Python2からPython3へ移行するときに知っておくといいかもしれない幾つかの事

リリース *1.0beta03*

Noboru Yamamoto [1],
J-PARC/KEK,
Tsukuba & Tokai, Ibaraki, Japapn

2024 年 02 月 16 日

Contents:

[1] mail-to:noboru.yamamoto at kek.jp

1 始めに

Python2 系列の開発およびサポートは 2020 年 1 月 1 日以降は完全に停止するとのアナウンスされています (PEP-344 PEP-344 web page)(現時点 2023 年 12 月 15 日 では既に EOL が宣言され, 更新は完全に停止されています。) 。

上記の期限をすぎた後も、Python2.7 が直ちに使用不能となる訳ではありませんが、セキュリティパッチも供給されず、新しい機能の追加もなくなります。なるだけ早く Python3 への移行を進めるしか方法は無いようです。この小文では、いくつかの python2 プログラムを Python2/Python3 対応に書き換えた経験を踏まえて、python2 プログラムを Python3 対応のプログラムに書き換える際の幾つかのヒントをまとめています。

幸いなことに、Pyhton2 から Python3 への移行をサポートするためのツールやモジュールが用意されています。`2to3` プログラムは Python の標準配布に含まれるモジュール／ツールで、python2 プログラムを python3 プログラムに書き換えてくれます。この `2to3` の使い方は次の第 3 章で説明します。

`2to3` は python2 から python3 への主な文法上の変更点を自動的に行ってくれます. しかしながら、いくつかの場合には、`2to3` で変換したプログラムを実行することはできるものの、結果が意図した物とは異なる場合がありました。此の場合には、`2to3` で変換したプログラムに手を加える必要がありました。特に、計測機器などとのバイナリデータのやり取りを行う場合には、python2 と python3 での文字列の扱いの違い (python2 では bytes 列、python3 ではユニコード文字列) に注意が必要です。第 4 章ではこれについて説明します。

また、2to3 で変換したスクリプトは python3 で動作しますが、そのままでは python2 では動作しなくなることもママ有ります。`future` モジュールを python2 に導入することによって、2to3 で変換したプログラムが python2 でも動作するようにすることが容易になります。`future` モジュールは標準モジュールではないものの、広く使われています。第 5 章 でこの `future` モジュールを紹介します。

プログラムが python2 および python3 の双方に対応させるのをサポートするモジュール `six` も広く使われています。これについて続く第 6 章で触れます。

私が Python3 対応に書き換えたプログラムには、Cython を使ったプログラムも含まれています。その際に注意すべき点について、第 7 章で説明します。

`py2app` `py2exe` は Macitosn OS および Windows OS でそれぞれ python で作成したプログラムをスタンドアローン アプリケーションに変換するプログラムです。実際には Python プログラムにラッパーをかけて、単独アプリケーションとして動作するようにしています。これらツールもサポートされる環境が Python3 に移行する方向で進んでいます。

実際新しい Windows のバージョンで動作する py2exe は Python3 のみで動作します。一方、Macintosh OS ではシステム付属の python は python2 系であるために、システム機能を, 特に GUI 環境, を安定に利用するためには システム付属の /usr/bin/pythonw を使う必要があります。(MacOS でも python2 のサポートは廃止の予定であり、現段階では python2 もまだ利用可能ではありますが、python3 への移行をすすめる必要があります。)

その際に注意すべき点について、第 8 章で説明します。

第 A 章ではこれまでの各章でカバーされなかった細々したことを説明します。

2 How to port Python 2 Code to Python 3

Python の公式ドキュメントページ中に How to portPython Code to Python3 があります。 "Japanese" タグを選択することで（未翻訳の部分も多いのですが）日本語版も閲覧できます。 ここには、著者が重要と思うこの文書の一部を抜粋して超訳したものを掲載します。

2.1 短い説明

一つのソースコードを Python 2/3 の両方に対応させるための基本的な手順は以下のようになります。：

1. Python 2.7 だけをサポートすることに気を配ってください。
2. 十分なカバレージを持つテストがあることを確認しましょう。(coverage.py が助けになります; `python -m pip install coverage`) （`coverage` の利用には pytest/unittest/nosetest などの test プログラムが用意されていることが前提になるようです)
3. Pytho2 と 3 の違いを学びましょう。
4. コードのアップデートに Futurize (あるいは Modernize) を使いましょう。(例えば、`python -m pip install future` でインストール)[1]
5. Pylint を使い、あなたのプログラムの Python3 サポートが後戻りしないようにしましょう。(`python -m pip install pylint`)
6. どのライブラリへの依存性が python3 への移行を妨げているかを確認するために、 caniusepython3 を使いましょう。 (`python -m pip install caniusepython3`)
7. ライブラリの依存性が解決されたら、継続的な統合手法を使い　Python2 および 3 との互換性を維持していることを確認しましょう。(tox は複数のバージョンの Python でのテストを助けてくれます。 `python -m pip install tox`)
8. あなたのプログラムでのデータ型の使い方が Python2 ／ 3 の両方で動作することを確認するために、 静的な型検査 を使うことを検討しましょう。(Python2/3 の双方でのデータ型の利用状況を検査するのに mypy を使いましょう。`python -m pip install mypy`).

