはイミュータブルなので、辞書のキーとして使えます。

いくつかの受け入れられなかった提案:

- アドレス (オブジェクト ID) のハッシュリスト。これは、同じ値の新しいリストを作っても見つからないので駄目です。例えば:

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

`[1, 2]` の 2 行目の `id` は 1 行目のそれと異なってしまうために `KeyError` 例外を送出するでしょう。言い換えれば、辞書のキーは `==` を使って比較されるべきであり、`is` ではないということです。

- リストをキーとして使うときにコピーを作る。リストはミュータブルなので、自分自身への参照を含むことができ、コードをコピーするときに無限ループにハマる可能性があるので、これは駄目です。
- リストをキーとして使うことを認めるが、ユーザにそれを変更させないように伝える。もしユーザが忘れたり、偶然にリストが変更されてしまったりしたら、追跡困難なバグの可能性を生じてしまいます。またこれは、`d.keys()` のすべての値は辞書のキーとして使えるという、辞書の重要な不変性も潰してしまいます。
- リストが一旦辞書のキーとして使われたら、読み出し専用のマークを付ける。問題は、値を変えられるのはトップレベルオブジェクトだけではないことです。リストを含むタプルもキーとして使えます。全てを辞書のキーとして導入すると、そこから到達可能な全てのオブジェクトに読み出し専用のマークを付ける必要があります -- そして再び、自己参照オブジェクトが無限ループを引き起こします。

必要ならばこれを回避する方法がありますが、自己責任のもとで行ってください。ミュータブルな構造を、`__eq__()` と `__hash__()` メソッドの両方を持つクラスインスタンスに含めることができます。その時、辞書 (またはハッシュに基づく別の構造体) に属するような全てのラッパーオブジェクトのハッシュ値が、そのオブジェクトが辞書 (その他の構造体) 中にある間固定され続けることを確実にしてください。

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

    def __eq__(self, other):
```

(次のページに続く)

(前のページからの続き)

```

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

```

なお、リストのメンバーの中にハッシュ化できないものがある可能性や、算術オーバーフローの可能性から、ハッシュ計算は複雑になります。

さらに、そのオブジェクトが辞書に含まれるか否かにかかわらず、`o1 == o2` (すなわち `o1.__eq__(o2)` is True) ならばいつでも `hash(o1) == hash(o2)` (すなわち `o1.__hash__() == o2.__hash__()`) でなくてはなりません。その制限に適合できなければ、辞書やその他のハッシュに基づく構造体は間違いを起こします。

この `ListWrapper` の例では、異常を避けるため、ラップオブジェクトが辞書内にある限りラップされたリストが変更されてはなりません。この条件と満たせなかった時の結果について知恵を絞る覚悟がない限り、これをしてはいけません。よく考えてください。

3.21 なぜ `list.sort()` はソートされたリストを返さないのですか？

パフォーマンスが問題となる状況では、ソートするためだけにリストのコピーを作るのは無駄が多いです。そこで、`list.sort()` はインプレースにリストをソートします。このことを忘れないため、この関数はソートされたリストを返しません。こうすることで、ソートされたコピーが必要で、ソートされていないものも残しておきたいときに、うっかり上書きしてしまうようなことがなくなります。

新しいリストを返したいなら、代わりに組み込みの `sorted()` 関数を使ってください。この関数は、与えられたイテレータ可能オブジェクトから新しいリストを生成し、ソートして返します。例えば、辞書のキーをソートされた順序でイテレートする方法は:

```

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

```

3.22 Python ではどのようにインターフェース仕様を特定し適用するのですか？

C++ や Java のような言語が提供するような、モジュールに対するインターフェース仕様の特定は、モジュールのメソッドや関数の原型を表現します。インターフェースの特定がコンパイル時に適用されることが、大きなプログラムの構成に役立つと、広く感じられています。

Python 2.6 で、抽象基底クラス (Abstract Base Class, ABC) が定義できるようになる `abc` モジュールが追加されました。なので `isinstance()` と `issubclass()` を使って、インスタンスやクラスが、ある ABC を実装しているかどうかチェックできます。`collections.abc` モジュールでは、`Iterable`、`Container`、`MutableMapping` などの便利な ABC が定義されています。

Python では、インターフェース仕様の多くの利点は、コンポーネントへの適切なテスト規律により得られます。

モジュールのための適切なテストスイートは、回帰テストを提供し、モジュールのインターフェース仕様や用例集としても役立ちます。多くの Python モジュールは、簡単な「自己テスト」を提供するスクリプトとして実行できます。複雑な外部インターフェースを使うモジュールさえ、外部インターフェースの細かい「スタブ」エミュレーションで単独にテストできることが多いです。`doctest` や `unittest` モジュール、あるいはサードパーティのテストフレームワークで、モジュールのコードの全ての行に及ぶ徹底的なテストスイートを構成できます。

Python で大きくて複雑なアプリケーションを構築するとき、インターフェース仕様と同様に、適切なテスト規律も役立ちます。実際には、インターフェース仕様ではテストできないプログラムの属性もあるので、それ以上にもなります。例えば、`list.append()` メソッドは新しい要素をある内部リストの終わりに加えます。インターフェース仕様ではこの `list.append()` の実装が実際にこれを行うかをテストできませんが、テストスイートならこの機能を簡単に確かめられます。

テストスイートを書くことはとても有用ですし、簡単にテストできるコード設計を心がけると良いでしょう。人気を博している開発手法の一つ、テスト駆動開発は、実際のコードを記述するよりも先に、まずテストスイートの部分を記述するよう求めています。ご心配なく、Python は、あなたがいい加減でもテストケースを全く書かなくても構いません。

3.23 なぜ goto が無いのですか？

1970 年代、人々は気付きました。秩序なき `goto` は、理解するのも手直しするのも困難という厄介な”スパゲッティ”コードに陥りがちであると。高水準言語では、分岐とループの手段があれば `goto` は不要です。(Python だと、分岐には `if` 文及び `or`・`and`・`if/else` 式を使います。ループには `while` 文と `for` 文を使い、ループ内に `continue`・`break` を含むことがあります)

関数の呼び出しをまたいでも動作する ”構造化された `goto`” をまかなうものとして例外を使えます。C、Fortran、その他の言語での `go` あるいは `goto` 構造の適切な用途は全て、例外で同じようなことをすれば便利であると、広く感じられています。例えば:

```
class label(Exception): pass # declare a label

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

例外ではループ内へ跳ぶことはできませんが、どちらにしてもそれは `goto` の乱用と見なされるものです。使うのは控えてください。

3.24 なぜ raw 文字列 (r-strings) はバックスラッシュで終わってはいけないのですか？

正確には、奇数個のバックスラッシュで終わってはいけません。終わりの対になっていないバックスラッシュは、閉じ引用文字をエスケープし、終わっていない文字列を残してしまいます。

raw 文字列は、独自にバックスラッシュの処理をしようとするプロセッサ (主に正規表現エンジン) への入力を生成しやすいように設計されたものです。このようなプロセッサは、終端の対になっていないバックスラッシュを結局エラーとみなすので、raw 文字列はそれを認めません。その代わりに、バックスラッシュでエスケープすることで、引用文字を文字列として渡すことができます。r-string が意図された目的に使われるときに、この規則が役に立つのです。

Windows のパス名を構築するときには、Windows のシステムコールは普通のスラッシュも受け付けることを覚えておいてください:

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

DOS コマンドのパス名を構築するときには、例えばこの中のどれかを試してください:

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

3.25 属性の代入に "with" 文が使えないのはなぜですか？

Python には、ブロックの実行を包む `with` 文があり、ブロックに入るときとブロックから出るときに、コードを呼び出します。以下のような構造を持つ言語があります:

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

Python では、このような構造は曖昧になるでしょう。

Object Pascal、Delphi、C++ のような他の言語では、静的な型を使うので、曖昧な方法でも、どのメンバに代入されているのか分かります。これが静的型付けの要点です -- コンパイラは **いつでも** コンパイル時にすべての変数のスコープを知るのです。

Python は動的な型を使います。実行時にどの属性が参照されるか事前に分かりません。動作中にメンバ属性が追加あるいは除去されるかもしれません。これでは、単純に読むだけではどのアトリビュートが参照されているか分かりません。ローカルなのか、グローバルなのか、メンバ属性なのか？

例えば、以下の不完全なコード片を考えましょう:

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

このコード片では、`a` は `x` というメンバ属性を持っていると仮定されています。しかし、Python ではインタプリタにはこの仮定を伝えられる仕組みはありません。`a` が、例えば整数だったら、どうなってしまおうでしょうか。`x` という名前のグローバル変数があったら、それが `with` ブロックの中で使われるのでしょうか。この通り、Python の動的な特質から、このような選択はとても難しい物になっています。

しかし、`with` やそれに類する言語の機能の一番の利点 (コード量の削減) は、Python では代入により簡単に手に入れられます:

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

こう書いてください:

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

Python では実行時に名前束縛が解決され、後者はその解決が一度で済むため、これには実行速度をあげる副作用もあります。

似た提案として、「先頭のドット」を使うなどして さらにコード量を減らす構文は、明白さを優先をして却下されました (<https://mail.python.org/pipermail/python-ideas/2016-May/040070.html> 参照)。

3.26 なぜジェネレータは `with` 文をサポートしないのですか？

技術的な理由で、ジェネレータは直接コンテキストマネージャとして使ってもうまく動きません。最も一般的なように、ジェネレータが最後まで回りきるイテレータとして使われる場合、クローズ処理は不要です。必要な場合は、`with` 文で `contextlib.closing(generator)` のようにラップします。

3.27 `if/while/def/class` 文にコロンが必要なのはなぜですか？

主に可読性を高めるため (実験的な ABC 言語の結果の一つ) に、コロンが必要です:

```
if a == b
    print(a)
```

と:

```
if a == b:
    print(a)
```

を考えれば、後者のほうが少し読みやすいでしょう。さらに言えば、この FAQ の解答例は次のようになるでしょう。これは、英語の標準的な用法です。

他の小さな理由は、コロンによってエディタがシンタックスハイライトをしやすくなることです。プログラムテキストの手の込んだ解析をしなくても、コロンを探せばいつインデントを増やすべきかを決められます。

3.28 なぜ Python ではリストやタプルの最後にカンマがあっても良いのですか？

Python では、リスト、タプル、辞書の最後の要素の後端にカンマをつけても良いことになっています:

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

これを許すのには、いくつかの理由があります。

リストやタプルや辞書のリテラルが複数行に渡っているときに、前の行にカンマを追加するのを覚えておく必要が無いと、要素を追加するのが楽になります。また、文法エラーを起こすこと無く、行の並べ替えを行うことができます。

間違えてカンマを落としてしまうと、診断しづらいエラーにつながります。例えば:

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

このリストには 4 つの要素があるように見えますが、実際には 3 つしかありません。"fee"、"fiefoo"、"fum" です。常にカンマを付けるようにすれば、この種のエラーが避けられます。

後端にカンマをつけても良いことにすれば、プログラムによるコード生成も簡単になります。

ライブラリと拡張 FAQ

4.1 ライブラリー般の質問

4.1.1 作業 X を行うためのモジュールやアプリケーションを探すにはどうしますか？

ライブラリリファンレス から関係がありそうな標準ライブラリモジュールがあるかどうか調べてください。(標準ライブラリに何があるかが分かるようになると、この段階をスキップすることができます。)

サードパーティのパッケージについては、[Python Package Index](#) を探したり、[Google](#) その他の web サーチエンジンを試してください。”Python”に加えて一つか二つのキーワードで興味のある話題を検索すれば、たいい役に立つものが見つかるでしょう。

4.1.2 math.py (socket.py, regex.py, etc.) のソースファイルはどこにありますか？

モジュールのソースファイルが見付けられない場合は、それは C、C++ かもしれない別のコンパイル言語で実装された、ビルトインもしくは動的に読み込まれるモジュールかもしれません。この場合、ソースは手に入らないかもしれませんし、`mathmodule.c` のようなものが (Python の読み込みパスに無い) C ソースディレクトリのどこかにあるかもしれません。

Python のモジュールには、(少なくとも) 3 種類あります:

- 1) Python で書かれたモジュール (.py)。
- 2) C で書かれ、動的にロードされるモジュール (.dll, .pyd, .so, .sl, etc)。
- 3) C で書かれ、インタプリタにリンクされているモジュール。このリストを得るには、こうタイプしてください:

```
import sys
print(sys.builtin_module_names)
```

