Maya® 2022 における Python 3 と 2 の相互運用のノウハウ

株式会社スクウェア・エニックス
テクノロジー推進部
リードテクニカルアーティスト

佐々木　隆典

SQUARE ENIX　ADVANCED TECHNOLOGY DIVISION

アジェンダ

- Maya® と Python
- Python 3 対応の戦略
- Python 2 と 3 の違い
  - __future__ モジュールに関すること
- コード書き換え手順
- Python 2 と 3 の違い
  - 移動や廃止
  - 様々なこと
  - クラス関連
  - 文字列とIO
- Python C API について
- まとめ

Maya®とPython

Pythonの組み込み状況

Python 2 から 3 へ

2021年1月23日	pip が Python 2.7 サポート終了
2020年10月5日	Python 3.9 リリース
2020年4月30日	Python 2.7.18 リリース（最後の 2.7.x）
2020年1月1日	Python 2.7 サポート終了
2018年6月27日	Python 3.7 リリース（後に Maya®2022 で採用）
2010年7月4日	Python 2.7 リリース
2008年12月3日	Python 3.0 リリース
2008年10月1日	Python 2.6 リリース

VFX Reference Platform

CG界隈では Python 3 化が遅れていたが、
VFX Reference Platform が CY2020 から Python 3.7.x を採用。

Maya®もそれに合わせている。

	linux gcc	Python	Qt	対応 Maya®
CY2021	9.3.1	3.7.x	5.15.x	2022
CY2020	6.3.1	3.7.x	5.12.x	
CY2019	6.3.1	2.7.9 – 2.7.latest	5.12.x	2020
CY2018	6.3.1	2.7.5 – 2.7.latest	5.6.1 – 5.6.latest	2019
CY2017	4.8.2 – 4.8.3	2.7.5 – 2.7.latest	5.6.1	2018

Retrieved from https://vfxplatform.com/

Maya®バージョンの変遷

Maya®	Python	Qt
2022	3.7.7 / 2.7.11	5.15.2 (PySide2 5.15.2)
2020	2.7.11	5.12.5 (PySide2 5.12.5)
2017 – 2019	2.7.11	5.6.1 (PySide2 alpha0)
2016	2.7.6	4.8.6 (PySide 1.2.0)
2015	2.7.3	4.8.5 (PySide 1.2.0)
2014	2.7.3	4.8.2 (PySide 1.1.1)
2012 – 2013	2.6.4	4.7.1
2011	2.6.4	4.5.3
2010	2.6.1	n/a
2008 – 2009	2.5.1	n/a
8.5	2.4.3	n/a

Maya® 2022 の Python

- Python 3 と 2 の両搭載
  - 標準は Python 3 だが、経過的措置として Python 2 も搭載している。
  - ただし、Mac® 版は Python 3 のみ搭載。
- 起動時に選択する方式
  - コマンドラインオプション(優先)： –pythonver 2
  - 環境変数： MAYA_PYTHON_VERSION=2
  - Maya.env ファイルでの設定は不可。
- 積極的に Python 2 モードを選択すべき理由はない
  - 過去リソースを 3 に対応できないが 2022 を使いたい場合などに。

Maya® 2022 のディレクトリ構造

Python27, Python37

Python モジュールは、2用と3用に完全に分かれている。

bin, bin2, bin3

exe や dll のパスは、標準（共通か3用）と、2専用、3専用に分かれている。
mayapy は bin 下に mayapy（3用）と mayapy2（2用）がある。

lib, lib2, lib3

SDK等のライブラリパスは、標準（共通か3用）と、2専用、3専用に分かれている。
C++プラグインでもPython依存がある場合は、2用と3用をビルドする必要がある。

include, include/Python27, include/Python37

SDK等のヘッダーパスは共通だが、Python CAPI は完全に分かれている。

Python 3 対応の戦略

これまでのコードをどのように移行し運用を継続するのか

状況によって戦略は異なる

- 今後 py2 を使うかどうか？
- py2用とpy3用のダブルメンテナンス？
- 過去のコードはどのくらいあるのか？
- テストがきちんと書かれているか？
- これまでどの程度準備してきたのか？
- どのような移行ツールをどこまで使うのか？
- どのような互換レイヤーを用いるのか？

py2 と py3 の違いを吸収する
補助的なモジュールのこと

共通戦略

- 今後 py2 を使うかどうか？ ← 使う前提の話
- py2用とpy3用のダブルメンテナンス？ ← 回避は難しくない
- 過去のコードはどのくらいあるのか？
- テストがきちんと書かれているか？
- これまでどの程度準備してきたのか？
- どのような移行ツールをどこまで使うのか？
- どのような互換レイヤーを用いるのか？

私の場合（当初）

- 今後 py2 を使うかどうか？ ← 使う前提の話
- py2用とpy3用のダブルメンテナンス？ ← 回避は難しくない
- 過去のコードはどのくらいあるのか？ ← 大量
- テストがきちんと書かれているか？ ← あまり整備されていない
- これまでどの程度準備してきたのか？ ← そこそこ意識していた
- どのような移行ツールをどこまで使うのか？ ← 手で書き換え
- どのような互換レイヤーを用いるのか？ ← 自作

自動変換は怖い？！

かなり大変と途中で思い知る

私の場合（最終）

- 今後 py2 を使うかどうか？ ← 使う前提の話
- py2用とpy3用のダブルメンテナンス？ ← 回避は難しくない
- 過去のコードはどのくらいあるのか？ ← 大量
- テストがきちんと書かれているか？ ← あまり整備されていない
- これまでどの程度準備してきたのか？ ← そこそこ意識していた
- どのような移行ツールをどこまで使うのか？ ← 手で書き換え / ツールもかなり使用
- どのような互換レイヤーを用いるのか？ ← 自作 / 少し引用

過去のコードがよほど少量でなければツールは使うべき

状況が異なれば、別の戦略も有り得る

- 今後 py2 を使うかどうか？ ← 使う前提の話
- py2用とpy3用のダブルメンテナンス？ ← 回避は難しくない
- 過去のコードはどのくらいあるのか？ ← 大量
- テストがきちんと書かれているか？ ← きちんと整備されている
- これまでどの程度準備してきたのか？ ← あまり意識していなかった
- どのような移行ツールをどこまで使うのか？ ← ツールで書き換え / 手でも修正
- どのような互換レイヤーを用いるのか？ ← ツールを利用

ただしツールは万能ではないので手での書き換えも必要

ツールの紹介

2to3 Python 本体に同梱

- py2 のコードを py3 用に自動変換するツール。Python 本体に含まれる。
- 一方通行なので使用しないが、コアの lib2to3 モジュールは他の変換ツールの基盤となっている。

six Maya®2022に同梱: py3=1.12.0, py2=1.10.0

- 互換レイヤーを提供するモジュール。Python 2.4 以上をサポート。

python-modernize

- py2 のコードを 3 と 2 の両方で動くように自動変換するツール。
- 互換レイヤーには six を利用。

python-future Maya®2022に同梱: 0.18.2 おすすめ!!

- py2 や py3 のコードを 3 と 2 の両方で動くように自動変換するツール。
- py2 を py3 のようにしてしまう強力な互換レイヤーを提供。Python 2.6 以上をサポート。

pylint おすすめ!!