注釈: 注釈: `python -m pip install` を使うことで、 `pip` を実行する Python に対応してインストールされてバージョンの `pip` であることが保証されます。システム既定の `pip` であるか virtual environment にインストールされた `pip` を適切に使い分けることができます。

---

[1] 最終更新は 2023.11.16 にチェックしたところでは、2020.10.1 の 0.9rc0 となっている。

2.2 注意を払い、確認すべき重要な事項：

2.2.1 除算

Python 3 では、`5 / 2 == 2.5` となります。これは Python2 での結果 2 とは異なっています。全ての `int` との間の除算の結果のデータ型は、`float` になります。この変更は、2002 年にリリースされた Python 2.2 から計画されました。それから、ユーザーは `from __future__ import division` を, `/` と `//` を使い分けるプログラムあるいは``-Q`` をつけて実行されるプログラムに、追加することを推奨されていました。もし、あなたがこの対応をまだ実行されていないなら、以下の二つのことを試して見る必要があるでしょう。

1. `from __future__ import division` をプログラムファイルの冒頭に追加する。
2. プログラムの除算を実行している場所で、必要とされる結果が整数の場合には `//` を、`float``の結果が必要な場所では ``/` を使うようにソースコードを変更する。

自動的に `/` が `//` に単純置換されないには理由があります。もしオブジェクトに `__turudive` は定義されいるが、 `__floordiv__` は定義されていない場合、この単純な置換を行なったプログラムは動作しなくなってしまいます。つまり、ユーザが定義したクラスが、 `/` を何らかの操作を示すために使っているが、`//` は別の意味を持たせているあるいは、それが未定義である場合です。

2.2.2 テキスト対バイナリデータ

Python 2 では `str` 型をテキストとバイナリデータのどちらにも使うことが出来ていました。不幸なことにこれは、2 つの異なる概念を重ね合わせていて、両方の種類のデータに対して、時々動作して時々はそうではない、といった傷つきやすいコードに繋がりやすいものでした。人々が特定の一つの型の代わりに `str` を受け付ける何かが、それが許容するのはテキストなのかバイナリデータなのかを名言しないときの、悩ましい API を生み出してしまう要因でもありました。これはとりわけマルチリンガルをサポートするための状況を、テキストデータをサポートしていると主張しているのに明示的に `unicode` をサポートすることに注意を払わない API、という形で複雑にしていました。

Python3 ではテキスト型とバイナリ型は全く別の型として定義されており、単純にこれらの型を混ぜて使うことはできません。テキストデータ方のみ、あるいは、バイナリデータだけを使うプログラムではこの分離は問題を起こしません。しかし、両方の型のデータを取り扱うプログラムでは、この分離は、あなたがテキストデータをバイナリデータと比較して使う場合には適切な対応を取る必要があることを示しています。これが完全に自動的な変換ができない理由です。

どの API がテキストデータを処理し、バイナリデータを処理する API はどれであるのかを決定しましょう。(両方のデータを受け付ける API を使わない設計が **強く** 推奨されています。これはこの動作を維持することの難しさによるものです。先に述べたようにこれはうまくやるには難しいことです。)

Python2 では、このことは テキスト型を取り扱う API は `unicode` を、バイナリ型を処理する API では Python3 由来の `bytes` を取り扱える必要があることを意味します。Python2 では `bytes` は `str` のサブセットで、`bytes` 型の別名として動作します。通常、最大の問題は、どのメソッドが Python2 および　Python3 で同時に利用可能なメソッドがどのデータ型に存在するのかを理解することです。テキストデータに対しては、Python2 では `str`, Python3 では `unicode` が使われます。バイナリデータに対しては、Python2 では `str` あるいは `bytes` が使われますが、Python3 では `bytes` が使われます。Python 2 and `bytes` in Python 3).

以下のテーブルは、Python2 および Python3 で共通な **ユニーク** なメソッドを示しています。つまり、 `decode()` メソッドは　Python2 でも Python3 でも等価なバイナリデータに対して使うことができます。しかしこのメソッドをテキストデータに対しては、Python2 と Python3 に対して一

貫性のあるやり方では使うことができません。Python3 の `str` にはこのメソッドが存在しないからです。　Python3.5 以降では、`__mod__()` メソッドが `bytes` 型に追加されたことにご注意ください。

Text data	Binary data
	decode
encode	
format	
isdecimal	
isnumeric	

型の区別を取り扱いに容易とすることは、テキストデータとバイナリデータの encoding と decoding による変換をプログラムの境界で行うことです。つまり、テキストデータをバイナリ型で受け取った場合には、直ちにデコードします。また、もしテキストデータをバイナリで送信する必要がある場合には、可能な限り最後までエンコードを引き伸ばします。この手法により、あなたのプログラムは内部ではテキストデータだけを取り扱い、今どちらのデータを取り扱っているのかを気にする必要がなくなります。

次の課題は、プログラム中の文字リテラルがテキストであるのかバイナリデータであるのかを明確に把握していることです。　バイナリデータを表す文字リテラルには, `b` プリフィックスをつけることを忘れてはいけません。テキストには `u` プリフィックスをつけます。 `__future__` モジュールを使うことで、全ての文字リテラルがユニコード文字列 (`u`) であることを強制することもできます、しかしこれは全ての文字列に `b` または `u` をつける方法ほどの有効性がありません。