4.1.3 Python のスクリプトを Unix で実行可能にするにはどうしますか？

二つの条件があります: スクリプトファイルのモードが実行可能で、最初の行が `#!` で始まり Python インタプリタのパスが続いていなければなりません。

前者は、`chmod +x scriptfile`、場合によっては `chmod 755 scriptfile` を実行すればできます。

後者は、いくつかの方法でできます。最も直接的な方法はこのように

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

のようにファイルの一番最初の行に、プラットフォーム上の Python がインストールされているパス名を書くことです。

スクリプトを Python インタプリタの場所に依存させたくない場合は、**env** プログラムが使えます。Python インタプリタがユーザの PATH のディレクトリにあることを前提とすれば、ほとんど全ての Unix 系 OS では次の書き方をサポートしています:

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

CGI スクリプトでこれをやっては **いけません**。CGI スクリプトの PATH 変数はたいてい最小限のものになっているので、実際のインタプリタの絶対パスを使う必要があります。

ときおり、ユーザ環境に余裕が無く **/usr/bin/env** プログラムが失敗することがあります; もしくは、env プログラム自体が無いことがあります。そのような場合は、次の (Alex Rezinsky による) ハックが試せます:

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

これには、スクリプトの `__doc__` 文字列を定義するというちょっとした欠点があります。しかし、これを付け足せば直せます:

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

4.1.4 Python には curses/termcap パッケージはありますか？

Unix 系 OS において: 標準の Python ソースディストリビューションには **Modules** サブディレクトリに **curses** モジュールが付いてきますが、デフォルトではコンパイルされていません。(これは Windows 用ディストリビューションでは利用できないことに注意してください -- Windows には **curses** モジュールはありません。)

curses モジュールは基本的な **curses** の機能や、色付きの表示、別の文字集合サポート、パッド、マウスサポートなどの **ncurses** や **SYSV curses** の多くの機能をサポートしています。このことは、モジュールが **BSD curses** だけしか持っていない OS とは互換性が無いことを意味しますが、現在メンテナンスされている OS でそういう類のものは無さそうです。

4.1.5 Python には C の `onexit()` に相当するものはありますか？

atexit モジュールは、C の `onexit()` と同じような関数登録を提供します。

4.1.6 シグナルハンドラが動かないのですがなぜですか？

最もありがちな問題は、シグナルハンドラが間違った引数リストで宣言されていることです。これは次のように呼び出されます

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

だから、これは二つの仮引数で宣言されるべきです:

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

4.2 よくある作業

4.2.1 Python のプログラムやコンポーネントをテストするにはどうしますか？

Python には二つのテストフレームワークがついています。doctest モジュールは、モジュールの docstring から使用例を見つけてそれらを実行し、出力を docstring によって与えられた望まれる出力と比較します。

unittest モジュールは、Java や Smalltalk のテストフレームワークを模した装飾されたテストフレームワークです。

テスト作業を簡単にするために、プログラムにおいてモジュール性の良い設計を使うべきです。プログラムでは、ほぼ全ての処理を関数やクラスのメソッドで包むべきです -- こうすることで、プログラムが速くなるという驚くような愉快的効果がときおり得られることがあります (というのも、ローカル変数へのアクセスはグローバルなアクセスよりも速いからです)。さらに言うと、テストを行うのがより難しくなってしまうため、プログラムは可変なグローバル変数に依存するのを避けるべきです。

プログラムの "global main logic" は

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

のように main モジュールの最後に出来る限りシンプルなものを書くのが良いでしょう。

プログラムが整理され、関数やクラスの動作が追やすい状態になったら、その動作を試すテスト関数を書くべきです。一連のテストを自動化するテストスイートは、それぞれのモジュールに関連付けることができます。これは手間が掛かりそうに思えますが、Python は簡素で融通が効くので、驚くほど簡単です。”製品コード (production code)” と並行でテスト関数を書くことで、バグや設計の不備でさえも早い段階で簡単に見付かるようになるので、コーディング作業をより心地良く楽しいものにできます。

プログラムのメインモジュールとして設計されたのではない ”補助モジュール” には、モジュールの自己テストを含めるといいでしょう。

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


複雑な外部インターフェースと作用し合うプログラムでさえ、外部インターフェースが使えない時でも、Python で実装された "fake" インターフェースを使ってテストできます。

4.2.2 Python のドキュメント文字列からドキュメントを生成するにはどうしますか？

pydoc モジュールで Python ソースコード内のドキュメント文字列から HTML を生成できます。純粋に docstring から API ドキュメントを生成するには、他に `epydoc` という選択肢もあります。Sphinx も docstring の内容を含めることができます。

4.2.3 一度に一つの押鍵を取得するにはどうしますか？

Unix 系 OS ではいくつか解決方法があります。curses を使うのが素直なやり方ですが、curses は学ぶには少し大き過ぎるモジュールです。

4.3 スレッド

4.3.1 スレッドを使ったプログラムを書くにはどうしますか？

`_thread` モジュールではなく、必ず `threading` モジュールを使ってください。`threading` モジュールは、`_thread` モジュールで提供される低レベルな基本要素の、便利な抽象化を構成します。

4.3.2 スレッドが一つも実行されていないようです。なぜですか？

メインスレッドが終了するとともに、全てのスレッドは終了されます。メインスレッドは速く働きすぎるので、スレッドには何をする時間も与えられません。

簡単な解決策は、プログラムの終わりに、スレッドが完了するのに十分な時間のスリープを加えることです：

```
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)  # <-----! 
```

しかし、実際は (ほとんどのプラットフォームでは) スレッドは並行して実行されるのではなく、一つずつ実行されるのです！ なぜなら、OS のスレッドスケジューラは、前のスレッドがブロックされるまで新しいスレッドを開始しないからです。

簡単に直すには、関数の実行の最初にちょっとスリープを加えることです：


```
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)
```

`time.sleep()` の良い遅延時間を推測しようとするよりも、セマフォのような仕組みを使う方が良いでしょう。1つのアイディアは `queue` モジュールを使って、キューオブジェクトを作り、各スレッドが完了したときにキューにトークンを追加し、メインスレッドにスレッドと同じ数のトークンをキューから読み出させることです。

4.3.3 たくさんのワーカースレッドに作業を割り振るにはどうしますか？

最も簡単な方法は、新しい `concurrent.futures` モジュール、特に `ThreadPoolExecutor` クラスを使うことです。

もしくは、実行アルゴリズムを上手にコントロールしたい場合は、自身の手でロジックを書くこともできます。`queue` モジュールを使って、ジョブのリストを含むキューを作ってください。`Queue` クラスはオブジェクトのリストを保持し、キューに要素を追加する `.put(obj)` メソッド、それら要素を返す `.get()` メソッドを持っています。このクラスは、それぞれのジョブがきっちり 1 回だけ取り出されることを保証するのに必要なロック処理にも配慮します。

ここにちょっとした例があります:

```
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:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
```

(次のページに続く)

(前のページからの続き)

```
    else:
        print('Worker', threading.current_thread(), end=' ')
        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)
```

実行時には、以下のように出力されます:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

より詳しいことはモジュールのドキュメントを調べてください; Queue クラスは多機能なインターフェースを提供しています。

4.3.4 グローバルな値のどんな種類の変更がスレッドセーフになるのですか？

global interpreter lock (GIL) が内部で使われ、Python VM で一度に一つだけのスレッドが実行されることが保証されています。一般に、Python ではスレッド間の切り替えをバイトコード命令の間でのみ行います。切り替えの周期は、`sys.setswitchinterval()` で設定できます。したがって、それぞれのバイトコード命令、そしてそれぞれの命令が届く全ての C 実装コードは、Python プログラムの観点からは、アトミックです。

このことから、理論上は、正確な勘定のためには PVM バイトコードの実装を理解することが必要です。実際上は、組み込みデータ型 (整数、リスト、辞書、等) の、変数を共有する "アトミックそうな" 演算は、実際にアトミックです。

例えば、以下の演算は全てアトミックです (L、L1、L2 はリスト、D、D1、D2 は辞書、x、y はオブジェクト、i、j は整数です):

```
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()
```

これらは、アトミックではありません:

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

他のオブジェクトを置き換えるような演算は、そのオブジェクトの参照カウントがゼロになったときに `__del__()` メソッドを呼び出すことがあり、これが影響を及ぼすかもしれません。これは特に、辞書やリストの大規模な更新に当てはまります。疑わしければ、`mutex` を使ってください！

4.3.5 グローバルインタプリタロック (Global Interpreter Lock) を取り除くことはできないのですか？

マルチスレッド Python プログラムは事実上一つの CPU しか使えず、(ほとんど) 全ての Python コードが **グローバルインタプリタロック** (GIL) が保持されている間しか作動しなくなるということで、GIL は、Python をハイエンドなマルチプロセッササーバマシン上に配備する上で邪魔であると見なされがちです。

With the approval of **PEP 703** work is now underway to remove the GIL from the CPython implementation of Python. Initially it will be implemented as an optional compiler flag when building the

interpreter, and so separate builds will be available with and without the GIL. Long-term, the hope is to settle on a single build, once the performance implications of removing the GIL are fully understood. Python 3.13 is likely to be the first release containing this work, although it may not be completely functional in this release.

The current work to remove the GIL is based on a [fork of Python 3.9 with the GIL removed](#) by Sam Gross. Prior to that, 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. Adam Olsen did a similar experiment in his [python-safethread](#) project. Unfortunately, both of these earlier experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL. The Python 3.9 fork is the first attempt at removing the GIL with an acceptable performance impact.

The presence of the GIL in current Python releases 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*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

C 拡張をうまく使うことも役立ちます。C 拡張を時間のかかるタスクの実行に使えば、その拡張は実行が C コードで行われている間 GIL を解放でき、その間に他のスレッドで作業が進められます。zlib や hashlib など、すでにこれを行なっている標準ライブラリモジュールもあります。

An alternative approach to reducing the impact of the GIL is to make the GIL a per-interpreter-state lock rather than truly global. This was first implemented in Python 3.12 and is available in the C API. A Python interface to it is expected in Python 3.13. The main limitation to it at the moment is likely to be 3rd party extension modules, since these must be written with multiple interpreters in mind in order to be usable, so many older extension modules will not be usable.

4.4 入力と出力

4.4.1 ファイルを削除するにはどうしますか？ (その他、ファイルに関する質問...)

`os.remove(filename)` または `os.unlink(filename)` を使ってください。ドキュメントは、`os` モジュールを参照してください。この二つの関数は同じものです。`unlink()` は単に、この関数の Unix システムコールの名称です。

ディレクトリを削除するには、`os.rmdir()` を使ってください。作成には `os.mkdir()` を使ってください。`os.makedirs(path)` は `path` の中間のディレクトリの、存在しないものを作成します。`os.removedirs(path)` は中間のディレクトリが空である限り、それらを削除します。ディレクトリツリー全体とその中身全てを削除したいなら、`shutil.rmtree()` を使ってください。

ファイルの名前を変更するには、`os.rename(old_path, new_path)` を使ってください。

ファイルを切り詰めるには、`f = open(filename, "rb+")` を使ってファイルを開き、`f.truncate(offset)` を使ってください; `offset` はデフォルトでは現在のシーク位置です。`os.open()` で開かれたファイル用の `os.ftruncate(fd, offset)` もあります。`fd` はファイルディスクリプタ (小さい整数値) です。

shutil モジュールにも、`copyfile()`、`copytree()`、`rmtree()` 等、ファイルに作用する関数がいくつか含まれます。

4.4.2 ファイルをコピーするにはどうしますか？

shutil モジュールに `copyfile()` 関数があります。注意点として、これは Windows NTFS ボリューム上の `alternate data streams` も macOS HFS+ 上の `resource forks` もコピーしません。ただし現在ではどちらも使われることはほとんどありません。またファイルのパーミッションやメタデータもコピーされませんが、この代わりに `shutil.copy2()` を使うとそのほとんど (すべてではありませんが) が保持されます。

4.4.3 バイナリデータを読み書きするにはどうしますか？

複雑なバイナリデータ形式の読み書きには、`struct` モジュールを使うのが一番です。これでバイナリデータ (通常は数) を含む文字列を取って、Python オブジェクトに変換することができますし、その逆もできます。

例えば、以下のコードはファイルから 2 バイトの整数 2 個と 4 バイトの整数 1 個をビッグエンディアンフォーマットで読み込みます:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