- 有名な静的解析ツールだが、コードが py3 で動作が問題ないかをチェックする機能も持つ。

重要なこと

- ツールを使うことで、退屈だが注意を要する単純な書き換え作業の大部分を自動化できる。ミス防止にもつながる。
- Python は動的型付け言語なので、静的解析に基づくツールでできることには限度がある。実際、単純な書き換えでは済まないことも結構ある。
- ツールでできないことを補うのは知識。とはいえ、ツールでチェックできるような py3 と py2 の細かな違いを網羅する必要はない。より困難な問題を克服する知識が必要。
- 本セミナーでは、私が行ったやり方を「おすすめ」として紹介するが、それが絶対的な正解というわけではない。

Python 2 と 3 の違い

__future__ モジュールに関すること

__future__ モジュールとは

- コードの先頭で import することで、そのコード内のみ、pythonインタプリタの一部の挙動が将来のものになる。
- py2 を py3 の挙動にするものは４つ。
  - absolute_import
  - division
  - print_function
  - unicode_literals
- それらの import は、py3 では冗長なだけで何も作用しない。標準で備えている互換レイヤーといえる。

絶対インポートと相対インポート

import には、絶対インポートと相対インポートがある。

ディレクトリ構造：

```
bbb.py
pkg/
  __init__.py
  aaa.py
  bbb.py
```

同名

pkg/aaa.py

```
# 絶対インポート: トップレベルの bbb をインポートしたい
import bbb
print(bbb.__name__)

# 相対インポート: パッケージ内の bbb をインポートしたい
from . import bbb as _bbb
print(_bbb.__name__)
```

しかし、py2 では絶対表記の場合でもパッケージ内が優先されてしまい、トップレベルと同名のモジュールがパッケージ内にあると、インポート出来ない事態に陥る。

py2 の場合：

```
>>> import pkg.aaa
pkg.bbb
pkg.bbb
```

困る

py3 の場合：

```
>>> import pkg.aaa
bbb
pkg.bbb
```

正しい

仮に、トップレベルに bbb が無ければ py3 ではエラーになる。

__future__.absolute_import

absolute_import をインポートすると、py2 のパッケージ内を優先する挙動がなくなり、py3 と同じように、絶対表記であれば正しい絶対インポートになる。

pkg/aaa.py

```
from __future__ import absolute_import

# 絶対インポート: トップレベルの bbb をインポートしたい
import bbb
print(bbb.__name__)

# 相対インポート: パッケージ内の bbb をインポートしたい
from . import bbb as _bbb
print(_bbb.__name__)
```

py2 でも、相対インポートをしたいなら常に相対表記にするのはもちろんのこと、absolute_import も常にインポートしておくと間違いない。

__future__.division

- py2 では、整数で整数を割ると、小数部は切り捨てられた整数になる。
  この演算は floordiv と呼ばれる。
- py3 では、常に完全な割り算となる。この演算は truediv と呼ばれる。
  division をインポートすると、 py2 でも py3 と同じ挙動になる。
- ちなみに、py2 でも py3 でも、// 演算子を用いると floordiv を明示できる。

py2 の場合：

```
>>> 7 / 2.0
3.5
>>> 7 / 2
3
>>> 7 // 2
3
>>> 7 // 2.0
3.0
```

py3 の場合：

```
>>> 7 / 2.0
3.5
>>> 7 / 2
3.5
>>> 7 // 2
3
>>> 7 // 2.0
3.0
```

```
>>> from __future__ import division
>>> 7 / 2.0
3.5
>>> 7 / 2
3.5
>>> 7 // 2
3
>>> 7 // 2.0
3.0
```

__future__.division の弊害

- division をインポートすると、オブジェクトを割り算したときに呼ばれる特殊メソッドが __div__ から __truediv__ に変わる。
- __truediv__ に未対応のクラスの割り算はエラーになってしまう。
- API 2.0 の MPoint と MVector がそれにあたる。
- API 1.0 は、Maya®2018 以降なら対応している。
- 次のように、掛け算に変更するなどして対応する必要がある。

```
>>> from __future__ import division
>>> import maya.api.OpenMaya as api
>>> d = 2
>>> api.MVector(1, 2, 3) / d
TypeError
```

```
>>> from __future__ import division
>>> import maya.api.OpenMaya as api
>>> d = 2
>>> api.MVector(1, 2, 3) * (1. / d)
maya.api.OpenMaya.MVector(0.5, 1, 1.5)
```

__future__.print_function

- print は、py2 では文、py3 では関数。
- py2の特殊な文法を使わなければ、常に括弧をつけておくことで py3 でも動くが、同じ意味ではない。

py2 の場合：

```
>>> print 1, 2, 3
1 2 3
>>> print(1, 2, 3)
(1, 2, 3)
>>> print >>sys.stderr, 'hey!'
hey!
```

py3 の場合：

```
>>> print(1, 2, 3)
1 2 3
>>> print((1, 2, 3))
(1, 2, 3)
>>> print('hey!', file=sys.stderr)
hey!
```

- print_function をインポートすると、py2 でも print が関数になる。

特に弊害もないため、常にインポートするのが良い。

__future__.unicode_literals

- 「ネイティブ文字列」は、py2 では「バイト列」、py3 では「ユニコード」。
- unicode_literals をインポートすると、接頭辞 u の無い文字列リテラルも py3 と同じようにユニコードになる。
- py3 に軸足を置いた開発において、py2 もなるべく py3 と同じ挙動であるべきとの理由から使うのも有り…だとは思う。
- 賛否が分かれると思うが、私は以下の理由により使用しないことを選択した。
  - 必要性がなかった（使わないとまずい場面が無かった）。
  - リテラルとネイティブ文字列の不一致を問題視した（使うとまずい場面が有った）。
  - ツールでも積極的なサポートがされていなかった。
  - 接頭辞 u は py3 では無意味なため一時廃止されたが 3.3 で復活し、価値があるものとみなされるようになった。(PEP 414 -- Explicit Unicode Literal for Python 3.3)

py2 と py3 における文字列データ型

- py2 でも py3 でも、「バイト列」と「ユニコード」というデータ型があり、それに割り当てる「型名」が変わったということ。
- 型名 str が割り当てられている方を「ネイティブ文字列」という。(PEP 3333)
- 文字列リテラルに接頭辞を付けることで、データ型を明示することができる。
  接頭辞無しの場合は「ネイティブ文字列」になるのが通常。

	Python 2 の型名	Python 3 の型名	リテラル接頭辞
バイト列	str	bytes	b
ユニコード	unicode	str	u
ネイティブ文字列	str	str	なし

__future__.unicode_literals の弊害

主に str（ネイティブ文字列）で動作するように設計された各種 API は多く、Python の多くの識別子（オブジェクト名、オブジェクト属性名、キーワード引数など）も str である。

たとえば、次のコードは py2 だとエラーになり、接頭辞 b を指定すると py3 でエラーになるので str() で囲って対応する必要がある。

```
from __future__ import unicode_literals

from types import ModuleType
print(ModuleType('foo'))

print(type('Foo', (object,), {}))

def func(*a, **k):
    print(a, k)
d = {'foo': 1, 'bar': 2, 'baz': 3}
func(**d)
```

エラー

2.6 だとエラー

```
from __future__ import unicode_literals

from types import ModuleType
print(ModuleType(str('foo')))

print(type(str('Foo'), (object,), {}))

def func(*a, **k):
    print(a, k)
d = {str('foo'): 1, str('bar'): 2, str('baz'): 3}
func(**d)
```