ファイルを開く時にも注意を怠ってはいけません。　ファイルを開くときにわざわざ `b` モードで開く (つまり、`rb` をつけてバイナリファイルを開くこと）ことは常にはしていないかもしれません。Python3 ではバイナリファイルとテキストファイルは明確に区別されていて、相互に互換性はありません。(詳細は `io` モジュールをご覧ください。) 従って、ファイルを開く際には、テキストモードで使われるのか、バイナリモードで使われるのかの決定を **必ず** 決定しなければなりません。また、ビルトインの `open()` ではなく、 `io` モジュールの `io.open()` を使います。 なぜなら、`io.open()` は Python2 と Python3 で同じように動作するからです。古い慣習の `codecs.open()` を使う必要はもうありません。これは Python2.5 との互換性を保つために必要であったことだからです。

`str` と `bytes` の生成子は Python2 と Python3 では、同じ引数に対して異なった意味を持っています。 Python2 で整数に引数を `bytes` に渡すと、整数を表現する文字列が作られます (`bytes(3) == '3'`)。しかし Python3 では, 指定された整数の長さの null バイトデータが返されます (`bytes(3) == b'\x00\x00\x00'`)。 同様の配慮が、バイトオブジェクトを `str()` に渡された時に必要です。Python2 では単にそのバイトオブジェクトが返されるだけですが (`str(b'3') == b'3'`), Python3 ではバイトデータの文字列表現が返されます (`str(b'3') == "b'3'"`).

最後に、バイナリデータのインデックスによる取り扱いに関する注意です (スライスによる取り扱いでは、特別な注意は必要ありません。) Python2 では、 `b'123'[1] == b'2'` ですが、 Python 3 では `b'123'[1] == 50` となります。なぜなら、Python3 においてはバイナリデータは単純にバイナリ数の集合ですから、整数としての値が返されます。しかるに、Python2 では `bytes == str` ですから、インデックスで指定された際には、一つの要素からなるバイトデータを返します。 six プロジェクトには、 `six.indexbytes()` という関数があり、Python3 のように指定された場所のバイナリデータに対応する整数を値として返します (`six.indexbytes(b'123', 1)`)。

まとめておくと：

1. あなたが定義する API のどれがテキストデータを引数にとり、どれがバイナリデータを引数とするのかを決定する

2. テキストデータの処理を行う際には、unicode データも同じように処理されることを確実にしておくこと。Python2 プログラムのバイナリデータを取り扱う部分では、bytes も同様に取り扱えること（上記のテーブルを参照のこと）
3. 全てのバイナリ　リテラル文字列は、 b で始め、テキストデータは u で始める。
4. バイナリからテキストへの変換 (decode) は即時に行われること。 逆にテキストからバイナリに変換される場合には、できるだけ後回しにしておく。
5. ファイルを開く際は、io.open1() を使うこと。また、必要な場合には、 b モードを適切につける
6. バイナリデータのインデックスによる利用には最大の注意を払う。

3 2to3 ツールの使い方

3.1 2to3 の概要

2to3 は Python と共にインストールされる **標準的なツール** で、その名の通り Python2 向けのソースコードを、Python3 対応のソースコードに変換してくれます。後に説明する future モジュールも類似の機能を持つています (*future* モジュール (ページ 12) 参照)。

2to3 は python2 対応コードを python3 対応コードに変換する際に必要な変更のほとんどに対応していますが、若干の手直しを必要とする場合があります。しかしながら、Python2 向けソースコードを Python3 対応にする為の最初の一歩としては必須のツールです。

future モジュールの futureize コマンドや Modernize コマンドも内部では lib2to3 の機能を使っています。

ヒント: 参考 URL

python-modernize

Automatic conversion to Py2/3

How to port Python 2 Code to Python 3

3.2 2to3 の基本的な使い方

2to3 が変更された文法のどれに対応しているかは、

```
2to3 -l
```

を実行することで確認できます。この文章作成時に実行したところ、52 の変換項目（変換サブプログラムに対応している）が挙げられました。[1]

- apply
- asserts
- basestring
- *buffer*
- dict
- except
- exec
- execfile
- exitfunc
- filter
- funcattrs
- future
- getcwdu
- has_key
- *idioms*
- import
- imports
- imports2
- input
- intern
- isinstance
- itertools
- itertools_imports
- long

[1] 2to3-3.10 でもこの数 (52) は同じでした。

- map
- metaclass
- methodattrs
- ne
- next
- nonzero
- numliterals
- operator
- paren
- print
- raise
- raw_input
- reduce
- reload
- renames
- repr
- *set_literal*
- standarderror
- sys_exc
- throw
- tuple_params
- types
- unicode
- urllib
- *ws_comma*
- xrange
- xreadlines
- zip

これらの変換項目は "`-f`"オプションや"`-x`"オプションを追加することで、明示的に変換項目に有効化／無効化を指定できます。これらの変換項目のいくつか（ *buffer*, *idioms*, *set_lieteral*, *ws_comma*) はオプショナルな変換項目となっており、通常は無効化されています。これらの変換項目による結果は、厳密に等価なプログラムを与える訳では無いことから、オプショナルな変換項目となっているようです。

`-f` を使って変換項目を指定すると、明示的に指定した変換項目だけが有効化されます。既定の変換項目に変換項目を追加するには、

```
2to3 -f all -f <変換項目名> .
```

とします。

3.2.1 変換項目 (変換プログラム)

上にあげた変換項目 (apply,...) の内容は、例えば https://docs.python.org/ja/3/library/2to3.html#fixers に説明されています。

これらの変換プログラムのうち、`imports/imports2` は `python2` から `python3` への移行時にモジュール名が変更されたモジュール (例えば Tkinter -> tkinter) に対応してくれます。imports では、Tkinter, FileDialo, Tix, ttk などの Tk 関係のモジュール、dbm に関連したモジュール, xmlrpc サーバ関係のモジュール、httplib や HTTPServer に関連したモジュールなどに対応しています。また、imports2 では whichdbm, anydbm の使用が dbm モジュールを使うように変更されます。imports と imports2 に分離されているのは、"単に技術的な制約のため" だそうです。