フォーマット中の '`>`' はデータを強制的にビッグエンディアンにします。ファイルから、文字 '`h`' は一つの”整数”(2 バイト) を読み込み、文字 '`l`' は一つの”long 整数”を読み込みます。

より規則的なデータ (例えば、整数や浮動小数点数の中身の型が揃ったリスト) に対しては、`array` モジュールを使うこともできます。

注釈

バイナリデータを読み書きするには、(ここにあるように "`rb`" を `open()` に渡して) ファイルをバイナリモードで開く義務があります。代わりに (デフォルトの) "`r`" を使った場合は、ファイルはテキストモードで開かれ、`f.read()` は `bytes` オブジェクトではなく `str` オブジェクトを返します。

4.4.4 `os.popen()` によって作られたパイプで `os.read()` が使われていないようです。なぜですか？

`os.read()` は、開かれたファイルを表す小さな整数、ファイルディスクリプタを引数に取る、低レベルの関数です。`os.popen()` は、組み込みの `open()` 関数の返回值と同じ型の、高レベルなファイルオブジェクトを作成します。従って、`os.popen()` によって作成されたパイプ `p` から `n` バイト分だけ読み取るには、`p.read(n)` を使う必要があります。

4.4.5 シリアル (RS232) ポートにアクセスするにはどうしますか？

Win32、OSX、Linux、BSD、Jython、IronPython では:

`pyserial`

Unix では、Mitch Chapman による Usenet の投稿を参照してください:

<https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com>

4.4.6 `sys.stdout` (`stdin`, `stderr`) を閉じようとしても実際に閉じられないのはなぜですか？

Python の *file object* は、低水準の C ファイルディスクリプタ上の、抽象の高水準レイヤです。

組み込みの `open()` 関数によって生成されたほとんどのファイルオブジェクトでは、`f.close()` は Python ファイルオブジェクトが Python の視点からは閉じられているものとする印をつけ、その下にある C ファイルディスクリプタを閉じるように手配します。これは、`f` がガベージとなったときにも、`f` のデストラクタで自動的に起こります。

しかし、`stdin`、`stdout`、`stderr` は C で特別な立場が与えられていることから、Python でも同様に特別に扱われます。`sys.stdout.close()` を実行すると、Python レベルのファイルオブジェクトには閉じられているものとする印がつけられますが、C ファイルディスクリプタは **閉じられません**。

下にある C ファイルディスクリプタのうち、この三つのどれかを閉じるには、まず本当に閉じる必要があることを確かめるべきです (例えば、拡張モジュールの I/O を混乱させてしまうかもしれません)。本当に必要ならば、`os.close()` を使ってください:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

または、数の定数としてそれぞれ 0, 1, 2 も使えます。

4.5 ネットワーク/インターネットプログラミング

4.5.1 Python の WWW ツールには何がありますか？

ライブラリリファレンスマニュアルの `internet` と `netdata` という章を参照してください。Python には、サーバサイドとクライアントサイドの web システムを構築するのに便利な多くのモジュールがあります。

利用可能なフレームワークの概要は Paul Boddie によって、<https://wiki.python.org/moin/WebProgramming> でメンテナンスされています。

4.5.2 どのモジュールが HTML の生成の役に立ちますか？

Web Programming についての wiki のページ から役に立つリンクが見付けられます。

4.5.3 Python のスクリプトからメールを送るにはどうしますか？

標準ライブラリモジュール `smtplib` を使ってください。

以下に示すのが、これを使ったごく単純な対話型のメール送信器です。このメソッドは SMTP リスナをサポートするホストならどこでも作動します。

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    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()
```

Unix 限定の代わりの選択肢は `sendmail` を使うことです。sendmail プログラムの場所はシステムによって様々です; あるときは `/usr/lib/sendmail` だったり、あるときは `/usr/sbin/sendmail` だったり。sendmail のマニュアルページが助けになるでしょう。サンプルコードはこうになります:

```
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)
```

4.5.4 ソケットの `connect()` メソッドでブロッキングされなくするにはどうしますか？

主に `select` モジュールがソケットの非同期の I/O を扱うのに使われます。

TCP 接続がブロッキングされないようにするために、ソケットをノンブロッキングモードに設定することが出来ます。そして `connect()` したときに、即座に接続できるか、エラー番号を `.errno` として含む例外を受け取るかのどちらかになります。`errno.EINPROGRESS` は、接続が進行中であるが、まだ完了していないということを示します。異なる OS では異なる値が返されるので、あなたのシステムで何が返されるかを確かめておく必要があります。

`connect_ex()` メソッドを使えば例外を生成しなくて済みます。これは単に `errno` の値を返すでしょう。ポーリングのためには、後でまた `connect_ex()` を呼び出すことができます -- 0 または `errno.EISCONN` は接続されたことを表します -- または、このソケットを `select.select()` に渡して書き込み可能か調べることができます。

注釈

`asyncio` モジュールは、ノンブロッキングのネットワークコードを書く作業のためにフレームワークのような方法を提供します。その他の選択肢として、`Twisted` は多機能で著名なサードパーティのライブラリです。

4.6 データベース

4.6.1 Python にはデータベースパッケージへのインターフェースはありますか？

はい。

標準の Python には、DBM や GDBM などの、ディスクベースのハッシュへのインターフェースも含まれています。また、`sqlite3` モジュールは、軽量なディスクベースの関係データベースを提供します。

ほとんどの関係データベースがサポートされています。詳細は [DatabaseProgramming wiki page](#) を参照してください。

4.6.2 Python で永続的なオブジェクトを実装するにはどうしますか？

`pickle` ライブラリモジュールで、ごく一般的な方法でこれを解決できます (開かれたファイル、ソケット、ウィンドウのようなものを保管することはできませんが)。`shelve` ライブラリモジュールは `pickle` と `(g)dbm` を使い、任意の Python オブジェクトを含む永続的なマッピングを生成します。

4.7 数学と数

4.7.1 Python で乱数を生成するにはどうしますか？

標準モジュールの `random` が乱数生成器を実装しています。使い方は単純です:

```
import random
random.random()
```


This returns a random floating-point number in the range [0, 1).

このモジュールにはその他多くの特化した生成器もあります。例えば:

- `randrange(a, b)` は区間 `[a, b)` から整数を選びます。
- `uniform(a, b)` chooses a floating-point number in the range `[a, b)`.
- `normalvariate(mean, sdev)` は正規 (ガウス) 分布をサンプリングします。

シーケンスに直接作用する高水準な関数もあります。例えば:

- `choice(S)` は与えられたシーケンスからランダムな要素を選びます。
- `shuffle(L)` はリストをインプレースにシャッフルします。すなわち、ランダムに並び替えます。

`Random` クラスのインスタンスを生成して、複数の独立な乱数生成器をつくることもできます。

拡張と埋め込み FAQ

5.1 C で独自の関数を作ることはできますか？

はい。関数、変数、例外、そして新しいタイプまで含んだビルトインモジュールを C で作れます。これはドキュメント `extending-index` で説明されています。

ほとんどの中級から上級の Python 本もこの話題を扱っています。

5.2 C++ で独自の関数を作ることはできますか？

はい。C++ 内にある C 互換機能を使ってできます。`extern "C" { ... }` で Python のインクルードファイルを囲み、`extern "C"` を Python インタプリタから呼ぶ各関数の前に置いてください。グローバルや静的な C++ オブジェクトの構造体を持つものは良くないでしょう。

5.3 C を書くのは大変です。他の方法はありませんか？

独自の C 拡張を書くための別のやり方は、目的によっていくつかあります。

[Cython](#) とその親戚 [Pyrex](#) は、わずかに変形した Python を受け取り、対応する C コードを生成します。Cython や Pyrex を使えば Python の C API を習得することなく拡張を書けます。

現時点で Python 拡張が存在しない C や C++ ライブラリへのインターフェースが必要な場合、[SWIG](#) のようなツールを使ってそのライブラリのデータ型や関数をラッピングできます。[SIP](#)、[CXX](#)、[Boost](#)、[Weave](#) でも C++ ライブラリをラッピングできます。

5.4 C から任意の Python 文を実行するにはどうしますか？

これを行う最高水準の関数は `PyRun_SimpleString()` で、一つの文字列引数を取り、モジュール `__main__` のコンテキストでそれを実行し、成功なら 0、例外 (`SyntaxError` を含む) が発生したら -1 を返します。更に制御したければ、`PyRun_String()` を使ってください。ソースは `Python/pythonrun.c` の `PyRun_SimpleString()` を参照してください。

5.5 C から任意の Python 式を評価するにはどうしますか？

先の質問の `PyRun_String()` を、スタートシンボル `Py_eval_input` を渡して呼び出してください。これは式を解析し、評価してその値を返します。

5.6 Python オブジェクトから C の値を取り出すにはどうしますか？

オブジェクトの型に依ります。タプルなら、`PyTuple_Size()` が長さを返し、`PyTuple_GetItem()` が指定されたインデックスの要素を返します。リストにも同様の関数 `PyList_Size()` と `PyList_GetItem()` があります。

bytes では、`PyBytes_Size()` がその長さを返し、`PyBytes_AsStringAndSize()` がその値へのポインタと長さを提供します。Python の bytes オブジェクトは null を含むこともできるので、C の `strlen()` を使うべきではないことに注意してください。

オブジェクトの型を検査するには、まず最初にそれが NULL でないことを確かめた上で、`PyBytes_Check()`、`PyTuple_Check()`、`PyList_Check()` などを使います。

いわゆる 'abstract' インターフェースから提供される、Python オブジェクトに対する高レベル API もあります -- より詳しいことは `Include/abstract.h` を読んでください。これを使うと、数値型プロトコル (`PyNumber_Index()` など) や `PyMapping` API のマップ型プロトコルなどの役に立つプロトコルに加え、`PySequence_Length()` や `PySequence_GetItem()` などの任意の種類の Python シーケンスへのインターフェースにアクセスできます。

5.7 Py_BuildValue() で任意長のタプルを作るにはどうしますか？

できません。代わりに `PyTuple_Pack()` を使ってください。

5.8 C からオブジェクトのメソッドを呼び出すにはどうしますか？

`PyObject_CallMethod()` 関数でオブジェクトの任意のメソッドを呼び出せます。パラメタは、オブジェクト、呼び出すメソッドの名前、`Py_BuildValue()` で使われるようなフォーマット文字列、そして引数です:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

これはメソッドを持ついかなるオブジェクトにも有効で、組み込みかユーザ定義かは関係ありません。返り値に対して `Py_DECREF()` する必要があることもあります。

例えば、あるファイルオブジェクトの "seek" メソッドを 10, 0 を引数として呼ぶとき (ファイルオブジェクトのポインタを "f" とします):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
}
```

(次のページに続く)

(前のページからの続き)

```

}
else {
    Py_DECREF(res);
}

```

なお、`PyObject_CallObject()` の引数リストには 常に タプルが必要です。関数を引数なしで呼び出すには、フォーマットに `"()`" を渡し、関数を一つの引数で呼び出すには、関数を括弧でくくって例えば `"(i)"` としてください。

5.9 PyErr_Print() (その他 stdout/stderr に印字するもの) からの出力を受け取るにはどうしますか？

Python コード内で、`write()` メソッドをサポートするオブジェクトを定義してください。そのオブジェクトを `sys.stdout` と `sys.stderr` に代入してください。`print_error` を呼び出すか、単に標準のトレースバック機構を作動させてください。そうすれば、出力は `write()` が送る任意の所に行きます。

最も簡単な方法は、`io.StringIO` クラスを使うことです:

```

>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!

```

これと同じことをするカスタムオブジェクトは次のようになります:

```

>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!