コード書き換え手順

python-future について

変換ツール

本セミナーでは
これを使用する

futurize

py2 コードを py2 でも py3 でも動く形に変換する。

pasteurize

py3 コードを py2 でも py3 でも動く形に変換する。

互換レイヤー

本セミナーでは使用しないが
要所要所で参考になる

バックポート (future モジュール等)

py3 の機能やふるまいを py2 でも利用可能にする。

フォワードポート (past モジュール等)

py2 の機能やふるまいを py3 でも利用可能にする。
futurize において、py2 のふるまいを維持しなければならない場面で使われるものなので、積極的に使う必要はない。

python-future による互換レイヤー

py2 でも「py3 のような挙動」にしてしまうことを基本とする。

```
from __future__ import (absolute_import, division,
                        print_function, unicode_literals)
from builtins import (
         bytes, dict, int, list, object, range, str,
         ascii, chr, hex, input, next, oct, open,
         pow, round, super,
         filter, map, zip)
```

当然、全部インポートが基本

py3 と同じ挙動をするビルトイン型や関数を bultins で提供

本来の py2 のビルトインは __builtins__

py3 で整理、再配置されたモジュール類を別名として提供。

```
>>> # Top-level packages with Py3 names provided on Py2:
>>> import html.parser
>>> import queue
>>> import tkinter.dialog
>>> import xmlrpc.client
>>> # etc.

>>> # Aliases provided for extensions to existing Py2 module names:
>>> from future.standard_library import install_aliases
>>> install_aliases()
>>> from collections import Counter, OrderedDict   # backported to Py2.6
>>> from collections import UserDict, UserList, UserString
>>> import urllib.request
>>> from itertools import filterfalse, zip_longest
>>> from subprocess import getoutput, getstatusoutput
```

python-future をインストールするとトップレベルにこれらも配置される

トップレベル以外のものは、install_alias を実行すると配置される

自作のシンプルな互換レイヤーの例

```python
# -*- coding: utf-8 -*-
u"""
シンプルな py2/3 互換レイヤー
"""
import sys as _sys
import re as _re

if _sys.hexversion < 0x3000000:
    BYTES = str
    UNICODE = unicode
    BASESTR = basestring
    LONG = long

    RePattern = _re._pattern_type

    lrange = range
    xrange = xrange

    lzip = zip
    lmap = map
    lfilter = filter
    from itertools import izip, imap, ifilter
    if _sys.hexversion >= 0x2060000:
        from itertools import izip_longest, ifilterfalse

    dict_get_keys = lambda d: d.keys()
    dict_get_values = lambda d: d.values()
    dict_get_items = lambda d: d.items()

    dict_iterkeys = lambda d: d.iterkeys()
    dict_itervalues = lambda d: d.itervalues()
    dict_iteritems = lambda d: d.iteritems()

    if hasattr(dict, 'viewkeys'):  # 2.7 以降
        dict_keys = lambda d: d.viewkeys()
        dict_values = lambda d: d.viewvalues()
        dict_items = lambda d: d.viewitems()
    else:
        dict_keys = None
        dict_values = None
        dict_items = None

    execfile = execfile  # ビルトインを同名でコピー（このモジュールで提供できる）

    fround = round

    def round(f, ndigits=None):
        return int(fround(f)) if ndigits is None else fround(f, ndigits)
```

```python
else:
    BYTES = bytes
    UNICODE = str
    BASESTR = str
    LONG = int

    RePattern = _re.Pattern

    lrange = lambda *a: list(range(*a))
    xrange = range

    lzip = lambda *a: list(zip(*a))
    lmap = lambda *a: list(map(*a))
    lfilter = lambda *a: list(filter(*a))
    izip = zip
    imap = map
    ifilter = filter
    from itertools import (
        zip_longest as izip_longest,
        filterfalse as ifilterfalse,
    )

    dict_get_keys = lambda d: list(d)
    dict_get_values = lambda d: list(d.values())
    dict_get_items = lambda d: list(d.items())

    dict_iterkeys = lambda d: iter(d)
    dict_itervalues = lambda d: iter(d.values())
    dict_iteritems = lambda d: iter(d.items())

    dict_keys = lambda d: d.keys()
    dict_values = lambda d: d.values()
    dict_items = lambda d: d.items()

    def execfile(fname, globals=None, locals=None):
        if globals is None:
            globals = {'__name__': '__main__'}
        exec(compile(open(fname, 'rb').read(), fname, 'exec'), globals, locals)

    def fround(f, ndigits=0):
        return round(f, ndigits or 0)

    round = round  # ビルトインを同名でコピー（このモジュールで提供できる）

MAXINT32 = int(2**31 - 1)  #: 32bit符号付き整数の最大値。Maya の sys.maxint はこれ（py3 で廃止）
MAXINT64 = int(2**63 - 1)  #: 64bit符号付き整数の最大値。Maya の sys.maxsize はこれ
```

futurize のステージ

futurize の変換操作は、2つのステージに分けられている。

Stage 1

本セミナーでは こちらのみ利用

- future や past モジュールを使用せずに、標準的な書き方で可能な範囲で、安全に py2/3 互換コードに書き換える。安心！便利！！

Stage 2

- future や past モジュールを駆使し、処理内容自体は維持しつつも、py2 を可能な限り py3 のような振る舞いに変更しつつ、py3 スタイルの py2/3 互換コードに書き換える。かなり過激な書き換えがされる。ちょっと怖い？
- ツールに完全に依存する戦略でいくなら使うと良い。
  そうでない場合は、自分できちんと対応した部分も変に書き換えられてしまうなどの問題もあり、お勧めはできない。
- 問題箇所を確認したり、互換レイヤーの手法を学ぶ手段としては使える。

pylint について

- --py3k オプションによって、py3 で問題になる部分をチェックできる。
- なるべく、大量の単純な書き換え作業は futurize にやらせて、それで対応できない箇所を検出するツールとして使える。
- futurize の stage 1 はもちろん、stage 2 でも出来ないことも、これで結構チェックできる。
- チェックしながら修正していくことで効率よく書き換えができ、当然ながら python-future の互換レイヤーに頼るよりも良いコードになる。

環境構築

- python-future は Maya®2022 にバンドルされているが、何故か変換ツールが動作しない。別途、作業環境のスタンドアロン Python を用意して、そちらにインストールする。
- 互換レイヤーを利用する場合は、Maya®2022 にバンドルされているものを利用できるが、他のバージョンの Maya® でも利用できるよう整備する必要がある。
- pylint も、同じ環境にインストールする。

コマンドプロンプトで virtualenv にインストールする例：

```
> py -m pip install virtualenv
> py -m virtualenv .¥work
> .¥work¥Scripts¥activate
(work)> pip install future
(work)> pip install pylint
```