`urllib` については `imports` とは別に変換プログラム ``urllib`` (fix_urllib.py) が用意されています。

`idioms` 変換プログラムは、"Python コードをより Python らしい書き方" にするいくつかの変形を行います。object の型 `type(obj)` と Type オブジェクト `T` との比較を、可能なところでは、`isInstance(obj,T)` に置き換える、`while 1:` を `while True:` に置き換える、可能な場所では `sorted(EXPR)` を使う、等の変換を行います。

3.2.2 Tab とスペース

http://python3porting.com/differences.html#index-14 には次の記載があります。

In Python 2 a tab will be equal to eight spaces as indentation, so you can indent one line with a tab, and the next line with eight spaces. This is confusing if you are using an editor that expands tabs to another number than eight spaces.

In Python 3 a tab is only equal to another tab. This means that each indentation level has to be consistent in its use of tabs and spaces. If you have a file where an indented block sometimes uses spaces and sometimes tabs, you will get the error TabError: inconsistent use of tabs and spaces in indentation.

つまり、python2 では 一つのブロック中で tab と space を混在しても、tab と space の関係 (1 tab = 8 spaces) を使ってブロック内の各行の始まりが一致していれば問題はありませんでした。一方 python3 では、同じブロックの中では、tab とスペースの数と位置が一致している必要があります。

このため、同じブロック内で tab とスペースの数が異なる行があるスクリプトでは、python2 では動作するけれど、python3 では動作しないことになります。

TAB と空白の混在はエディタなどの環境の違いによって、見かけ上の構造と Python インタプリタからみた構造が異なる場合があります。これは python2 においても同様ですので、python2/python3 にかかわらず TAB と空白の混在は避けておくのが得策です。

ちなみに私自身は emacs の Python-mode を使い python プログラムを開発します。此の環境では、キーボードからの TAB 入力は複数の空白に置き換えてくれます。デフォルトは 4 個の空白ですが、これを変更することも可能です (Preference で `Python Indent Offset` を変更する)。

3.2.3 変更部分の確認

`2to3` を出力を指定するオプションをつけずに実行することで、`2to3` が加える変更を diff 形式の出力で確認できます。

```
2to3 *.py
```

あるいは

```
2to3 .
```

で現在にディレクトリにある python コードで必要な変換内容を確認します。必要に応じて、`-f`` あるいは-x``オプションをつけて変更項目の有効化／無効化を試行して見ます。

3.2.4 新しいディレクトリに Pytho3 対応ソースコードを作成する。

`2to3` には様々なオプションが用意されていますが、次のシェルコマンド

```
mkdir ../PY3
2to3 -W -n -o ../PY3 .
```

を知っていれば、最低限の用は足りるでしょう。このコマンドは、python2 のソースコードのディレクトリに現在居るとして、

1. Python3 用の作業ディレクトリ../PY3 を作成し、(mkdir ../PY3)
2. そのディレクトリに現在のディレクトリにある全ての Python ソースコードを `2to3` で必要な場合には、Python3 向けに変換し、(`2to3`)
3. 変換の必要の無いファイルを含めて (-W -n)
4. 新しいディレクトリ (../PY3) に書き出します。(-o ../PY3)
5. **`-n` オプションは、オリジナルの Python2 向けコードを backup として作成しないことを指定しています。**
   `-n` オプションは `-o` オプション を使う際には必須となっています。

`2to3` コマンドの詳細は、

```
2to3 --help
```

で確認できます。

3.3 mercurial などを使って管理している場合

ソースコードを mercurial などの管理ツールを使って管理している場合には、ブランチなどの機能を使って、python3 版の開発を進めることが考えられます。

```
mkdir PY3
cd PY3
hg clone <repository> .
```

によって、現在のバージョンのコピーを作業ディレクトリ (PY3) に作成します。ここで、

```
2to3 -w -n .
```

を実行すると、Python3 への変更が必要なファイルだけが 2to3 によって変更されます。

変更前のファイルをバックアップとして残して置きたい場合には、

```
2to3 -w
```

とします。もっとも、hg で管理しているのであれば、hg diff で見ることができるので、あまり必要な無いように思われます。

3.3.1 setup ツールとの連携

setuptools を使った install script では、use_2to3 オプションを設定することで、”setup.py intall"実行時に自動的に 2to3 を適用してからインストールすることができます。後で見るように、2to3 で自動変換したプログラムが python3 では意図した動作とならない場合があることから、その有用性はあまり無いのかもしれません。 setuptools のドキュメントでも

> Setuptools provides a facility to invoke 2to3 on the code as a part of the build process, by setting the keyword parameter use_2to3 to True, but the Setuptools project strongly recommends instead developing a unified codebase using six, future, or another compatibility library.
>
> —setuptools 45.2.0 documentation

となっています。一般的には、この記述にも触れられている six あるいは future などのモジュールを使うことが推奨されています。必要な変更の数が少ない場合には、有効な方法ですので、ダメもとで一度試して見るだけの価値は有りそうです。

4 2to3 の先に

|2to3| は優れたツールですが、python3 セーフにするためには、もう少し手を加える必要がある場合もあります。特に制御関係ではバイナリデータを取り扱う場合に注意が必要です。