```

5.10 C から Python で書かれたモジュールにアクセスするにはどうしますか？

以下のようにモジュールオブジェクトへのポインタを得られます:

```
module = PyImport_ImportModule("<modulename>");
```

そのモジュールがまだインポートされていない (つまり、まだ `sys.modules` に現れていない) なら、これはモジュールを初期化します。そうでなければ、単純に `sys.modules["<modulename>"]` の値を返します。なお、これはモジュールをいかなる名前空間にも代入しません。これはモジュールが初期化されて `'sys.modules'` に保管されていることを保証するだけです。

これで、モジュールの属性 (つまり、モジュールで定義された任意の名前) に以下のようにアクセスできるようになります:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

`PyObject_SetAttrString()` を呼んでモジュールの変数に代入することもできます。

5.11 Python から C++ ヘインターフェースするにはどうしますか？

やりたいことに応じて、いろいろな方法があります。手動でやるなら、“拡張と埋め込み”ドキュメントを読むことから始めてください。なお、Python ランタイムシステムにとっては、C と C++ はあまり変わりません。だから、C 構造体 (ポインタ) 型に基づいて新しい Python の型を構築する方針は C++ オブジェクトに対しても有効です。

C++ ライブラリに関しては、*C* を書くのは大変です。他の方法はありませんか？を参照してください。

5.12 セットアップファイルでモジュールを追加しようとしたらメイクに失敗しました。なぜですか？

セットアップは改行で終わらなければならないと、改行がないと、ビルド工程は失敗します。(これを直すには、ある種の醜いシェルスクリプトハックが必要ですが、このバグは小さいものですから努力に見合う価値はないでしょう。)

5.13 拡張をデバッグするにはどうしますか？

動的にロードされた拡張に GDB を使うとき、拡張がロードされるまでブレークポイントを設定してはいけません。

`.gdbinit` ファイルに (または対話的に)、このコマンドを加えてください:

```
br _PyImport_LoadDynamicModule
```

そして、GDB を起動するときに:

```
$ 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 Linux システムで Python モジュールをコンパイルしたいのですが、見つからないファイルがあります。なぜですか？

Most packaged versions of Python omit some files required for compiling Python extensions.

For Red Hat, install the python3-devel RPM to get the necessary files.

For Debian, run `apt-get install python3-dev`.

5.15 ”不完全 (incomplete) な入力” を ”不適切 (invalid) な入力” から区別するにはどうしますか？

Python インタラクティブインタプリタでは、入力が不完全なとき (例えば、`if` 文の始まりをタイプした時や、カッコや三重文字列引用符を閉じていない時など) には継続プロンプトを与えられますが、入力が不適切であるときには即座に構文エラーメッセージが与えられます。このようなふるまいを模倣したいことがあります。

Python では構文解析器のふるまいに十分に近い `codeop` モジュールが使えます。例えば IDLE がこれを使っています。

これを C で行う最も簡単な方法は、`PyRun_InteractiveLoop()` を (おそらく別のスレッドで) 呼び出し、Python インタプリタにあなたの入力を扱わせることです。独自の入力関数を指定するのに `PyOS_ReadlineFunctionPointer()` を設定することもできます。詳しいヒントは、`Modules/readline.c` や `Parser/myreadline.c` を参照してください。

5.16 未定義の g++ シンボル `__builtin_new` や `__pure_virtual` を見つけるにはどうしますか？

g++ モジュールを動的にロードするには、Python を再コンパイルし、それを g++ で再リンク (Python Modules Makefile 内の `LINKCC` を変更) し、拡張を g++ でリンク (例えば `g++ -shared -o mymodule.so mymodule.o`) しなければなりません。

5.17 メソッドのいくつかは C で、その他は Python で実装されたオブジェクトクラスを (継承などで) 作ることができますか？

はい、`int`、`list`、`dict` などのビルトインクラスから継承できます。

The Boost Python Library (BPL, <https://www.boost.org/libs/python/doc/index.html>) を使えば、これを C++ からできます。(すなわち、BPL を使って C++ で書かれた拡張クラスを継承できます)。

WINDOWS 上の PYTHON FAQ

6.1 Python プログラムを Windows で動かすにはどうしますか？

これは必ずしも単純な質問ではありません。Windows コマンドラインからプログラムを実行するのに慣れている場合は、全て明らかなことに思えるでしょう; そうでない場合は、もう少し手引きが必要でしょう。

ある種の統合開発環境を使っているのであれば、最終的には ” コマンドプロンプト ” などと呼ばれているものに、Windows コマンドをタイプすることになるでしょう。通常そういうウィンドウは検索バーで `cmd` を検索して起動します。通常は次のような見た目の Windows の ” コマンド プロンプト ” が現れるので、ウィンドウが起動したのが分かるはずです:

```
C:\>
```

この文字は異なっていたり、続きがあったりするので、コンピュータの設定や、あなたが最近何をしたかに依って、このようになっていることもあるでしょう:

```
D:\YourName\Projects\Python>
```

このようなウィンドウさえ開けば、Python プログラムを動かす手順は順調に進みます。

Python スクリプトは Python **インタプリタ** と呼ばれる別のプログラムで処理されなければならないということを、理解する必要があります。インタプリタはスクリプトを読み込み、バイトコードにコンパイルし、バイトコードを実行しプログラムを走らせます。では、インタプリタが Python スクリプトを取り扱うためには、どんな用意をするのでしょうか?

まず最初に、コマンドウィンドウに ”`py`” という単語がインタプリタを起動する指示であると認識させる必要があります。コマンドウィンドウを開いたら、`py` というコマンドを入力し、リターンキーを叩いてください。:

```
C:\Users\YourName> py
```

次のような出力が見えるでしょう:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on
↳win32
```

(次のページに続く)

(前のページからの続き)

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

インタプリタの”対話モード”が始まりました。これで Python の文や式を対話的に入力して、待っている間に実行や評価させることができます。これが Python の最強の機能のひとつです。いくつかの式を選んで入力し、結果を見て確かめてみましょう:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

多くの人が対話モードを便利で高度なプログラム可能計算機として使っています。対話式 Python セッションを終わらせたいなら、`exit()` 関数を呼び出すか、`Ctrl` キーを押しながら `Z` を入力し ”Enter” キーを押して、Windows コマンドプロンプトに戻ってください。

スタート すべてのプログラム *Python 3.x Python (command line)* のような項目がスタートメニューに見付かるかもしれません。これを起動すると、新しいウィンドウに `>>>` というプロンプトが見られるでしょう。その場合、`exit()` 関数を呼び出すか `Ctrl-Z` を入力するとウィンドウは閉じます; Windows は 1 つの ”python” コマンドをウィンドウで実行していて、インタプリタを終了させたときに、そのウィンドウを閉じます。

`py` コマンドが認識されていることが確認できたので、Python スクリプトを与えられるようになりました。そのためには Python スクリプトの絶対パスあるいは相対パスを与える必要があります。Python スクリプトがデスクトップにあり、`hello.py` という名前で、ちょうど良いことにコマンドプロンプトがホームディレクトリを開いていると仮定しましょう。そうすると次のような表示が見られます:

```
C:\Users\YourName>
```

ここまで来れば `py` の後ろにスクリプトのパスを入力することで、Python にスクリプトを渡すよう `py` コマンドに命令できます:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 Python スクリプトを実行可能にするにはどうしますか？

Windows では、標準の Python インストーラはすでに `.py` 拡張子のあるファイル型 (Python.File) に関連付け、そのファイル型にインタプリタを実行するオープンコマンド (`D:\Program Files\Python\python.exe "%1" %*`) を与えます。コマンドプロンプトから `'foo.py'` としてスクリプトを実行可能にするにはこれで十分です。スクリプトを拡張子なしで `'foo'` とだけタイプして実行したいのなら、`PATHEXT` 環境変数に `.py` を加えてください。

6.3 Python の起動に時間がかかることがあるのはなぜですか？

通常 Python は Windows でとても早く起動しますが、ときどき Python が急にスタートアップに時間がかかるようになったというバグレポートがあります。更に複雑なことに、Python は同様に設定された他の Windows システムではきちんと動くのです。

この問題はそのマシンのウイルス対策ソフトウェアの設定ミスによって起こされることがあります。ウイルススキャナの中には、ファイルシステムからの全ての読み込みを監視するように設定した場合に、二桁のスタートアップオーバーヘッドを引き起すことが知られているものがあります。あなたのシステムのウイルススキャンソフトウェアの設定を確かめて、本当に同様に設定されていることを確実にしてください。

6.4 どうすれば Python スクリプトを EXE に出来ますか？

実行可能ファイルを作るツールの一覧 [どうしたら Python スクリプトからスタンドアロンバイナリを作れますか？](#) を参照してください。

6.5 *.pyd ファイルは DLL と同じですか？

はい、.pyd ファイルは dll と同じようなものですが、少し違いがあります。foo.pyd という名前の DLL があったとしたら、それには関数 PyInit_foo() が含まれていなければなりません。そうすれば Python で "import foo" を書いて、Python は foo.pyd (や foo.py, foo.pyc) を探して、あれば、PyInit_foo() を呼び出して初期化しようとします。Windows が DLL の存在を必要とするのと違い、.exe ファイルを foo.lib にリンクするわけではありません。

なお、foo.pyd を検索するパスは PYTHONPATH であり、Windows が foo.dll を検索するパスと同じではありません。また、プログラムを dll にリンクしたときはプログラムの実行に dll が必要ですが、foo.pyd は実行には必要はありません。もちろん、import foo したいなら foo.pyd は必要です。DLL では、リンクはソースコード内で __declspec(dllexport) によって宣言されます。.pyd では、リンクは使える関数のリストで定義されます。

6.6 Python を Windows アプリケーションに埋め込むにはどうしたらいいですか？

Python インタプリタを Windows app に埋め込む方法は、次のように要約できます:

1. あなたの .exe 内に直接 Python を取り込んでビルド **しないで** 下さい。Windows では、それ自身が DLL になっているようなモジュールのインポートを処理するために、Python は DLL でなければなりません。(ドキュメント化されていない重大な事実の一つ目です。) 代わりに、pythonNN.dll とリンクしてください; 普通はそれは C:\Windows\System にインストールされています。NN は Python バージョンで Python 3.3 であれば "33" のような数字です。

Python には、load-time に、または run-time にリンクできます。load-time なリンクは、pythonNN.lib に対してリンクするもので、run-time なリンクは pythonNN.dll に対してリンクするものです。(一般的な注意: pythonNN.lib は pythonNN.dll に対するいわゆる "インポートライブラリ" です。これは単にリンクに対するシンボルを定義します。)

run-time なリンクは、リンクの選択を大いに単純化します。全ては実行時に行われます。コードは

Windows の `LoadLibraryEx()` ルーチンで `pythonNN.dll` をロードしなければなりません。コードはまた、Windows の `GetProcAddress()` ルーチンで得られるポインタで、`pythonNN.dll` (すなわち、Python の C API) のルーチンとデータへアクセスしていなければなりません。マクロによって、このポインタを Python の C API のルーチンを呼び出す任意の C コードに通して使えます。

2. SWIG を使えば、app のデータとメソッドを Python で使えるようにする Python ” 拡張モジュール” を簡単に作れます。SWIG は雑用を殆どやってくれるでしょう。結果として、.exe ファイル **の中に** リンクする C コードができます (!)。DLL を作 **らなくてもよく**、リンクも簡潔になります。
3. SWIG は拡張の名前に依る名前の `init` 関数 (C 関数) を作ります。例えば、モジュールの名前が `leo` なら、`init` 関数の名前は `initleo()` になります。SWIG shadow クラスを使ったほうがよく、そうすると `init` 関数の名前は `initleoc()` になります。これは shadow クラスが使うほとんど隠れた helper クラスを初期化します。

ステップ 2 の C コードを .exe ファイルにリンクできるのは、初期化関数の呼び出しと Python へのモジュールのインポートが同等だからです ! (ドキュメント化されていない重大な事実の二つ目です)

4. 要するに、以下のコードを使って Python インタプリタを拡張モジュール込みで初期化することができます。

```
#include <Python.h>
...
Py_Initialize(); // Initialize Python.
initmyAppc(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

5. Python の C API には、`pythonNN.dll` をビルドするのに使われたコンパイラ MSVC 以外のコンパイラを使うと現れる二つの問題があります。

問題 1: コンパイラによって `struct FILE` に対する概念が異なるため、`FILE *` 引数を取るいわゆる ” 超高水準 ” 関数は、多コンパイラ環境で動きません。実装の観点から、これらは超低水準関数になっています。

問題 2: SWIG は `void` 関数へのラッパを生成するときに以下のコードを生成します:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

ああ、`Py_none` は `pythonNN.dll` 内の `_Py_NoneStruct` という複雑なデータ構造に展開するマクロです。また、このコードは他コンパイラ環境では失敗します。このコードを次のように置き換えてください:

```
return Py_BuildValue("");
```

これで、SWIG をまだ仕事に使えない (私は SWIG の完全な初心者です) 私でも、SWIG の `%typemap` コマンドを使って自動的に変更できるようになります。

6. Python シェルスクリプトを使って Windows app 内から Python インタプリタウィンドウを掲示するのはいい方法ではありません。そのように表示されるウィンドウは app のウィンドウシステムとは関係ありません。むしろ "ネイティブな" インタプリタウィンドウを (wxPythonWindow を使ったりして) 作るべきです。そのウィンドウを Python インタプリタにつなぐのは簡単です。Python の i/o は読み書きをサポートする `__どんな__` オブジェクトにもリダイレクトできるので、`read()` と `write()` メソッドを含む (拡張モジュールで定義された) Python オブジェクトさえあればいいのです。

6.7 エディタが Python ソースにタブを勝手に挿入しないようにするにはどうしますか？

この FAQ ではタブを使うことを勧めません。Python スタイルガイド [PEP 8](#) では、配布される Python コードにはスペース 4 つを使うことを推奨しています。これは Emacs の python-mode のデフォルトでも同じです。

いかなるエディタでも、タブとスペースを混ぜるのは良くないです。MSVC も全く同じ立場であり、スペースを使うようにする設定が簡単にできます。*Tools Options Tabs* を選択し、ファイルタイプの "デフォルト" の "タブ幅" と "インデント幅" に 4 を設定して、"スペースを挿入する" のラジオボタンを選択してください。

Python は、もしタブとスペースが混在していることで先頭の空白に問題がある場合、`IndentationError` または `TabError` を送出します。`tabnanny` モジュールを実行することで、ディレクトリツリーをバッチモードでチェックすることができます。

6.8 ブロックすることなく押鍵を検出するにはどうしますか？

`msvcrt` モジュールを使ってください。これは標準の Windows 専用拡張モジュールです。これはキーボードが打たれているかを調べる関数 `kbhit()` と、反響することなく一文字を得る `getch()` を定義します。

6.9 missing api-ms-win-crt-runtime-l1-1-0.dll エラーを解決するにはどうしますか？

この問題は更新がインストールされていない Windows 8.1 以前を使っていると Python 3.5 以降で発生することがあります。まずはあなたのオペレーティングシステムがサポートされており最新であることを確認してください。それでも解決しない場合は、C ランタイムアップデートの手動インストール方法について [Microsoft support page](#) を参照してください。

グラフィックユーザインターフェース FAQ

7.1 一般的な GUI の質問

7.2 Python の GUI ツールキットには何がありますか？

Python の標準的なビルドには、tkinter という Tcl/Tk ウィジェットセットのオブジェクト指向インターフェースが含まれています。これは最も簡単にインストールして使えるでしょう (なぜなら、これは Python のほとんどの バイナリディストリビューション に同梱されているからです)。ソースへのポインタなど、Tk に関する詳しい情報は、[Tcl/Tk ホームページ](#) を参照してください。Tcl/Tk は、macOS、Windows、Unix プラットフォームに完全にポータブルです。

対象とするプラットフォームによっては他の選択肢もあります。[プラットフォーム互換](#) および [プラットフォーム固有](#) の GUI フレームワークのリストを [python wiki](#) で参照できます。

7.3 Tkinter の質問

7.3.1 Tkinter アプリケーションを凍結するにはどうしますか？

Freeze はスタンドアロンアプリケーションを生成するツールです。Tkinter アプリケーションを凍結するとき、それは Tcl と Tk ライブラリを必要とするので、真のスタンドアロンにはなりません。

一つの解決策は、アプリケーションに Tcl と Tk ライブラリを同梱し、環境変数 `TCL_LIBRARY` と `TK_LIBRARY` でランタイムに指定することです。

Various third-party freeze libraries such as py2exe and cx_Freeze have handling for Tkinter applications built-in.

7.3.2 I/O を待つ間に扱われる Tk イベントを作れますか？

Windows 以外のプラットフォームについては、はい、スレッドさえ必要ありません！ただし、I/O コードを少し再構成しなければなりません。Tk には `Xt` の `XtAddInput()` コールと同等なものがあるので、ファイルディスクリプタ上で I/O が可能なときに Tk メインループから呼ばれるコールバック関数を登録できます。[tkinter-file-handlers](#) を参照してください。

7.3.3 Tkinter で働くキーバインディングが得られません。なぜですか？

`bind()` メソッドでイベントに 結び付けられた イベントハンドラが、適切なキーが押されたときにさえハンドルされないという苦情がよく聞かれます。

最も一般的な原因は、バインディングが適用されるウィジェットが ” キーボードフォーカス ” を持たないことです。Tk ドキュメントでフォーカスコマンドを確認してください。通常はウィジェットの中をクリックすることでキーボードフォーカスを与えられます (ただしラベルには与えられません。takefocus オプションを参照してください)。

”なぜ PYTHON が私のコンピュータにインストールされているのですか？” FAQ

8.1 Python とは何ですか？

Python はプログラミング言語の一つです。Python は様々なアプリケーションに使われています。Python は習得しやすいので、一部の高校や大学で入門用のプログラミング言語として使われています。一方で、Google、NASA、Lucasfilm Ltd. のような組織でプロフェッショナルなソフトウェア開発者にも使われています。

Python についてもっと詳しく知りたいなら、[Beginner's Guide to Python](#) から始めましょう。

8.2 なぜ Python が私のマシンにインストールされているのですか？

インストールした覚えがないのに Python があなたのシステムにインストールされているとしたら、その理由にはいくつかの可能性があります。

- もしかしたら、コンピュータの他のユーザがプログラミングを学ぶためにインストールしたのかもしれませんが。誰がそのマシンを使っていて Python をインストールしたか知る必要があります。
- マシンにインストールされているサードパーティのアプリケーションが、Python で書かれていて Python のインストールを含んでいるかもしれません。そのようなアプリケーションは、GUI プログラムからネットワーク・サーバや管理スクリプトまで、たくさんあります。
- Windows マシンにも Python がインストールされたものがあります。この執筆時点で Hewlett-Packard や Compaq が Python を組み込んでいることを確認しています。HP/Compaq の管理ツールの中には、Python で書かれたものもあるようです。
- macOS や一部の Linux ディストリビューションなど、多くの Unix 互換オペレーティングシステムシステムには、Python がデフォルトでインストールされています。標準インストールに Python が含まれているのです。

8.3 Python を削除してもいいですか？

Python をどこから手に入れたかによります。

誰かが Python を意図してダウンロードしたのであれば、何の問題もなく削除することができます。Windows では、コントロールパネルのプログラムの追加と削除を使ってください。

Python がサードパーティのアプリケーションによってインストールされたなら、同様に削除できますが、そのアプリケーションは動作しなくなります。Python のディレクトリを削除するのではなく、そのアプリケーションのアンインストーラーを使うべきです。

Python がオペレーティングシステムに含まれて来たなら、削除はお勧めできません。削除してしまうと、Python で書かれたプログラムは全く動かなくなり、その中には重要なものが含まれているかもしれません。その場合、修復にはシステム全体の再インストールが必要になるでしょう。

用語集

>>> 対話型 (*interactive*) シェルにおけるデフォルトの Python プロンプトです。インタプリターで対話的に実行されるコード例でよく見られます。

... 次のものが考えられます:

- 対話型 (*interactive*) シェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧、角括弧、波括弧、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。
- 組み込みの定数 `Ellipsis`。

abstract base class

(抽象基底クラス) 抽象基底クラスは *duck-typing* を補完するもので、`hasattr()` などの別のテクニックでは不恰好であったり微妙に誤る (例えば `magic methods` の場合) 場合にインターフェースを定義する方法を提供します。ABC は仮想 (virtual) サブクラスを導入します。これは親クラスから継承しませんが、それでも `isinstance()` や `issubclass()` に認識されます; `abc` モジュールのドキュメントを参照してください。Python には、多くの組み込み ABC が同梱されています。その対象は、(`collections.abc` モジュールで) データ構造、(`numbers` モジュールで) 数、(`io` モジュールで) ストリーム、(`importlib.abc` モジュールで) インポートファインダー及びローダーです。`abc` モジュールを利用して独自の ABC を作成できます。

annotation

(アノテーション) 変数、クラス属性、関数のパラメータや戻り値に関係するラベルです。慣例により *type hint* として使われています。

ローカル変数のアノテーションは実行時にはアクセスできませんが、グローバル変数、クラス属性、関数のアノテーションはそれぞれモジュール、クラス、関数の `__annotations__` 特殊属性に保持されています。

機能の説明がある *variable annotation*, *function annotation*, [PEP 484](#), [PEP 526](#) を参照してください。また、アノテーションを利用するベストプラクティスとして [annotations-howto](#) も参照してください。

引数 (argument)

(実引数) 関数を呼び出す際に、**関数** (または **メソッド**) に渡す値です。実引数には2種類あります:

- **キーワード引数**: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、`3` と `5` がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- **位置引数**: キーワード引数以外の引数。位置引数は引数リストの先頭を書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では `3` と `5` は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については `calls` を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数、FAQ の **実引数と仮引数の違いは何ですか?**、**PEP 362** を参照してください。

asynchronous context manager

(非同期コンテキストマネージャ) `__aenter__()` と `__aexit__()` メソッドを定義することで `async with` 文内の環境を管理するオブジェクトです。**PEP 492** で導入されました。

asynchronous generator

(非同期ジェネレータ) *asynchronous generator iterator* を返す関数です。 `async def` で定義されたコルーチン関数に似ていますが、`yield` 式を持つ点で異なります。 `yield` 式は `async for` ループで利用できる値の並びを生成するのに使用されます。

通常は非同期ジェネレータ関数を指しますが、文脈によっては **非同期ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

非同期ジェネレータ関数には、`async for` 文や `async with` 文だけでなく `await` 式もあります。

asynchronous generator iterator

(非同期ジェネレータイテレータ) *asynchronous generator* 関数で生成されるオブジェクトです。

これは *asynchronous iterator* で、`__anext__()` メソッドを使って呼ばれると `awaitable` オブジェクトを返します。この `awaitable` オブジェクトは、次の `yield` 式まで非同期ジェネレータ関数の本体を実行します。

各 `yield` では一時的に処理を中断し、その場の実行状態 (ローカル変数や保留中の `try` 文を含む) を記憶します。**非同期ジェネレータイテレータ** が `__anext__()` で返された他の `awaitable` で実際に再開する時には、その中断箇所が選ばれます。**PEP 492** および **PEP 525** を参照してください。

asynchronous iterable

(非同期イテラブル) `async for` 文の中で使用できるオブジェクトです。自身の `__aiter__()` メソッドから *asynchronous iterator* を返さなければなりません。PEP 492 で導入されました。

asynchronous iterator

(非同期イテレータ) `__aiter__()` と `__anext__()` メソッドを実装したオブジェクトです。`__anext__()` は *awaitable* オブジェクトを返さなければなりません。`async for` は `StopAsyncIteration` 例外を送出するまで、非同期イテレータの `__anext__()` メソッドが返す *awaitable* を解決します。PEP 492 で導入されました。

属性

(属性) オブジェクトに関連付けられ、ドット表記式によって名前通常参照される値です。例えば、オブジェクト `o` が属性 `a` を持っているとき、その属性は `o.a` で参照されます。

オブジェクトには、`identifiers` で定義される識別子ではない名前の属性を与えることができます。たとえば `setattr()` を使い、オブジェクトがそれを許可している場合に行えます。このような属性はドット表記式ではアクセスできず、代わりに `getattr()` を使って取る必要があります。

awaitable

(待機可能) `await` 式で 사용할 수 있는オブジェクトです。*coroutine* か、`__await__()` メソッドがあるオブジェクトです。PEP 492 を参照してください。

BDFL

慈

悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

binary file

(バイナリファイル) *bytes-like* オブジェクト の読み込みおよび書き込みができる **ファイルオブジェクト** です。バイナリファイルの例は、バイナリモード ('rb', 'wb' or 'rb+') で開かれたファイル、`sys.stdin.buffer`、`sys.stdout.buffer`、`io.BytesIO` や `gzip.GzipFile` のインスタンスです。

`str` オブジェクトの読み書きができるファイルオブジェクトについては、*text file* も参照してください。

borrowed reference

In

Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object

`bufferobjects` をサポートしていて、C 言語の意味で **連続した** バッファーを提供可能なオブジェクト。`bytes`、`bytearray`、`array.array` や、多くの一般的な *memoryview* オブジェクトがこれに当たります。*bytes-like* オブジェクトは、データ圧縮、バイナリファイルへの保存、ソケットを経由した送信など、バイナリデータを要求するいろいろな操作に利用することができます。

幾つかの操作ではバイナリデータを変更する必要があります。その操作のドキュメントではよく "読

み書き可能な bytes-like オブジェクト” に言及しています。変更可能なバッファオブジェクトには、`bytearray` と `bytearray` の `memoryview` などが含まれます。また、他の幾つかの操作では不変なオブジェクト内のバイナリデータ (“読み出し専用の bytes-like オブジェクト”) を必要します。それには `bytes` と `bytes` の `memoryview` オブジェクトが含まれます。

bytecode

(バイトコード) Python のソースコードは、Python プログラムの CPython インタプリタの内部表現であるバイトコードへとコンパイルされます。バイトコードは `.pyc` ファイルにキャッシュされ、同じファイルが二度目に実行される時はより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。この “中間言語 (intermediate language)” は、各々のバイトコードに対応する機械語を実行する **仮想マシン** で動作するといえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することや、Python リリース間で安定であることは期待されていません。

バイトコードの命令一覧は `dis` モジュール にあります。

callable

A

callable is an object that can be called, possibly with a set of arguments (see [argument](#)), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A [function](#), and by extension a [method](#), is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback

(コールバック) 将来のある時点で実行されるために引数として渡される関数

クラス

(クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

class variable

(クラス変数) クラス上に定義され、クラスレベルで (つまり、クラスのインスタンス上ではなしに) 変更されることを目的としている変数です。

closure variable

A

[free variable](#) referenced from a [nested scope](#) that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are [free variables](#), but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
```