他にあると良いもの

バージョン管理システムと GUI クライアント

- ツールによる書き換え差分のチェックに使える。
- 書き換えフェーズごとにコミットして、間違えても戻れるように。
- 私は TortoiseGit を使用。

ディレクトリ階層を grep できる使い慣れたテキストエディタ

- できれば正規表現を使えるもの（できなくても可）。
- ツールで対応できないものをいっきに検出。
- 私は長年使っている某エディタを使用。

テストコード

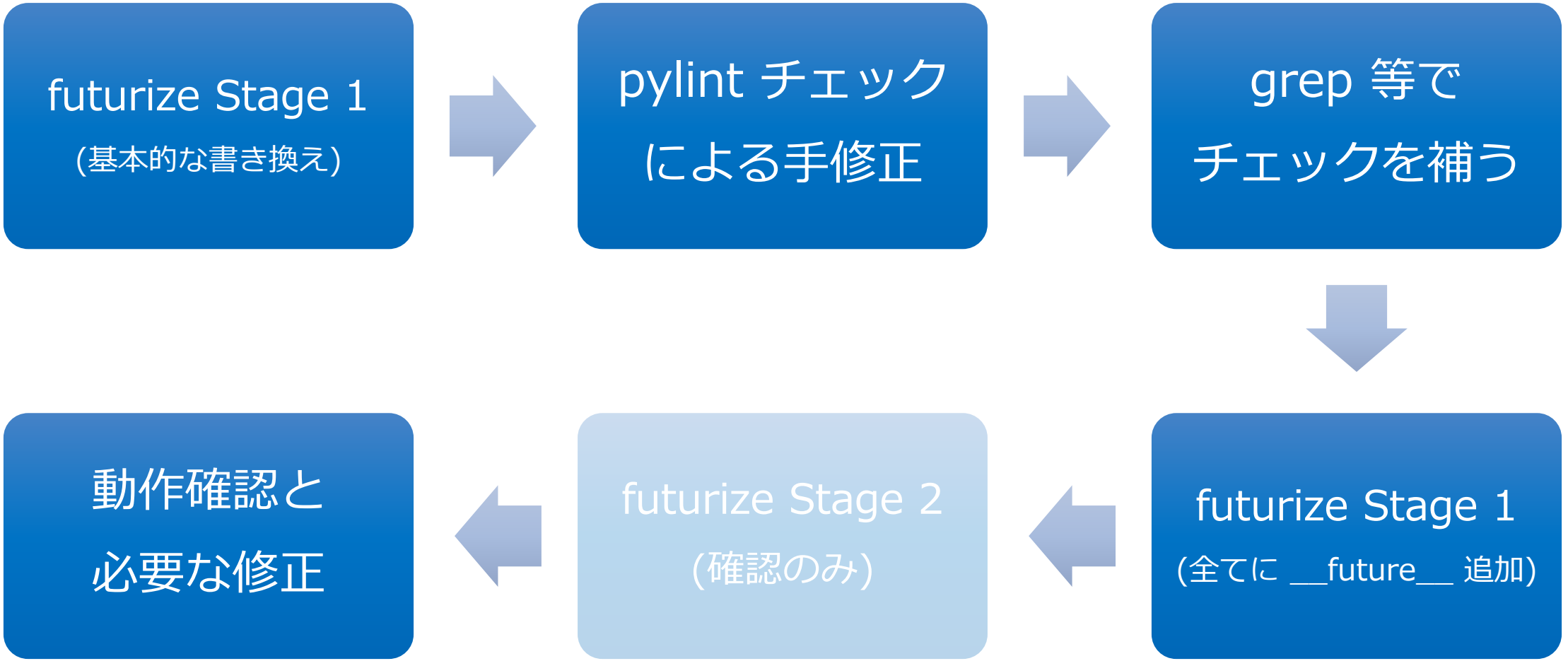
書き換え後の動作チェックに必要。整備されていなければ、少なくとも、全ての import がうまくいくかくらいはチェックできるようにする。

import テストスクリプトの例

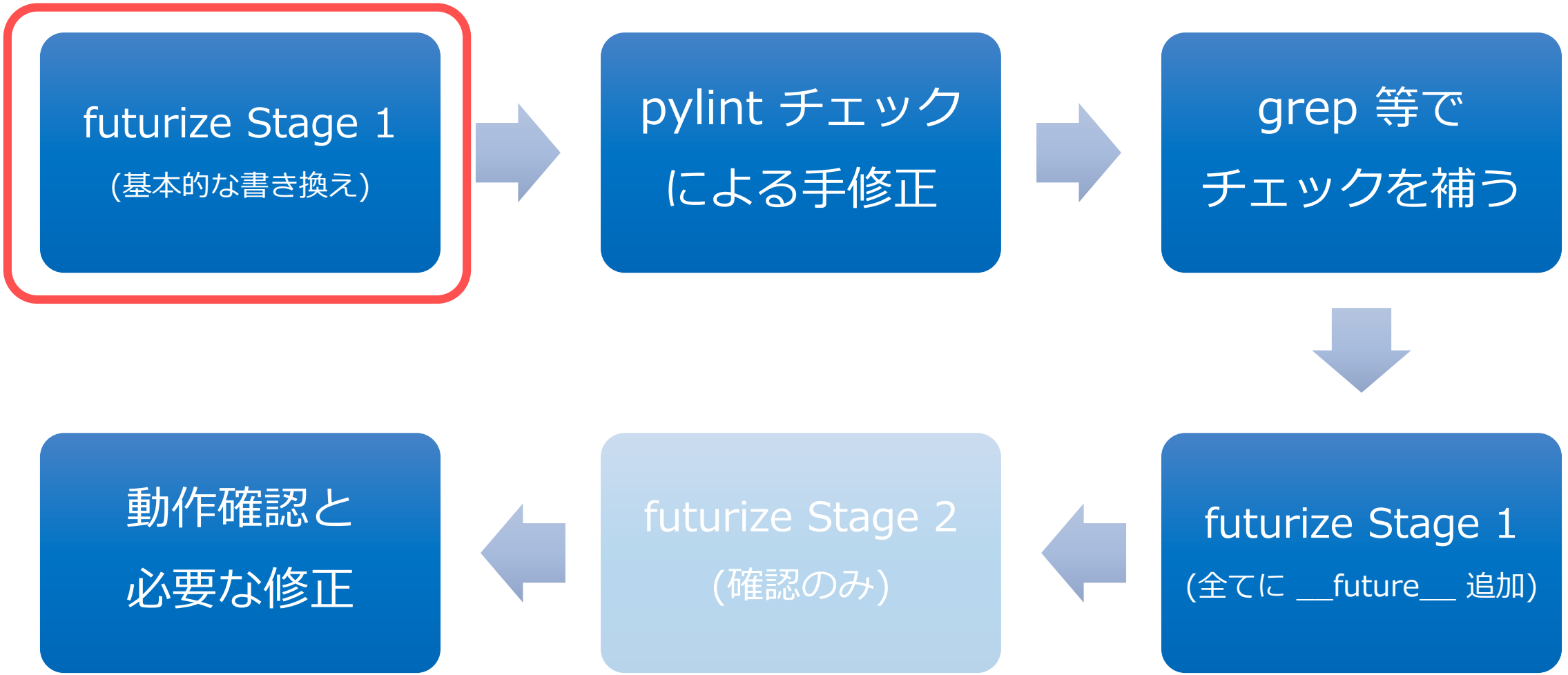
```python
# -*- coding: utf-8 -*-
u"""
指定した名前のパッケージ下の全モジュールのインポートを試す。
__path__ の拡張には対応。
py3 の namespace package には対応しない。
"""
import sys
import os
try:
    from imp import importlib
except:
    import importlib
import_module = importlib.import_module


def removeAll(name):
    name_ = name + '.'
    for x in list(sys.modules):
        if x == name or x.startswith(name_):
            del sys.modules[x]


def doit(name, exclist=tuple()):
    def recursiveImport(name):
        if name in exclist:
            return

        count[0] += 1
        try:
            mod = import_module(name)
            count[1] += 1
        except:
            print(name + ' : FAILURE')
            count[2] += 1
            return

        if os.path.basename(mod.__file__) != '__init__.py':
            return

        subset = set()
        for path in mod.__path__:
            for sub in os.listdir(path):
                fullname = os.path.join(path, sub)
                if os.path.isdir(fullname):
                    if os.path.isfile(os.path.join(fullname, '__init__.py')):
                        subset.add(sub)
                elif sub == '__init__.py':
                    continue
                elif sub.endswith('.py'):
                    subset.add(sub[:-3])

        prefix = mod.__name__ + '.'
        for sub in subset:
            recursiveImport(prefix + sub)

    removeAll(name)

    count = [0, 0, 0]
    recursiveImport(name)
    print('# name=%s, pyver=%d.%d.%d' % ((name,) + sys.version_info[:3]))
    print('# all=%d, success=%d, failure=%d' % tuple(count))

if __name__ == '__main__':
    doit('PackageName')
```