4.1 文字型／バイナリ型データ

Python2 では C 言語からの伝統を受け継いで (?)、文字型データ (str) とバイト型データ (bytes) には本質的な違いはありませんでした。というか、bytes は str の別名となっています。また、2byte 文字を取り扱うために、unicode 型が導入されています。unicode 型と str 型の変換は encode/decode などのメソッドを使って行います。

ところが、Python3 では文字列 (str) は unicode 型の同意語となります。unicode 文字列を表すバイト型データ (bytes) は unicode 文字列を encode することで得られます。また、バイト列を unicode 文字列 (str) に変換するためには、decode メソッドの助けを借りなければいけません。

ソースコード 4.1.1: Python での文字型データ

```
>>> ("あ",u"あ", r"あ",b"\xe3\x81\x82")
("あ",u"あ", r"あ",b"\xe3\x81\x82")
('\xe3\x81\x82', u'\u3042', '\xe3\x81\x82', '\xe3\x81\x82')

>>> s="あいう"
s="あいう"
>>> type(s)
type(s)
<type 'str'>
>>> u=s.decode('utf-8')
u=s.decode('utf-8')
>>> type(u)
type(u)
<type 'unicode'>
>>> u
u
u'\u3042\u3044\u3046'
>>> s
s
'\xe3\x81\x82\xe3\x81\x84\xe3\x81\x86'
>>>
```

ソースコード 4.1.2: Python での文字型データ

```
>>> ("あ",u"あ", r"あ",b"\xe3\x81\x82")
("あ",u"あ", r"あ",b"\xe3\x81\x82")
(' あ', ' あ', ' あ', b'\xe3\x81\x82')
>>>s="あいう"
s="あいう"
>>> type(s)
type(s)
<class 'str'>
>>> b=s.encode('utf-8')
b=s.encode('utf-8')
>>> type(b)
type(b)
<class 'bytes'>
>>> b
b
```

(次のページに続く)

(前のページからの続き)

```
b'\xe3\x81\x82\xe3\x81\x84\xe3\x81\x86'
>>>
```

Vers. types	python2 encode	decode	[0]	python3 encode	decode	[0]
bytes	->str(b)	->u	str	**N/A**	->str(u)	int
str	->str(b)	-> u	str	->b	**N/A**	u
unicode	->str(b)	-> u	u	->b	**N/A**	u

通常の文字型データだけを取り扱っている限りでは、2to3 の変換後のプログラムは問題なく動作するでしょう。しかし Python2 で str 型データを bytes 型データとして取り扱うために使っているプログラムでは、2to3 は自動変換を行うことができません。str 型データとして取り扱うべきか (その場合には Unicode 型=python3 の str 型に変換）,bytes 型データとして取り扱うべきか (その場合には Python3 では明示的に bytes 型を指定する必要がある）を python2 のソースから自動的に判断することが 2to3　にはできないためです。

py2/p3 で両立するためには、bytes と unicode を明示的に使い、bytes -> unicode の時に decode を、unicode-> bytes の変換で encode を使うと決めておけば、2to3 はそれを頼りに変換を行います。2to3 適用前にこれを実施しておけば、あとは何度 2to3 を通しても問題無いはずです。

- 文字列として使う場合は、unicode
- バイト列として使う場合は、bytes

cVXI11 や EPICS CA モジュールでは python2/python3 の双方をサポートするために、この様な変更を加えています。また、cVXI11 や EPICS CA モジュールは setup ツールを使った build 時に 2to3 ツールを自動的に適用する様にしています。

リテラルなバイト列表現 (b"abc"など）は Python2, Python3 の双方で利用可能です。バイト列として取り扱う定数データは、この表現にして置くことで、python2/python3 の双方でほぼ同じ様に取り扱うことができます。バイト列 (byte) から 1 バイトのデータを取り出す際には、注意が必要です。Python2 では [n] オペレータで 1 バイトのデータをバイト列 (＝ str) を取り出すことができます。しかし、python3 では byte 列から [n] オペレータを使って 1 バイトデータを取り出すと、int 型のデータとなってしまいます。一方 [n:n+1] で一バイト分のデータを取り出した場合には、bytes 型になっています。　　この様な動作の違いを吸収するためには、python2 でも 1 バイトのデータを取り出す場合にも [n:n+1] を使っておくという配慮が必要です。

また、python2/python3 とも bytearray 型を持っています。bytearray 型は PY2/PY3 で概ね同じ methods を持っている (例外：copy,clear,maketrans) ので、byte 型ではなく、bytearray 型を積極的に使うのも python2/pytho3 互換とするためには有効と考えられます。

4.1.1 open の encoding

文字型／ bytes 列の取扱で注意すべき事として、open 関数の python2/python3 での違いがあります。

python3 において、文字型データのファイル入出力では、入出力モードがテキストモード ("r"あるいは"w") である場合には、ファイルが期待するエンコーディングへの変換が必要となります。デフォルトのエンコーディングはプラットフォーム依存で, locale.getpreferredencoding(False) の返す値が使われます。 バイナリデータのファイル入出力ではバイナリモード ("rb",'wb"など）を使うことが推奨されています。また、そうしておけば、python2/python3 の双方で想定した通りの動作をするでしょう。

ソースコード 4.1.1.1: python2 での open() 関数のプロトタイプ

```
>>> help(open)
open(...)
          open(name[, mode[, buffering]]) -> file object

#デフォルトのエンコーディング
>>> locale.getpreferredencoding(False)
locale.getpreferredencoding(False)
'US-ASCII'
```