(次のページに続く)

(前のページからの続き)

```

    nonlocal x
    x += 1
    print(x)
return inner

```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number

(複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (-1 の平方根) に実数を掛けたもので、一般に数学では i と書かれ、工学では j と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に j をつけて書きます。例えば $3+1j$ です。`math` モジュールの複素数版を利用するには、`cmath` を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってもよいでしょう。

context

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

コンテキスト管理プロトコル

The `__enter__()` and `__exit__()` methods called by the `with` statement. See [PEP 343](#).

context manager

An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See [PEP 343](#).

context variable

A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous

(隣接、連続) バッファが厳密に **C-連続** または *Fortran 連続* である場合に、そのバッファは連続しているとみなせます。ゼロ次元バッファは C 連続であり Fortran 連続です。一次元の配列では、その要素は必ずメモリ上で隣接するように配置され、添字がゼロから始まり増えていく順序で並びます。多次元の C-連続な配列では、メモリアドレス順に要素を巡る際には最後の添え字が最初に変わるのに対し、Fortran 連続な配列では最初の添え字が最初に動きます。

コルーチン

(コルーチン) コルーチンはサブルーチンのより一般的な形式です。サブルーチンには決められた地点から入り、別の決められた地点から出ます。コルーチンには多くの様々な地点から入る、出る、再開することができます。コルーチンは `async def` 文で実装できます。[PEP 492](#) を参照してください。

coroutine function

(コルーチン関数) *coroutine* オブジェクトを返す関数です。コルーチン関数は `async def` 文で実装され、`await`、`async for`、および `async with` キーワードを持つことが出来ます。これらは [PEP 492](#) で導入されました。

CPython

python.org で配布されている、Python プログラミング言語の標準的な実装です。”CPython” という単語は、この実装を Jython や IronPython といった他の実装と区別する必要がある場合に利用されます。

current context

The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

decorator

(デコレータ) 別の関数を返す関数で、通常、`@wrapper` 構文で関数変換として適用されます。デコレータの一般的な利用例は、`classmethod()` と `staticmethod()` です。

デコレータの文法はシンタックスシュガーです。次の2つの関数定義は意味的に同じものです:

```
def f(arg):
    ...

f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、関数定義 および クラス定義 のドキュメントを参照してください。

descriptor

Any 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.

デスクリプタのメソッドに関する詳細は、[descriptors](#) や [Descriptor How To Guide](#) を参照してください

さい。

dictionary

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.

dictionary comprehension

(辞書内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる辞書を返すコンパクトな書き方です。 `results = {n: n ** 2 for n in range(10)}` とすると、キー `n` を値 `n ** 2` に対応付ける辞書を生成します。comprehensions を参照してください。

dictionary view

(辞書ビュー) `dict.keys()`、`dict.values()`、`dict.items()` が返すオブジェクトです。辞書の項目の動的なビューを提供します。すなわち、辞書が変更されるとビューはそれを反映します。辞書ビューを強制的に完全なリストにするには `list(dictview)` を使用してください。dict-views を参照してください。

docstring

A

string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing

あ

るオブジェクトが正しいインターフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。(「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。」) インターフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許して柔軟性を向上させます。ダックタイピングは `type()` や `isinstance()` による判定を避けます。(ただし、ダックタイピングを **抽象基底クラス** で補完することもできます。) その代わりに、典型的に `hasattr()` 判定や *EAFP* プログラミングを利用します。

EAFP

「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。この Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、`try` 文および `except` 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている *LBYL* スタイルと対照的なものです。

expression

(式) 何かの値と評価される、一まとまりの構文 (a piece of syntax) です。言い換えると、式とはリテラル、名前、属性アクセス、演算子や関数呼び出しなど、値を返す式の要素の積み重ねです。他の多くの言語と違い、Python では言語の全ての構成要素が式というわけではありません。while のように、式としては使えない **文** もあります。代入も式ではなく文です。

extension module

(拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コア

やユーザーコードとやりとりします。

f-string

'f' や 'F' が先頭に付いた文字列リテラルは "f-string" と呼ばれ、これは フォーマット済み文字列リテラル の短縮形の名称です。 [PEP 498](#) も参照してください。

file object

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

ファイルオブジェクトには実際には 3 種類あります: 生の [バイナリーファイル](#)、バッファされた [バイナリーファイル](#)、そして [テキストファイル](#) です。インターフェイスは `io` モジュールで定義されています。ファイルオブジェクトを作る標準的な方法は `open()` 関数を使うことです。

file-like object

file object と同義です。

filesystem encoding and error handler

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

ファイルシステムのエンコーディングでは、すべてが 128 バイト以下に正常にデコードされることが保証されなくてはなりません。ファイルシステムのエンコーディングでこれが保証されなかった場合は、API 関数が `UnicodeError` を送出することがあります。

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder

(ファインダ) インポートされているモジュールの *loader* の発見を試行するオブジェクトです。

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See `finders-and-loaders` and `importlib` for much more detail.

floor division

(切り捨て除算) 一番近い整数に切り捨てる数学的除算。切り捨て除算演算子は `//` です。例えば、`11 // 4` は `2` になり、それとは対称に浮動小数点数の真の除算では `2.75` が返ってきます。`(-11) // 4` は `-2.75` を **小さい方に丸める** (訳注: 負の無限大への丸めを行う) ので `-3` になることに注意してください。 [PEP 238](#) を参照してください。

free threading

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See [PEP 703](#).

free variable

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

関数

(関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。[仮引数](#)、[メソッド](#)、[function](#) を参照してください。

function annotation

(関数アノテーション) 関数のパラメータや返り値の *annotation* です。

関数アノテーションは、通常は **型ヒント** のために使われます: 例えば、この関数は 2 つの `int` 型の引数を取ると期待され、また `int` 型の返り値を持つと期待されています。

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

関数アノテーションの文法は [function](#) の節で解説されています。

機能の説明がある [variable annotation](#), [PEP 484](#), を参照してください。また、アノテーションを利用するベストプラクティスとして [annotations-howto](#) も参照してください。

__future__

`from __future__ import <feature>` という future 文は、コンパイラーに将来の Python リリースで標準となる構文や意味を使用して現在のモジュールをコンパイルするよう指示します。`__future__` モジュールでは、*feature* のとりうる値をドキュメント化しています。このモジュールをインポートし、その変数を評価することで、新機能が最初に言語に追加されたのはいつかや、いつデフォルトになるか(またはなったか) を見ることができます:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

(ガベージコレクション) これ以降使われることのないメモリを解放する処理です。Python は、参照カウントと、循環参照を検出し破壊する循環ガベージコレクタを使ってガベージコレクションを行います。ガベージコレクタは `gc` モジュールを使って操作できます。

ジェネレータ

(ジェネレータ) *generator iterator* を返す関数です。通常の関数に似ていますが、`yield` 式を持つ点で

異なります。yield 式は、for ループで使用できたり、next() 関数で値を 1 つずつ取り出したりできる、値の並びを生成するのに使用されます。

通常はジェネレータ関数を指しますが、文脈によっては **ジェネレータイテレータ** を指す場合があります。意図された意味が明らかなでない場合、明瞭化のために完全な単語を使用します。

generator iterator

(ジェネレータイテレータ) *generator* 関数で生成されるオブジェクトです。

yield のたびに局所実行状態 (局所変数や未処理の try 文などを含む) を記憶して、処理は一時的に中断されます。**ジェネレータイテレータ** が再開されると、中断した位置を取得します (通常関数が実行のたびに新しい状態から開始するのと対照的です)。

generator expression

An *expression* that returns an *iterator*. It looks like a normal expression followed by a for clause defining a loop variable, range, and an optional if clause. 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
```

generic function

(ジェネリック関数) 異なる型に対し同じ操作をする関数群から構成される関数です。呼び出し時にどの実装を用いるかはディスパッチアルゴリズムにより決定されます。

single dispatch、`functools.singledispatch()` デコレータ、**PEP 443** を参照してください。

generic type

A *type* that can be parameterized; typically a container class such as list or dict. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL

global interpreter lock を参照してください。

global interpreter lock

(グローバルインタプリタロック) *CPython* インタプリタが利用している、一度に Python の **バイトコード** を実行するスレッドは一つだけであることを保証する仕組みです。これにより (dict などの重要な組み込み型を含む) オブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストと引き換えに、インタプリタを簡単にマルチスレッド化できるようになります。

ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解除するように設計されています。また、I/O 処理をする場合 GIL は常に解除されます。

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration.

After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see [PEP 703](#).

hash-based pyc

(ハッシュベース pyc ファイル) 正当性を判別するために、対応するソースファイルの最終更新時刻ではなくハッシュ値を使用するバイトコードのキャッシュファイルです。pyc-invalidation を参照してください。

hashable

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__()` method). Hashable objects which compare equal must have the same hash value.

ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

Python のイミュータブルな組み込みオブジェクトは、ほとんどがハッシュ可能です。(リストや辞書のような) ミュータブルなコンテナはハッシュ不可能です。(タプルや `frozenset` のような) イミュータブルなコンテナは、要素がハッシュ可能であるときのみハッシュ可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て (自身を除いて) 比較結果は非等価であり、ハッシュ値は `id()` より得られます。

IDLE

Python の統合開発環境 (Integrated DeveLopment Environment) 及び学習環境 (Learning Environment) です。idle は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

永続オブジェクト (immortal)

Immortal objects are a CPython implementation detail introduced in [PEP 683](#).

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

immutable

(イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

import path

path based finder が `import` するモジュールを検索する場所 (または *path entry*) のリスト。`import` 中、このリストは通常 `sys.path` から来ますが、サブパッケージの場合は親パッケージの `__path__` 属性からも来ます。

importing

あ
るモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer

モ

ジュールを探してロードするオブジェクト。*finder* と *loader* のどちらでもあるオブジェクト。

interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see `tut-interac`.

interpreted

Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発／デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。[対話的](#) も参照してください。

interpreter shutdown

Python インタープリターはシャットダウンを要請された時に、モジュールやすべてのクリティカルな内部構造をなどの、すべての確保したリソースを段階的に開放する、特別なフェーズに入ります。このフェーズは [ガベージコレクタ](#) を複数回呼び出します。これによりユーザー定義のデストラクターや `weakref` コールバックが呼び出されることがあります。シャットダウンフェーズ中に実行されるコードは、それが依存するリソースがすでに機能しない (よくある例はライブラリーモジュールや `warning` 機構です) ために様々な例外に直面します。

インタープリタがシャットダウンする主な理由は `__main__` モジュールや実行されていたスクリプトの実行が終了したことです。

iterable

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`, [file objects](#), and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements [sequence](#) semantics.