成功数と失敗数をレポート

作業フロー



作業フロー



futurize Stage 1 (基本的な書き換え)

Stage 1 の基本的な書き換えを全て実行する。

```
> futurize -1 -w -n TARGET
```

書き換え処理を実行。指定しないとチェックのみ。

指定するとバックアップファイルが作られない。

書き換え対象のファイルやフォルダ。フォルダ指定の場合はその階層下の py ファイルが全て処理される。

- 実行前に py2 と py3 で、全 import チェックを行い、成功数と失敗数を確認。
- 実行後、差分を確認し、問題があれば手で修正。
- 修正を終えたら、全 import チェックを行う。成功数が増えるはず。
  成功していたものが失敗にならなければ OK 。

futurize -1 : __future__ 関連

print_function

インポートされていない状態で print が使用されていたら、インポートされ、書き換えが行われる。
私の経験では、常に完璧に書き換えてくれ、非常に助かった。

absolute_import

インポートされていない状態での絶対表記による相対インポートが検出されたら、インポートされ修正される。

division

stage 1 では対応できないため stage 2 での対応となるが、本セミナーでは、次フェーズの pylint での検出と手動書き換えを推奨。

unicode_literals

-u オプションを付けると import されるが、それだけなので、するべきではない。

futurize -1 : __future__ 関連

✓ **print_function**

インポートされていない状態で print が使用されていたら、インポートされ、書き換えが行われる。
私の経験では、常に完璧に書き換えてくれ、非常に助かった。

✓ **absolute_import**

インポートされていない状態での絶対表記による相対インポートが検出されたら、インポートされ修正される。

~~division~~

stage 1 では対応できないため stage 2 での対応となるが、本セミナーでは、次フェーズの pylint での検出と手動書き換えを推奨。

~~unicode_literals~~

-u オプションを付けると import されるが、それだけなので、するべきではない。

-a オプションを付けると unicode_literals 以外の3つが無条件で import されるが、特に division は対策が必須なので、この段階ではすべきではない。

futurize -1 : __future__ の注意点

__future__ の import は、ファイルの先頭か docstring の後に挿入される。しかし、その後にも別の文字列が続いていたりすると、futurize は挿入位置を誤ってしまう。

```
# coding: utf-8
u"""
○○○モジュール。

これは※※※を□□□するものです。
"""
from __future__ import print_function

u"""
何か別の文字列がさらに続いている場合。
"""
from __future__ import print_function


def hello():
    print('Hello world')
#### さらに続く・・・
```

本来はここに挿入されるべき

間違えてしまう

このモジュールはインポートエラーになるので、全 import チェックですぐに気付ける。

futurize -1：属性名などの修正

py3 で廃止される属性へのアクセスは py2/3 で問題ないように修正される。

関数の属性

- func_closure -> __closure__
- func_code -> __code__
- func_defaults -> __defaults__
- func_doc -> __doc__
- func_dict -> __dict__
- func_globals -> __globals__
- func_name -> __name__

メソッドの属性

- im_func -> __func__
- im_self -> __self__
- im_class -> __self__.__class__

処理中の例外情報取得

- (sys.exc_type, sys.exc_value, sys.exc_traceback) -> sys.exc_info()

futurize -1：関数の引数アンパックの対策

py3 では、関数の引数のアンパック受け取りが廃止されるので、適当な変数を介する形に書き換えられる。

```
def foo(a, (b, c)):
  print((a, b, c))
foo(1, (2, 3))
```

```
def foo(a, xxx_todo_changeme):
  (b, c) = xxx_todo_changeme
  print((a, b, c))
foo(1, (2, 3))
```

確認して、変数を適切な名前に変更する程度で良いだろう

```
lambda (a, b): b
```

```
lambda a_b: a_b[1]
```

lambda でも同様

```
a, (b, c) = [1, [2, 3]]
print((a, b, c))
```

このような代入は py3 でも問題ない

futurize -1：その他にされること

- 組み込み関数 reduce() を使用していたら functools から import
- x.next() を next(x) に書き換え

  x が何であろうが単純な書き換えがされるだけなので要確認。
- x.has_key(b) を b in x に書き換え

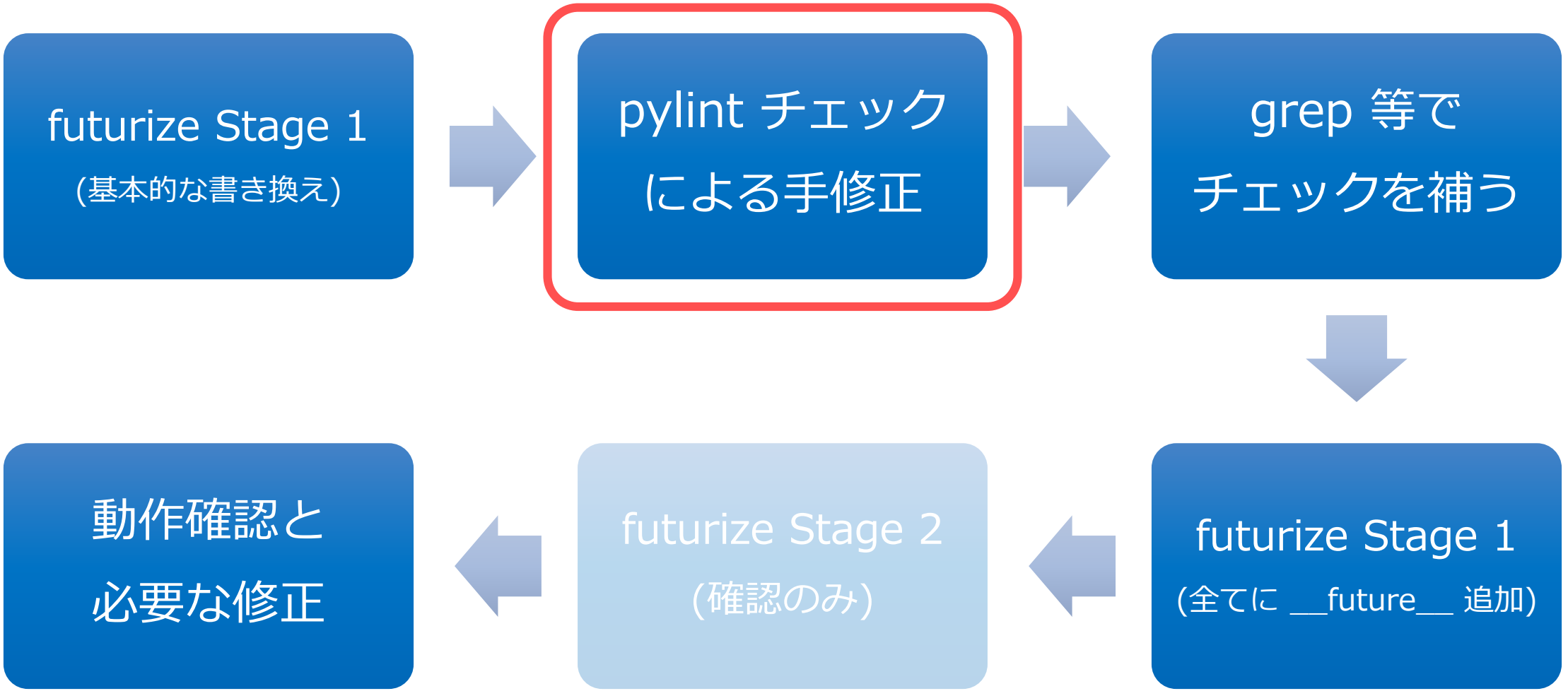
  x が何であろうが単純な書き換えがされるだけなので要確認。
- sys.maxint を使用していたら sys.maxsize に書き換え