ソースコード 4.1.1.2: python3 での open() 関数のプロトタイプ

```
>>> help(open)
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,␣
↪closefd=True, opener=None)
    Open file and return a stream.  Raise OSError upon failure.

#デフォルトのエンコーディング
>>> locale.getpreferredencoding(False)
locale.getpreferredencoding(False)
'UTF-8'
```

Python2/Python3 の判定

python2/python3 の双方に対応するプログラムを開発するためには、今動作中の環境を判定する必要が出てきます。その場合には次のコードを使うのが良さそうです。

```
import sys

PY3 = sys.version_info > (3,)
```

これで, 変数 PY3 は python2 では False,python3 では、True となっています。 sys.version_info は 2.7 以降は version_info 型のオブジェクトですが、2.6 あるいはそれ以前のバージョンでは、tuple となっています。しかし、上記の式はいずれの環境においても正しく動作します。

5 future モジュール

future モジュール は[1] , python2/python3 の双方で動作するスクリプトの開発を支援するためのモジュールです。このモジュールのホームページ Easy, clean, reliable Python 2/3 compatibility には次の様に記述されています。

[1] : future モジュールと __future__ モジュールは似た名前ですが、全く異なるモジュールです。 __future__ は Python の標準ライブラリの一つで、 将来のバージョンで導入される（された）機能をそれ以前のバージョンでも利用可能とするためのモジュールです。Python3 で導入された機能を Python2 で利用するためにも利用されます。 一方、future モジュールは、 python-future プロジェクトによって保守されている、"Python2 と Python3 の互換性を保つ" のに役立つモジュールです。

ホームページの記述

python-future is the missing compatibility layer between Python 2 and Python 3. It allows you to use a single, clean Python 3.x-compatible codebase to support both Python 2 and Python 3 with minimal overhead.

---

python-future は Python2 と Python3 の間にかけている互換性レイヤを提供します。このモジュールを使うことによって、Python2 と Python3 の双方で最小のオーバーヘッドで実行可能な python プログラムをサポートする、単一の綺麗な, Python3 互換のプログラムの基盤を手に入れます。

future モジュールのホームページジ Easy, clean, reliable Python 2/3 compatibility は python2/python3 で共通に動作するプログラムを書くために役に立つ python2/pytho3 の違いを Cheat Sheet にまとめています。これらの違いのうち多くは 2to3 スクリプトで適切に処理されます。

5.1 インストール

future モジュールは pip を使って簡単にインストールすることができます。

```
python2 -m pip install future
```

RH, SL, CentOS などの OS では yum を使ってインストールすることも可能です。

```
yum install python2-future
```

5.2 futurize

futurize は　future モジュールと一緒にインストールされるコマンドツールです。　future module を使ったシェルレベルのコマンドです。python2 向けに書かれたスクリプトを、python2/3 の双方で利用可能に書き換えることをサポートしてくれます。futurize コマンドは

- python2 のスクリプトを"modern"な スクリプト に変換する stage1(-1/--staget1) と
- future モジュールなどを使って、python2/python3 の双方に対応可能なスクリプトに変換する stage2(-2/--stage2)

のふたつのステージがあります。デフォルトでは、このふたつのステージを連続して行います (-0/--both-stagge)。**futurize** コマンドは内部で 2to3 のライブラリを利用しています。

5.3 pasteurize

pasturize は, futurerize とは逆に、python3 のスクリプト を python3/2 両用のスクリプト に変換します。　futurize と同様に future module を利用しています。

3to2 というモジュールも存在しますが, 2015 年 4 月 16 日のバージョンで更新が止まっている様子です。

future モジュールの最新版は 0.18.3 で Jan. 13, 2023 に更新されています (。Python2 の現在の Version は 2.7.18(Rel. April 20,2020) ですから、ほぼ最新版の Python2 に対応していると思って良い様です。

脚注

6 Six: Python 2 and 3 Compatibility Library

Six は、その名前 six (= 2 × 3) が示しているように、python2 と python3 の両方で動作するプログラムの開発をサポートしてくれるモジュールです。python2/pytho3 互換とする為のマクロやユーティリティ関数を提供してくれます。

six は数多くの 3rd パーティモジュール (例えば matplotlib) で採用されていますが、それ自体も 3rd パーティモジュールの扱いになっています。したがって、貴方のプログラムを six モジュールを使って、python2/python3 互換とした場合、貴方のプログラムを配布する際には、そのユーザには six モジュールを別途インストールする様に注意するか、.egg などの方法で貴方のプログラムと同時に six モジュールがインストールされる様に設定しておく必要があります。

six.py のライセンスは MIT ライセンスとなっていますので、ライセンスの要求する著作権表示の要件などを満たせば、配布パッケージの中にソースコードを含ませることもできるので、大きな問題ではありませんが。

と言うことで、pythonCA, pyVXI11 では `six` モジュールは使わないようにしています。

6.1 おまけ

https://docs.python.org/ja/3/howto/pyporting.html

には、互換性を保ちながら、pytho2 から python3 への移植を行う際の手順が説明されています。此の中では、`futurize` や `Modernize` と言ったツールが紹介されています。

7 Cython での注意点

これまでに CPython2/3 について述べた事について注意を払えば、Cython のソースコード (.pyx) は python2/python3 の双方で使うことができます。