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](#).

iterator

データの流れを表現するオブジェクトです。イテレータの `__next__()` メソッドを繰り返し呼び出す (または組み込み関数 `next()` に渡す) と、流れの中の要素を一つずつ返します。データがなくなると、代わりに `StopIteration` 例外を送出します。その時点で、イテレータオブジェクトは尽きており、それ以降は `__next__()` を何度呼んでも `StopIteration` を送 out します。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならないので、イテレータは他の

iterable を受理するほとんどの場所で利用できます。はっきりとした例外は複数の反復を行うようなコードです。(list のような) コンテナオブジェクトは、自身を `iter()` 関数にオブジェクトに渡したり `for` ループ内で使うたびに、新たな未使用のイテレータを生成します。これをイテレータで行おうとすると、前回のイテレーションで使用済みの同じイテレータオブジェクトを単純に返すため、空のコンテナのようになってしまいます。

詳細な情報は `typeiter` にあります。

CPython 実装の詳細: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function

(キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、`locale.strxfrm()` をキー関数にえば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python の多くのツールはキー関数を受け取り要素の並び順やグループ化を管理します。`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 等があります。

キー関数を作る方法はいくつかあります。例えば `str.lower()` メソッドを大文字小文字を区別しないソートを行うキー関数として使うことが出来ます。あるいは、`lambda r: (r[0], r[2])` のような `lambda` 式からキー関数を作ることができます。また、`operator.attrgetter()`, `operator.itemgetter()`, `operator.methodcaller()` の 3 つのキー関数コンストラクタがあります。キー関数の作り方と使い方の例は `Sorting HOW TO` を参照してください。

keyword argument

実

[引数](#) を参照してください。

lambda

(ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの [式](#) を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL

「ころばぬ先の杖 (look before you leap)」の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。[EAFP](#) アプローチと対照的で、`if` 文がたくさん使われるのが特徴的です。

マルチスレッド化された環境では、LBYL アプローチは ”見る” 過程と ”飛ぶ” 過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に `mapping` から `key` を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

list

A

built-in Python [sequence](#). Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension

(リスト内包表記) シーケンス中の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな方法です。`result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。if 節はオプションです。if 節がない場合、`range(256)` の全ての要素が処理されます。

loader

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See also:

- [finders-and-loaders](#)
- `importlib.abc.Loader`
- [PEP 302](#)

ロケールエンコーディング

On Unix, it is the encoding of the `LC_CTYPE` locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: "cp1252").

On Android and VxWorks, Python uses "utf-8" as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method

special method のくだけた同義語です。

mapping

(マッピング) 任意のキー探索をサポートしていて、`collections.abc.Mapping` か `collections.abc.MutableMapping` の 抽象基底クラス で指定されたメソッドを実装しているコンテナオブジェクトです。例えば、`dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` などです。

meta path finder

`sys.meta_path` を検索して得られた *finder*. meta path finder は *path entry finder* と関係はありませんが、別物です。

meta path finder が実装するメソッドについては `importlib.abc.MetaPathFinder` を参照してください。

metaclass

(メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注: メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーにとって、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。た

たとえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

詳細は `metaclasses` を参照してください。

メソッド

(メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一 **引数** として受け取ります (この第一引数は通常 `self` と呼ばれます)。**関数** と **ネストされたスコープ** も参照してください。

method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See `python_2.3_mro` for details of the algorithm used by the Python interpreter since the 2.3 release.

module

(モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは *importing* の処理によって Python に読み込まれます。

パッケージ を参照してください。

module spec

モ

ジュールをロードするのに使われるインポート関連の情報を含む名前空間です。`importlib.machinery.ModuleSpec` のインスタンスです。

See also `module-specs`.

MRO

method resolution order を参照してください。

mutable

(ミュータブル) ミュータブルなオブジェクトは、`id()` を変えることなく値を変更できます。**イミュータブル** も参照してください。

named tuple

”

名前付きタプル” という用語は、タプルを継承していて、インデックスが付く要素に対し属性を使ってのアクセスもできる任意の型やクラスに应用されています。その型やクラスは他の機能も持っていることもあります。

`time.localtime()` や `os.stat()` の戻り値を含むいくつかの組み込み型は名前付きタプルです。他の例は `sys.float_info` です:

```
>>> sys.float_info[1]           # indexed access
1024
>>> sys.float_info.max_exp      # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace

(名前空間) 変数が格納される場所です。名前空間は辞書として実装されます。名前空間にはオブジェクトの (メソッドの) 入れ子になったものだけでなく、局所的なもの、大域的なもの、そして組み込みのものがああります。名前空間は名前の衝突を防ぐことによってモジュール性をサポートする。例えば関数 `builtins.open` と `os.open()` は名前空間で区別されています。また、どのモジュールが関数を実装しているか明示することによって名前空間は可読性と保守性を支援します。例えば、`random.seed()` や `itertools.islice()` と書くと、それぞれモジュール `random` や `itertools` で実装されていることが明らかです。

namespace package

(名前空間パッケージ) サブパッケージのコンテナとしてのみ提供される [PEP 420](#) で定義された *package* です。名前空間パッケージは物理的な表現を持たないことができ、`__init__.py` ファイルを持たないため、*regular package* とは異なります。

module を参照してください。

nested scope

(ネストされたスコープ) 外側で定義されている変数を参照する機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープはデフォルトでは変数の参照だけができ、変数の代入はできないので注意してください。ローカル変数は、最も内側のスコープで変数を読み書きします。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。`nonlocal` で外側の変数に書き込みます。

new-style class

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattr__()`, class methods, and static methods.

object

(オブジェクト) 状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての [新スタイルクラス](#) の究極の基底クラスのこと。

optimized scope

A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package

(パッケージ) サブモジュールや再帰的にサブパッケージを含むことの出来る *module* のことです。専

門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

regular package と *namespace package* を参照してください。

parameter

(仮引数) 名前付の実体で **関数** (や **メソッド**) の定義において関数が受ける **実引数** を指定します。仮引数には 5 種類あります:

- **位置またはキーワード:** **位置** あるいは **キーワード引数** として渡すことができる引数を指定します。これはたとえば以下の `foo` や `bar` のように、デフォルトの仮引数の種類です:

```
def func(foo, bar=None): ...
```

- **位置専用:** 位置によってのみ与えられる引数を指定します。位置専用の引数は 関数定義の引数のリストの中でそれらの後ろに `/` を含めることで定義できます。例えば下記の `posonly1` と `posonly2` は位置専用引数になります:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- **キーワード専用:** キーワードによってのみ与えられる引数を指定します。キーワード専用の引数を定義できる場所は、例えば以下の `kw_only1` や `kw_only2` のように、関数定義の仮引数リストに含めた可変長位置引数または裸の `*` の後です:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- **可変長位置:** (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の `args` のように仮引数名の前に `*` をつけることで定義できます:

```
def func(*args, **kwargs): ...
```

- **可変長キーワード:** (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の `kwargs` のように仮引数名の前に `**` をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

仮引数、FAQ の **実引数と仮引数の違いは何ですか?**、`inspect.Parameter` クラス、`function` セクション、**PEP 362** を参照してください。

path entry

path based finder が `import` するモジュールを探す *import path* 上の 1 つの場所です。

path entry finder

`sys.path_hooks` にある callable (つまり *path entry hook*) が返した *finder* です。与えられた *path entry* にあるモジュールを見つける方法を知っています。

パスエントリーファインダが実装するメソッドについては `importlib.abc.PathEntryFinder` を参照してください。

path entry hook A
callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder デ
フォルトの *meta path finder* の 1 つは、モジュールの *import path* を検索します。

path-like object
(path-like オブジェクト) ファイルシステムパスを表します。path-like オブジェクトは、パスを表す `str` オブジェクトや `bytes` オブジェクト、または `os.PathLike` プロトコルを実装したオブジェクトのどれかです。`os.PathLike` プロトコルをサポートしているオブジェクトは `os.fspath()` を呼び出すことで `str` または `bytes` のファイルシステムパスに変換できます。`os.fsdecode()` と `os.fsencode()` はそれぞれ `str` あるいは `bytes` になるのを保証するのに使えます。**PEP 519** で導入されました。

PEP

Python Enhancement Proposal. PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

PEP 1 を参照してください。

portion
PEP 420 で定義されている、namespace package に属する、複数のファイルが (zip ファイルに格納されている場合もある) 1 つのディレクトリに格納されたもの。

位置引数 (positional argument) 実
引数 を参照してください。

provisional API
(暫定 API) 標準ライブラリの後方互換性保証から計画的に除外されたものです。そのようなインターフェースへの大きな変更は、暫定であるとされている間は期待されていませんが、コア開発者によって必要とみなされれば、後方非互換な変更 (インターフェースの削除まで含まれる) が行われえます。このような変更はむやみに行われるものではありません -- これは API を組み込む前には見落とされていた重大な欠陥が露呈したときにのみ行われます。

暫定 API についても、後方互換性のない変更は「最終手段」とみなされています。問題点が判明した場合でも後方互換な解決策を探すべきです。

このプロセスにより、標準ライブラリは問題となるデザインエラーに長い間閉じ込められることなく、時代を超えて進化を続けられます。詳細は **PEP 411** を参照してください。

provisional package

provisional API を参照してください。

Python 3000

Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) "Py3k" と略されることもあります。

Pythonic

他

の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは `for` 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

```
for i in range(len(food)):
    print(food[i])
```

これに対し、きれいな Pythonic な方法は:

```
for piece in food:
    print(piece)
```

qualified name

(修飾名) モジュールのグローバルスコープから、そのモジュールで定義されたクラス、関数、メソッドへの、"パス" を表すドット名表記です。**PEP 3155** で定義されています。トップレベルの関数やクラスでは、修飾名はオブジェクトの名前と同じです:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

モジュールへの参照で使われると、**完全修飾名** (*fully qualified name*) はすべての親パッケージを含む全体のドット名表記、例えば `email.mime.text` を意味します:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

(参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。永続 であり、参照カウントが決して変更されないために割り当てが解除されないオブジェクトもあります。参照カウントは通常は Python のコード上には現れませんが、CPython 実装の重要な要素です。プログラマーは、任意のオブジェクトの参照カウントを知るために `sys.getrefcount()` 関数を呼び出すことが出来ます。

regular package

伝

統的な、`__init__.py` ファイルを含むディレクトリとしての *package*。

namespace package を参照してください。

REPL

”read – eval – print loop” の頭字語で、対話型 インタープリターシェルの別名。

`__slots__`

ク

ラス内での宣言で、インスタンス属性の領域をあらかじめ定義しておき、インスタンス辞書を排除することで、メモリを節約します。これはよく使われるテクニックですが、正しく扱うには少しトリッキーなので、稀なケース、例えばメモリが死活問題となるアプリケーションでインスタンスが大量に存在する、といったときを除き、使わないのがベストです。

sequence

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 `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension

(集合内包表記) `iterable` 内の全てあるいは一部の要素を処理して、その結果からなる集合を返すコンパクトな書き方です。 `results = {c for c in 'abracadabra' if c not in 'abc'}` とすると、`{'r', 'd'}` という文字列の辞書を生成します。comprehensions を参照してください。

single dispatch

generic function の一種で実装は一つの引数の型により選択されます。

slice

(スライス) 一般に シーケンス の一部を含むオブジェクト。スライスは、添字表記 `[]` で与えられた複数の数の間にコロンを書くことで作られます。例えば、`variable_name[1:3:5]` です。角括弧 (添字) 記号は `slice` オブジェクトを内部で利用しています。

soft deprecated

A

soft deprecated API should not be used in new code, but it is safe for already existing code to use

it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See [PEP 387: Soft Deprecation](#).

special method

(特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついています。特殊メソッドについては [specialnames](#) で解説されています。

statement

(文) 文はスイート (コードの”ブロック”) に不可欠な要素です。文は [式](#) かキーワードから構成されるもののどちらかです。後者には `if`、`while`、`for` があります。

static type checker

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also [type hints](#) and the [typing](#) module.

strong reference

In

Python’s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also [borrowed reference](#).

text encoding

A

string in Python is a sequence of Unicode code points (in range U+0000--U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as ”encoding”, and recreating the string from the sequence of bytes is known as ”decoding”.

There are a variety of different text serialization codecs, which are collectively referred to as ”text encodings”.

text file

(テキストファイル) `str` オブジェクトを読み書きできる [file object](#) です。しばしば、テキストファイルは実際にバイト指向のデータストリームにアクセスし、[テキストエンコーディング](#) を自動的に行います。テキストファイルの例は、`sys.stdin`、`sys.stdout`、`io.StringIO` インスタンスなどをテキストモード (`'r'` or `'w'`) で開いたファイルです。

[bytes-like](#) **オブジェクト** を読み書きできるファイルオブジェクトについては、[バイナリファイル](#) も参照してください。

triple-quoted string

(三重クォート文字列) 3つの連続したクォート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1つか2つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることのできるため、ドキュメンテーション文字列を書く時に特に便利です。

type

The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias

(型エイリアス) 型の別名で、型を識別子に代入して作成します。

型エイリアスは [型ヒント](#) を単純化するのに有用です。例えば:

```
def remove_gray_shades(
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

これは次のように読みやすくなります:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

機能の説明がある [typing](#) と [PEP 484](#) を参照してください。

type hint

(型ヒント) 変数、クラス属性、関数のパラメータや戻り値の期待される型を指定する *annotation* です。

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

グローバル変数、クラス属性、関数で、ローカル変数でないものの型ヒントは `typing.get_type_hints()` で取得できます。

機能の説明がある [typing](#) と [PEP 484](#) を参照してください。

universal newlines

デ

キストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 '`\n`'、Windows の規定 '`\r\n`'、古い Macintosh の規定 '`\r`'。利用法について詳しくは、[PEP 278](#) と [PEP 3116](#)、さらに `bytes.splitlines()` も参照してください。

variable annotation

(変数アノテーション) 変数あるいはクラス属性の *annotation* 。

変数あるいはクラス属性に注釈を付けたときは、代入部分は任意です:


```
class C:
    field: 'annotation'
```

変数アノテーションは通常は **型ヒント** のために使われます: 例えば、この変数は `int` の値を取ることを期待されています:

```
count: int = 0
```

変数アノテーションの構文については `annassign` 節で解説しています。

機能の説明がある *function annotation*, **PEP 484**, **PEP 526** を参照してください。また、アノテーションを利用するベストプラクティスとして `annotations-howto` も参照してください。

virtual environment

(仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

`venv` を参照してください。

virtual machine

(仮想マシン) 完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力した **バイトコード** を実行します。

Zen of Python

(Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで `"import this"` とするとこのリストを読めます。

ABOUT THIS DOCUMENTATION

Python's documentation is generated from [reStructuredText](#) sources using [Sphinx](#), a documentation generator originally created for Python and now maintained as an independent project.

ドキュメントとそのツール群の開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて [reporting-bugs](#) ページをご覧ください。新しいボランティアはいつでも歓迎です！（訳注：日本語訳の問題については、GitHub 上の [Issue Tracker](#) で報告をお願いします。）

多大な感謝を：

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and author of much of the content;
- [Docutils](#) プロジェクト [reStructuredText](#) と [Docutils](#) ツールセットを作成しました。
- Fredrik Lundh の [Alternative Python Reference](#) プロジェクトから [Sphinx](#) は多くのアイデアを得ました。

B.1 Contributors to the Python documentation

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の [Misc/ACKS](#) に、それら貢献してくれた人々を部分的にはありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメンテーションは生まれませんでした -- ありがとう！

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/> 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, <https://www.cnri.reston.va.us/> 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、Guido と Python のコア開発チームは BeOpen.com に移り、BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, <https://www.zope.org/> 参照) に移りました。そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF, <https://www.python.org/psf/> 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は <https://opensource.org/> を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

i 注釈

「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にくださった多くのボランティアのみなさんに感謝します。

C.2 Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.1

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.13.1 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 3.13.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights Reserved" are retained in Python 3.13.1 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 3.13.1 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 3.13.1.
4. PSF is making Python 3.13.1 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 3.13.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.1, 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. 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 3.13.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 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

(次のページに続く)

(前のページからの続き)

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 <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 CNRI LICENSE AGREEMENT FOR 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>."

(次のページに続く)

(前のページからの続き)

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 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 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.1 DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, 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 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` C extension underlying the `random` module includes code based on a download from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

```
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).
```

```
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.
```

(次のページに続く)

(前のページからの続き)

```
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 ソケット

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <https://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.

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 Asynchronous socket services

The `test.support.asyncchat` and `test.support.asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

(次のページに続く)

(前のページからの続き)

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.3.4 Cookie management

The `http.cookies` 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

(次のページに続く)

(前のページからの続き)

PERFORMANCE OF THIS SOFTWARE.

C.3.5 Execution tracing

The trace module contains the following notice:

```
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.3.6 UUencode and UUdecode functions

The uu codec contains the following notice:

```
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,
```

(次のページに続く)

(前のページからの続き)

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 version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` 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

(次のページに続く)

(前のページからの続き)

```
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.3.8 test_epoll

The `test.test_epoll` module 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
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.3.9 Select kqueue

`select` モジュールは `kqueue` インターフェースについての次の告知を含んでいます:

```
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
```

(次のページに続く)

(前のページからの続き)

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.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski's implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

<MIT License>

Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

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.

</MIT License>

Original location:

<https://github.com/majek/csiphash/>

Solution inspired by code from:

(次のページに続く)

(前のページからの続き)

Samuel Neves (supercop/crypto_auth/siphash24/little)
 djb (supercop/crypto_auth/siphash24/little2)
 Jean-Philippe Aumasson (<https://131002.net/siphash/siphash24.c>)

C.3.11 strtod と dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```

/*****
*
* 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.
*
* 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.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

Apache License
 Version 2.0, January 2004
<https://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

(次のページに続く)

(前のページからの続き)

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

(次のページに続く)

(前のページからの続き)

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the

(次のページに続く)

(前のページからの続き)

Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
- (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
- (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work

(次のページに続く)

(前のページからの続き)

by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason

(次のページに続く)

(前のページからの続き)

of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

C.3.13 expat

The `pyexpat` extension is built using an included copy of the `expat` sources unless the build is configured `--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.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the `libffi` sources unless the build is configured `--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,

(次のページに続く)

(前のページからの続き)

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 zlib

The `zlib` extension is built using an included copy of the `zlib` sources if the `zlib` version found on the system is too old to be used for the build:

Copyright (C) 1995-2011 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

(次のページに続く)

(前のページからの続き)

jloup@gzip.org

madler@alumni.caltech.edu

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` で使用しているハッシュテーブルの実装は、cfuhash プロジェクトのものに基づきます:

```
Copyright (c) 2005 Don Owens
```

```
All rights reserved.
```

```
This code is released under the BSD license:
```

```
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:
```

- ```
* Redistributions of source code must retain the above copyright
 notice, this list of conditions and the following disclaimer.
```
- ```
* 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.
```
- ```
* Neither the name of the author nor the names of its
 contributors may 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.
```

### C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. 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.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

```
* Redistributions of works must retain the original copyright notice,
```

(次のページに続く)

(前のページからの続き)

this list of conditions and the following disclaimer.

- \* Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work 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.

### C.3.19 mimalloc

MIT License:

Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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

(次のページに続く)

(前のページからの続き)

SOFTWARE.

### C.3.20 asyncio

Parts of the asyncio module are incorporated from [uvloop 0.16](#), which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc. http://magic.io
```

```
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.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in `subr_smr.c`. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>
```

```
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 unmodified, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright

(次のページに続く)

(前のページからの続き)

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 ``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 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.



COPYRIGHT

Python and this documentation is:

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

---

ライセンスおよび許諾に関する完全な情報は、[歴史とライセンス](#) を参照してください。





## 索引

## アルファベット以外

..., 95  
 >>>, 95  
 \_\_future\_\_, 103  
 \_\_slots\_\_, 114  
 クラス, 98  
 コルーチン, 100  
 コンテキスト管理プロトコル, 99  
 ジェネレータ, 103  
 メソッド, 109  
   magic, 108  
   特殊, 115  
 ロケールエンコーディング, 108  
 位置引数 (*positional argument*), 112  
 特殊  
   メソッド, 115  
 環境変数  
   PATH, 66  
   PYTHON\_GIL, 105  
   PYTHONDONTWRITEBYTECODE, 46  
 関数, 103

## A

abstract base class, 95  
 annotation, 95  
 asynchronous context manager, 96  
 asynchronous generator, 96  
 asynchronous generator iterator, 96  
 asynchronous iterable, 97  
 asynchronous iterator, 97  
 awaitable, 97

## B

BDFL, 97  
 binary file, 97  
 borrowed reference, 97  
 bytecode, 98  
 bytes-like object, 97

## C

callable, 98  
 callback, 98  
 C-contiguous, 99  
 class variable, 98  
 closure variable, 98  
 complex number, 99  
 context, 99  
 context manager, 99  
 context variable, 99  
 contiguous, 99  
 coroutine function, 100

CPython, 100  
 current context, 100

## D

decorator, 100  
 descriptor, 100  
 dictionary, 101  
 dictionary comprehension, 101  
 dictionary view, 101  
 docstring, 101  
 duck-typing, 101

## E

EAFP, 101  
 expression, 101  
 extension module, 101

## F

f-string, 102  
 file object, 102  
 file-like object, 102  
 filesystem encoding and error  
   handler, 102  
 finder, 102  
 floor division, 102  
 Fortran contiguous, 99  
 free threading, 103  
 free variable, 103  
 function annotation, 103

## G

garbage collection, 103  
 generator expression, 104  
 generator iterator, 104  
 generic function, 104  
 generic type, 104  
 GIL, 104  
 global interpreter lock, 104

## H

hash-based pyc, 105  
 hashable, 105

## I

IDLE, 105  
 immutable, 105  
 import path, 105  
 importer, 106  
 importing, 105

interactive, 106  
 interpreted, 106  
 interpreter shutdown, 106  
 iterable, 106  
 iterator, 106

## K

key function, 107  
 keyword argument, 107

## L

lambda, 107  
 LBYL, 107  
 list, 107  
 list comprehension, 107  
 loader, 108

## M

magic  
   メソッド, 108  
 magic method, 108  
 mapping, 108  
 meta path finder, 108  
 metaclass, 108  
 method resolution order, 109  
 module, 109  
 module spec, 109  
 MRO, 109  
 mutable, 109

## N

named tuple, 109  
 namespace, 110  
 namespace package, 110  
 nested scope, 110  
 new-style class, 110

## O

object, 110  
 optimized scope, 110

## P

package, 110  
 parameter, 111  
   difference from argument, 16  
 PATH, 66  
 path based finder, 112  
 path entry, 111  
 path entry finder, 111

path entry hook, [112](#)  
path-like object, [112](#)  
PEP, [112](#)  
portion, [112](#)  
provisional API, [112](#)  
provisional package, [113](#)  
Python 3000, [113](#)  
Python Enhancement Proposals  
  PEP 1, [112](#)  
  PEP 5, [6](#)  
  PEP 8, [10](#), [43](#), [89](#)  
  PEP 238, [102](#)  
  PEP 278, [116](#)  
  PEP 302, [108](#)  
  PEP 343, [99](#)  
  PEP 362, [96](#), [111](#)  
  PEP 373, [5](#)  
  PEP 387, [3](#)  
  PEP 411, [112](#)  
  PEP 420, [110](#), [112](#)  
  PEP 443, [104](#)  
  PEP 483, [104](#)  
  PEP 484, [95](#), [103](#), [104](#), [116](#), [117](#)  
  PEP 492, [96](#), [97](#), [100](#)  
  PEP 498, [102](#)  
  PEP 519, [112](#)  
  PEP 525, [96](#)  
  PEP 526, [95](#), [117](#)  
  PEP 572, [53](#)

  PEP 585, [104](#)  
  PEP 602, [5](#)  
  PEP 683, [105](#)  
  PEP 703, [71](#), [103](#), [105](#)  
  PEP 3116, [116](#)  
  PEP 3147, [46](#)  
  PEP 3155, [113](#)  
PYTHON\_GIL, [105](#)  
PYTHONDONTWRITEBYTECODE, [46](#)  
Pythonic, [113](#)

## Q

qualified name, [113](#)

## R

reference count, [113](#)  
regular package, [114](#)  
REPL, [114](#)

## S

sequence, [114](#)  
set comprehension, [114](#)  
single dispatch, [114](#)  
slice, [114](#)  
soft deprecated, [114](#)  
special method, [115](#)  
statement, [115](#)  
static type checker, [115](#)

strong reference, [115](#)

## T

text encoding, [115](#)  
text file, [115](#)  
triple-quoted string, [116](#)  
type, [116](#)  
type alias, [116](#)  
type hint, [116](#)

## U

universal newlines, [116](#)

## V

variable annotation, [116](#)  
virtual environment, [117](#)  
virtual machine, [117](#)  
属性, [97](#)  
引数 (*argument*), [96](#)  
  difference from parameter, [16](#)

## W

永続オブジェクト (*immortal*), [105](#)

## Z

Zen of Python, [117](#)