  本当にそれで良いかは要確認。
  私は、互換レイヤーに以下のような定数を用意して使用（MAXINT32）。

```
MAXINT32 = int(2**31 - 1)  #: 32bit符号付き整数の最大値。Maya の sys.maxint はこれ（py3 で廃止）
MAXINT64 = int(2**63 - 1)  #: 64bit符号付き整数の最大値。Maya の sys.maxsize はこれ
```

- ほかにも、いろいろ…

作業フロー



pylint --py3k

py3 で問題になる部分をチェックする。

```
> pylint --py3k -d W1618 TARGET
```

無視する警告を指定。
ここでは absolute_import が書かれていない場合の警告を無視するようにしている。

書き換え対象のファイルやフォルダ。フォルダはパッケージのみ認識され階層下が全て処理されるわけではない。__path__ 拡張にも対応しない。

- 最低限必要な absolute_import は前のフェーズで挿入されており、全てに挿入する処理はこの後で行うので、ここでは無視。
- 警告箇所１つ１つについて確認し、必要な対策をし、再びチェックの繰り返し。
- 書き換えたら、全 import チェックも行う。

pylint --py3k：割り算のチェック

- __future__.division を import しておらず、割り算の結果の違いが問題になりそうな箇所が警告される。
- 確認して必要な対策をする。
  - ファイル先頭で __future__.division を import 。
  - int で int を割っているなら、演算子を // に変更。
  - そうでなければ、演算子はそのまま。
  - ただの float ではなく truediv が実装されていないなら掛け算に変更。

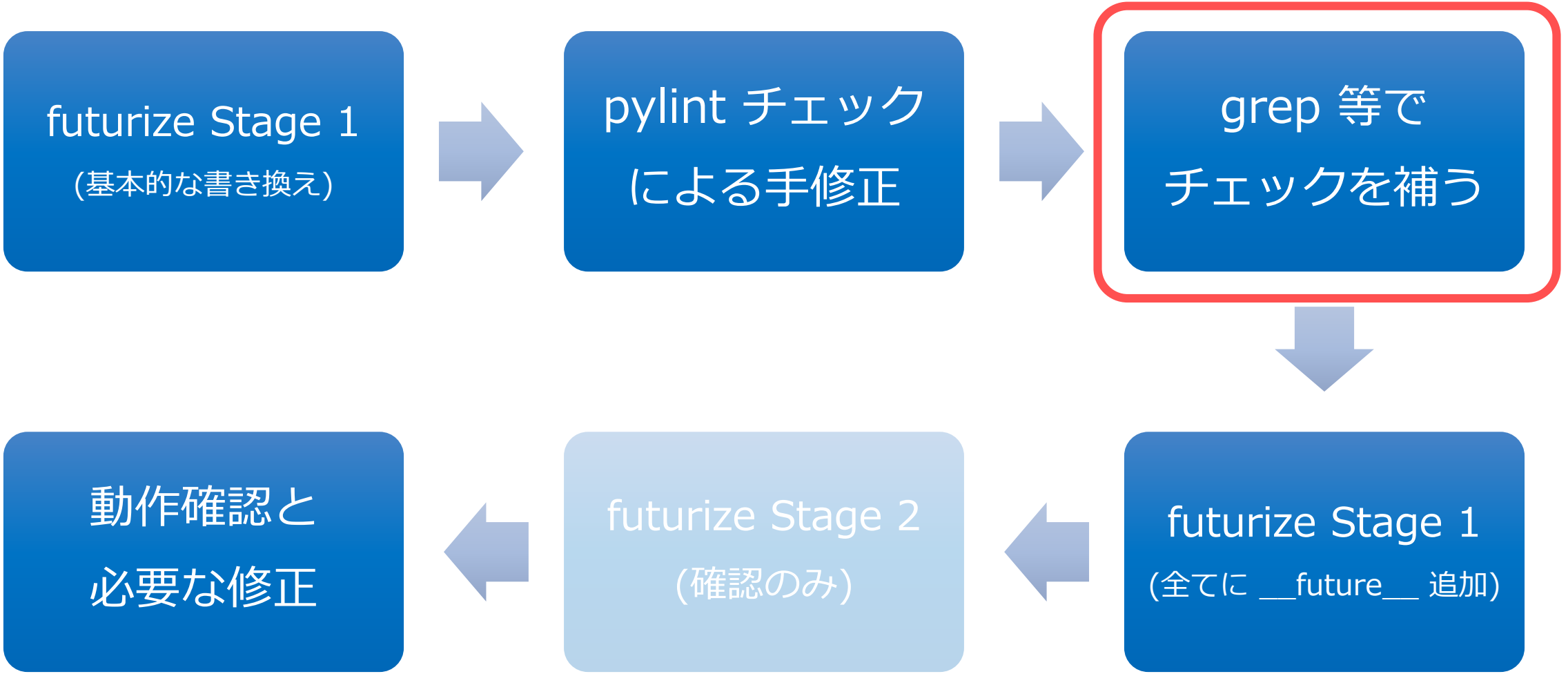
```
import maya.api.OpenMaya as api
d = 2
api.MVector(1, 2, 3) / d
```

```
from __future__ import division
import maya.api.OpenMaya as api
d = 2
api.MVector(1, 2, 3) * (1. / d)
```

pylint −−py3k：その他

- 挙げればきりがないので、ここでは挙げません。
  - 後ほどの「py2/3 違い徹底解説」のときに言及。
  - 全て知っていなくても「ああ、py2/3 にはこんな違いもあるのか」という発見がある。
  - futurize -2 でも対応できないことまでチェックできて便利。
- 逆に、チェックできないことを重点的に手動で対策。
  - 単純なことでも何故かチェックできないものがある -> 次フェーズ
  - 静的解析の限界や根が深い問題 -> いろいろある（後ほど説明）

作業フロー



grep などでのチェック

pylint で検出されないもののうち、全ファイルの文字列検索で単純に見つけ出せそうなものを追加でチェックする。

演算子 /= の使用箇所

何故か pylint は警告してくれないので、自力で探して division 対策。

dict のループ中の内容変更

keys(), values(), items() のループ中で内容変更をしている箇所。
たとえば、左のコードは py3 ではエラーになるので、右のように対策が必要。