Cython はソースコード (.pyx) から C あるいは CPP のソースコードを作成します。 注意すべきは、同じ Cython のソースコード (.pyx) から python2/Cython で作成された c/cpp コードと python3/Cython で作った c/cpp コードには微秒な違いがあります。Python2/Cython および Python3/Cython の双方で Extension を作成する場合、同じ pyx ファイル名を使って python2/pytho ん 3 の双方でそれぞれ Extension を作成する場合、ビルドの前に cython が生成した C ／ C ＋＋のソースコードを消去しておく必要があります。 私が作成した Python_CA、および cVXI11 モジュールでは、Cython ソースコード (cVXI11.pyx など) のリンク (cVXI11-2.pyx および cVXI11-3.pyx) を作成しておき、setup.py 中では、

```
# cVXI11-2.pyx and cVXI11-3.pyx are hard links to cVXI11.pyx
cVXI11_source_PY2="cVXI11-2.pyx"
cVXI11_source_PY3="cVXI11-3.pyx"

if sys.version_info >= (3,):
     cVXI11_source=cVXI11_source_PY3
else:
     cVXI11_source=cVXI11_source_PY2
if not os.path.exists(cVXI11_source):
     os.link("cVXI11.pyx", cVXI11_source)
```

などとしています。これにより、同じ pyx ソースコードで、python2/python3 では、それぞれ cVXI11-2.cpp, cVXI11-3.cpp を cython が作成し、これを C++がコンパイルする事になります。

8 py2app/py2exe

py2app と py2exe はそれぞれ、macitosh OS および Windows で Python で作成されたプログラムをそれぞれのプラットフォームでのアプリケーションとしてまとめ上げてくれるツールです。

py2app/py2exe では python2/python3 双方に対応するコードを用意する必要はないでしょう。(パッケージ化されているので、利用者はその違いを意識することはないはずです。) が、python2/pytho3 両対応にすることはそれほど難しくはありません。

他の python ベースのツールと同じく、いずれは py2app/py2exe は python3 でのみ動作するようになるでしょう。(py2exe では Windows のバージョンによっては python3 だけに対応という事になっているようです。)

py2app では、python2/python3 の違いの問題とは別に、Tkinter の GUI を使ったアプリケーションで原因不明の現象[1]が起きることがありますので、ご注意ください。

A その他諸々

A.1 zip,enumerate, range and other iterator/generaotor

python の for 文は 反復可能なオブジェクト (iterable oject) に対して続くスイートの中身を実行します。iterable オブジェクトとしてはリスト (list) やタプル (tuple) がよく知られているが、その他にもイテラブルなオブジェクトが存在する。python3 では、`zip`, `range`, `enumerate`, `map` はクラスとなりその呼び出しはイテラブルなオブジェクトを返すようになりました。python2 では `zip`, `enumerate`, `map()` はリストを返していましたので、結果のリストが必要な場所では、これらの呼び出しを `list()` でラップしておく必要があります。

python3 の `itertools` には、for 文でよく使われるパターンを簡略するためにさまざまなイテラブルなオブジェクトを生成するためのメソッドが提供されています。

A.1.1 `iterator.count()` の使い方

例えば、ある条件を満たすまで、繰り返しを行いつつ、実行回数の把握が必要な場合には、

```
i=0
while True:
  do_something()
  if check_condition(i):
    break
  i +=1
```

のようなパターンがよく使われていました。このパターンを itertools の `count()` を使って書き換えると

[1] Tk のウィンドウを表示すると、突如セッションがログアウトされ、ログイン画面に戻ってしまう。同じ Python プログラムをコマンドラインで起動した場合には此の現象は起きず、py2app で作成したアプリケーションを起動した時のみ、此の現象が起きる。

```
import itertools

for i in itertools.count():
  do_something()
  if check_condition(i):
    break
```

と見易い形に書き換えることができます。

A.2 functools module and cmp_to_key() method

リストのソートメソッドの使い方は python2 から python3 で大きく変わったことのひとつです。

python2 では list の `sort()` のプロトタイプは、

sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
    cmp(x, y) -> -1, 0, 1

となっています。通常は文字、数字などの自然な並びに従って整列させる場合には、とくに引数は必要ありませんが、特別なデータ型や、標準的でない順番に整列させ r たい場合には、cmp 引数に、ふたつの要素を比較するための関数を指定します。この関数はふたつの引数の大小に従って、-1,0,1 のいずれかを返すことを期待されています。(python2.4 以降では python3 と同じく、key 関数を使うこともできますが、それはまた別のお話。)

例えば、

```
1 s=list("python")
2 s.sort(lambda x,y: 0 if ord(x) == ord(y) else 1 if ord(x) > ord(y) else -1)
3 print(s)
```

を python2 で実行すると

```
['h', 'n', 'o', 'p', 't', 'y']
```

が出力されます。

一方、python3 では `sort()` のインタフェースは

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

となっています。ソート順を制御するための cmp に代えて、key と呼ばれる関数を用意することになっています。この key() 関数は、ソート対象の各要素毎に評価され、その結果に基づいてソートが実行されます。

click to download python script

click to download jupyter notebook

この cmp と key の違いを吸収するための関数 cmp_to_key() がモジュール functools の中に用意されています。これを使って、先ほどの python2 でのソートを実行するには、次の様に書き換えます。

```
import functools
s=list("python")
s.sort(key=functools.cmp_to_key(lambda x,y: 0 if ord(x) == ord(y) else 1 if ord(x) > ord(y) else -1))
print(s)
```

```
['h', 'n', 'o', 'p', 't', 'y']
```