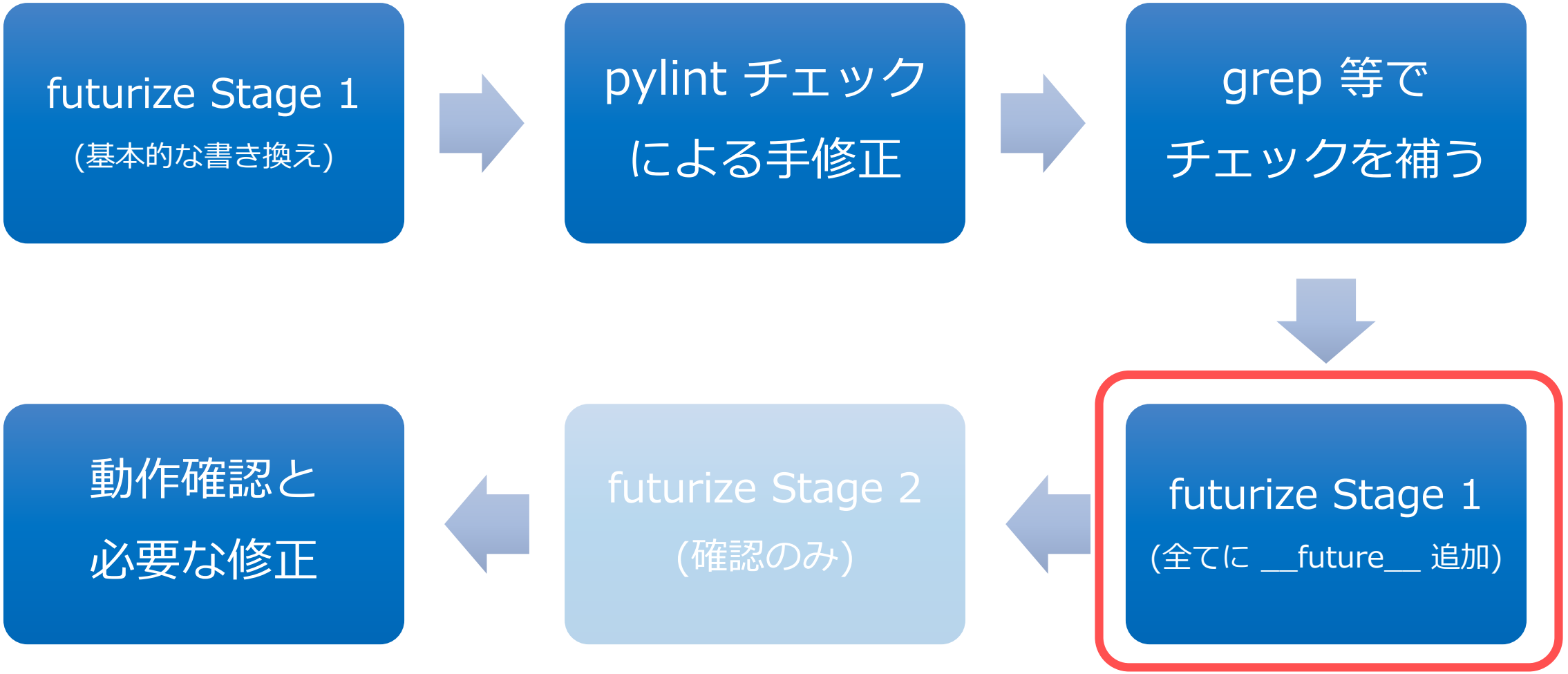
```
for k in mydic.keys():
    if k.startswith('hoge'):
        del mydic[k]
```

```
for k in list(mydic):
    if k.startswith('hoge'):
        del mydic[k]
```

次のような正規表現でループを探して、処理内容をチェックするなど。

```
for .+ in .+¥.(keys|values|items)¥(¥):
```

作業フロー



futurize Stage 1 (全てに __future__ 追加)

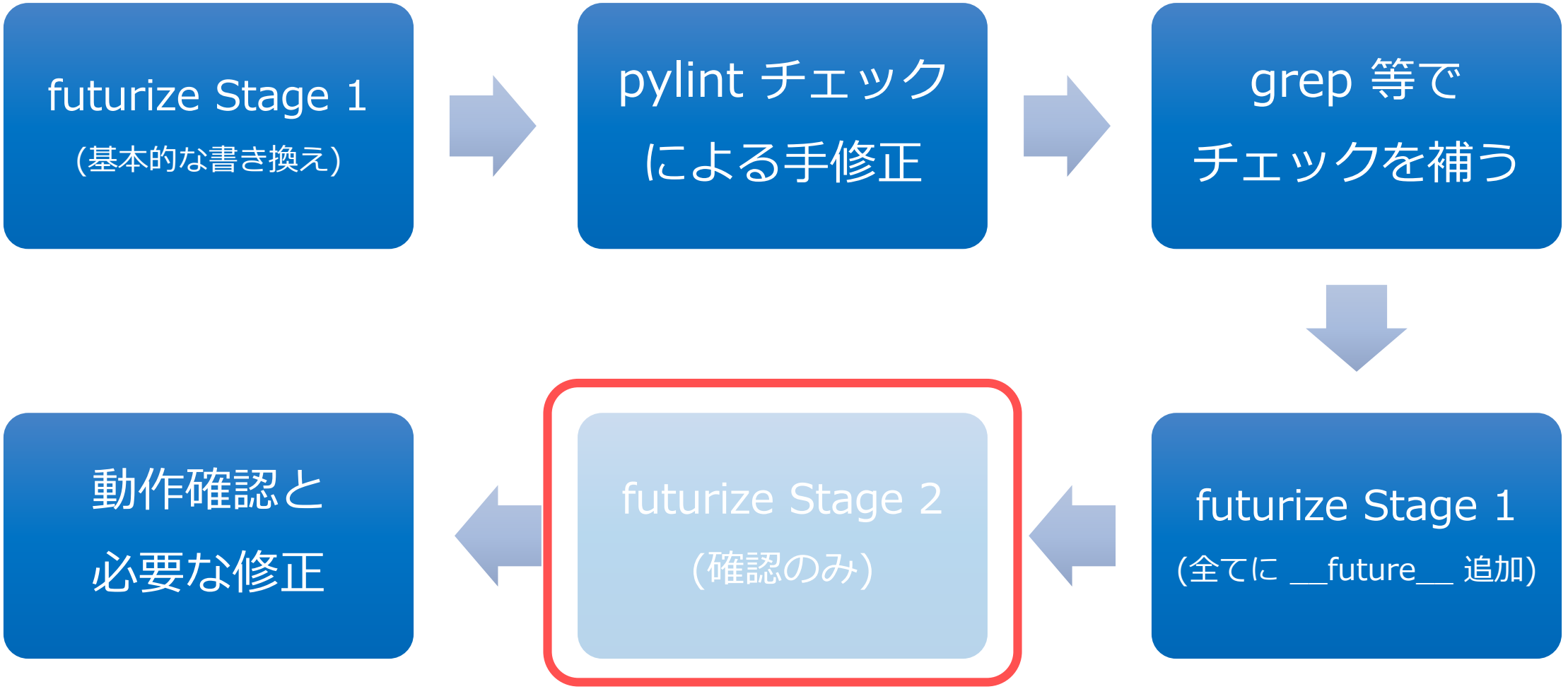
全ファイルに __future__ を追加する。

```
> futurize -1 -a -w -n TARGET
```

unicode_literals 以外の
全てを無条件で追加する

- ２回目なので、それ以外の書き換えは何も起こらない。
- 単純なので、実行後の差分を確認するまでもない。
- __future__ が正しくない位置に挿入される問題もあるので、
  全 import チェックは行い、間違いがあれば手で修正する。