この様に、python2 で使っていた cmp() 関数を、pytyon3 で動作するようにすることができます。

click to download python script

click to download jupyter notebook

ちなみに、python3 での sort() を key パラメータを使って、同様のソートを行うには

```
s=list("python")
s.sort(key=lambda x:ord(x))
print(s)
```

```
['h', 'n', 'o', 'p', 't', 'y']
```

とすれば十分です。　この様に、cmp() から key() への切り替えは、それほど困難なことではありません。　cmp_to_key() はあくまで、python2 で使っていた cmp() 関数をどうしてもそのまま使いたい場合（例えば大量のソースコードを python2 から python3 に機械的に変換したい時など）に限定しておくのが良さそうです。

A.3 reload 問題

Python2 と Python3 では、reload の取り扱いが変わっています。Python2 では reload 関数は組み込み関数で、global なネームスペースに存在していますが、Python3 においては、reload 関数は、importlib モジュールの中で定義されています。アプリケーションの中で、モジュールのリロードを行うことはほとんど考えられないので、問題になることは無いかと思いますが、プログラム開発中にはしばしば　修正済みのモジュールを reload することになるでしょう。

python3 では、 reload 関数の利用に先立ち、

ソースコード 1.3.1: Python3 での reload 関数の利用

```
>>> from importlib import reload
reload
<function reload at 0x10275e158>
```

として reload 関数を import します。この後、

ソースコード 1.3.2: mymodule をリロードする。

```
reload(mymodule)
```

としてやることで、python2 と同じく、モジュールを reload してやることができます。

A.4 print 問題

print は python2 では **文**, python3 では **関数** となっています。2to3 はこの違いを適切に取り扱って、プログラムを更新してくれます。ただし、この場合変更後のコードは python2 では動作しなくなります。

このため、同じコードベースを python2/python3 のどちらでも動作させるには、ちょっと工夫が必要です。

A.4.1 print() 関数

まずは、python2 では__future__モジュールを使って python2 でも python3 の print 関数が使えるようにすることができます。

```
from __future__ import print_function
```

__future__ モジュールのインポートはファイル中の最初の実行文である必要があるので、注意してください。

A.4.2 sys.stdout/stderr

もともと、print 文は、標準出力に指定された文字列を出力します。標準出力および標準エラー出力は sys モジュールの sys.stdout および stderr としてアクセス可能ですから、print 文を使う代わりに、sys.stdout あるいは sys.sterr に文字列を送り出すことで、print 文を使用しないようにもできます。

```
import sys

sys.stdout.write("Hello World")
```

A.4.3 logging モジュール

そもそも、デバグメッセージを表示するために print 文を使っているなら、logging などのモジュールを使いメッセージを表示させるのが良いでしょう。logging を使えば、開発のフェーズ毎に"Debug", "Info","Warnig"などのメッセージレベルを使って出力されるメッセージを制御することができます。 使い方は、次の:py:mod:*logging* モジュールの使用例をご覧ください。

```
import logging
logging.getLogger().setLevel(logging.WARNING)
...

logging.warning("これは警告です。")
```

出力例：

WARNING:root: これは警告です。

logging モジュールには、 {'DEBUG':10, 'INFO':20, 'WARNING':30, 'ERROR':40, 'CRITICAL':50, 'FATAL':50} のレベルが定義されています。また、このレベルに対応した出力関数 `debuf, info, warning, error, critical, fatal()` が提供されています。実行時の設定レベル以上のメッセージが出力されます。 メッセージの出力先は、端末だけでなく、ファイル、syslog などに変更／追加することも可能です。

A.5 exceptions モジュール

python3 では `exceptions` モジュールは廃止されています。python2 で `exceptions` モジュール経由で呼び出す Exception のクラスは全て `__builtins__` の中に入っていますので、module を `import` する必要は有りません。`exceptions.xxxException` の様な使い方をしていた場合には、直接 `xxxException` を使う様にすれば良いかと思われます。

A.6 maxint

python3 では取り扱える整数の最大値が **存在しません** 。従って、 `sys.maxint` も廃止されています。取り扱える上限を決めているものが何かによって、対応が変わるかと思いますが、`sys.maxint` ではない別のものにしておくべきでしょう[1] 。

A.7 time.mktime の 2000 年問題 (2023/12/15 追記)

```
>>> time.mktime((2023,12,15,0,0,0,0,0,0))
1702566000.0
>>> time.mktime((23,12,15,0,0,0,0,0,0))
time.mktime((23,12,15,0,0,0,0,0,0))
1702566000.0
```

となりますが、python3.6 では、

```
>>> time.mktime((2023,12,15,0,0,0,0,0,0))
1702566000.0
```

(次のページに続く)

[1] (`maxint`, exceptions の問題は plex を python3 に対応させるために必要でした。)

```
>>> time.mktime((23,12,15,0,0,0,0,0,0))
-61411339139.0
```

python3.9 では、

```
>>> time.mktime((2023,12,15,0,0,0,0,0,0))
1702566000.0
>>> time.mktime((23,12,15,0,0,0,0,0,0))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OverflowError: mktime argument out of range
```

となっています。ともあれ、 time.mktime() で二桁の年号をお使いの場合は、気をつけましょう。python3.9 では, mktime() の year の引数が 1900 より小さい場合には、 OverFlowError を返すようです。

```
>>> time.mktime ((1900, 1, 1,0,0,0,0,0,0 ))
-2209021200.0
```

なので、単に「Unix 時間が負になる」ことは問題では無いようにも見えます。