作業フロー



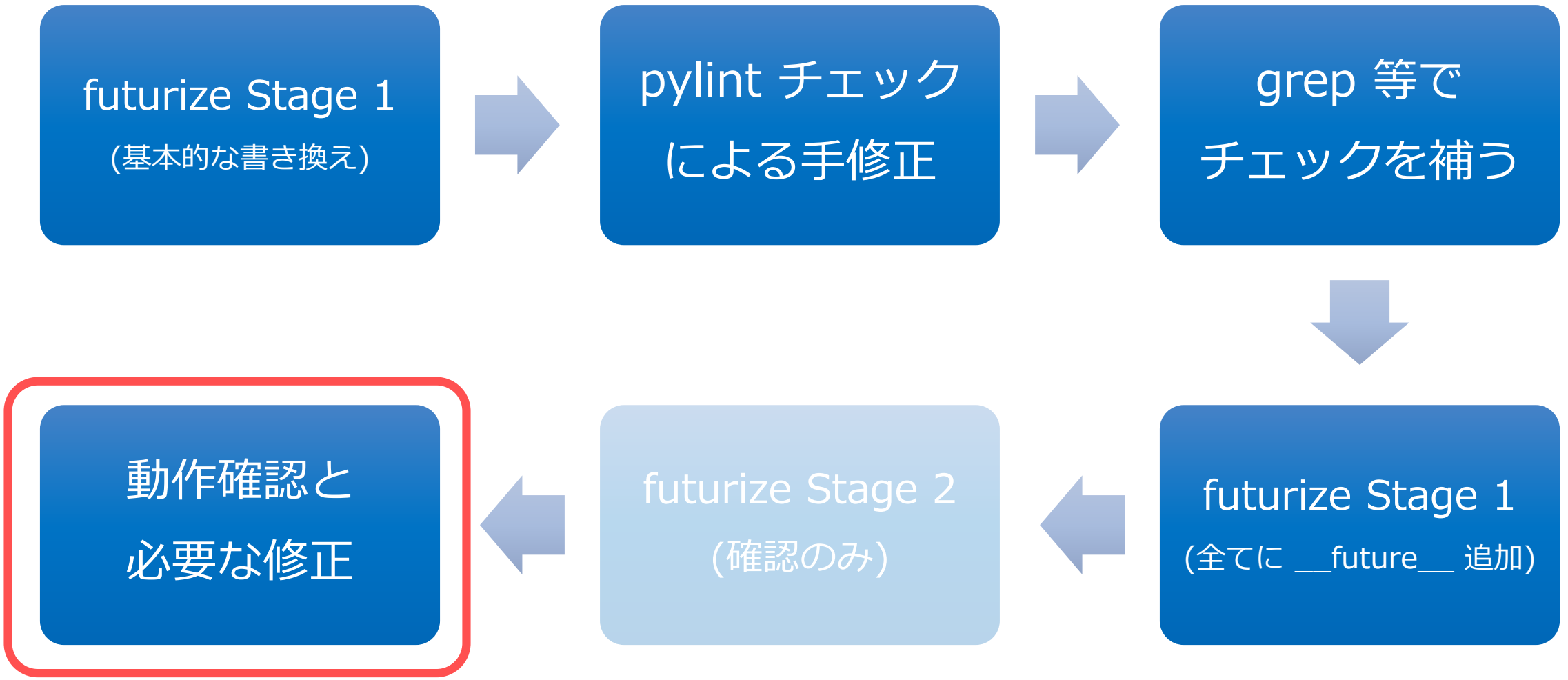
futurize Stage 2 (確認のみ)

書き換えはせずに、問題箇所のチェックツールとして利用。

```
> futurize -2 TARGET
```

- pylint チェックをきちんとしていれば、必要なことは何もされないことが多い。
  それよりも、あらかじめ書き直した部分が、変な風に書き換えられてしまう。
- 私は、当初はチェック用に使っていたが、途中で使わなくなった。
- Stage 2 も使う方針なら、その前の pylint チェックや自身での対策は何もしない方が良いかもしれない。

作業フロー



Python 2 と 3 の違い

移動や廃止

廃止：futurize -1 で変換できるもの

- py2/3 共通の書き方に移行できるもの　futurize -1 で変換可能
  - メソッドの属性 im_* -> 特殊属性に移行
  - 関数の属性 func_* -> 特殊属性に移行
  - 組み込み関数 reduce() -> itertools の関数へ
  - 処理中の例外情報 sys.exc_* -> sys.exc_info() に移行
  - d.has_key(k) -> k in d に統一
  - i.next() -> next(i) に移行
- 完全に廃止されるもの（書き換えは容易）　futurize -1 で変換可能
  - 関数の引数アンパック受け取り -> 不可
  - sys.maxint -> int の上限が無くなり廃止

ライブラリの移動：pylint で検出できるもの

- UserList/UserString -> collections.*
- html 関連 -> html.*
- http 関連 -> http.*
- urlparse/urllib/urlparse/urllib/urllib2 -> urllib.*
- XML-RPC 関連 -> xmlrpc.*
- Tkinter 関連 -> tkinter.*
- Queue -> queue
- ConfigParser -> configparser
- SocketServer -> socketserver
- _winreg -> winreg
- copy_reg -> copyreg
- repr -> reprlib
- markupbase -> _markupbase
- thread -> _thread

もう利用価値は無いようなものだが、
なぜか UserDict は pylint でも futurize でも検出できない

一部 pylint で検出されないものがあった

Maya® には備わっていない（2022 py3 には含まれる）

futurize -2 で対応可能

python-future 互換レイヤーを使えば、py3 の書き方に統一できるが、
使わなくても次のような形で書けば良いだけ。

```
try:
    from urlparse import urlparse
except ImportError:
    from urllib.parse import urlparse
```

ライブラリの移動：pylint で検出できないもの

pylint で検出されないが、futurize -2 で書き換え可能：

- urllib2.Request -> urllib.request.Request
- urllib2.HTTPError -> urllib.error.HTTPError

pylint でも futurize -2 でも検出されないが、互換レイヤーではサポート：

- UserDict.UserDict -> collections.UserDict

pylint でも futurize -2 でも互換レイヤーでも非サポート：

- collections.* (ABC) -> collections.abc.*

  こんな感じに書いて対応する。

```
try:
    from collections.abc import Container, Hashable, Iterable, Sized, Sequence
except ImportError:
    from collections import Container, Hashable, Iterable, Sized, Sequence
```

その他の廃止

組み込み関数 execfile() 廃止

pylint で検出可能

futurize -2 で対応可能

互換レイヤーに次のような関数を用意して代用する。

```
def execfile(fname, globals=None, locals=None):
    if globals is None:
        globals = {'__name__': '__main__'}
    exec(compile(open(fname, 'rb').read(), fname, 'exec'), globals, locals)
```

os.path.walk() 廃止

pylint でも futurize でも検出不可

py2 でも利用可能な os.walk() に移行すれば良いが、使い方が違うので書き換える。

```
def visit(arg, dirname, files):
    for fname in files:
        x = os.path.join(dirname, fname)
        if os.path.isfile(x):
            print(x)
os.path.walk(root, visit, None)
```

```
for dirname, dirs, files in os.walk(root):
    for fname in files:
        print(os.path.join(dirname, fname))
```

その他の廃止

組み込み関数 reload() 廃止

pylint で検出可能

py2 でも利用可能な imp.reload() に移行すれば良いが…、そもそも reload を使わずに、sys.modules から import 済みモジュールを削除する手法がおすすめ。

```
# hoge から始まるもの（hoge* や hoge.* など）を全て削除（import していないことに）する。
for k in list(sys.modules):
    if k.startswith('hoge'):
        del sys.modules[k]
```

types に定義された組み込み型の別名は廃止

pylint で検出可能

単に、型名を直接利用すれば良い。

```
from types import FloatType
print(isinstance(1.1, FloatType))
```

```
print(isinstance(1.1, float))
```

組み込み型 file 廃止

pylint で検出可能

ファイルを開く際は、file() ではなく常に open() を利用すれば良いが、注意点も（後述）。

Python 2 と 3 の違い

様々なこと

基本型の変更：long が int に

- py2 の整数型には int と long の２種類がある
  - int
    - sys.maxint−1 ～ sys.maxint の範囲の値を表現
    - 通常のマシンでは 32bit 精度 (sys.maxint = 2147483647)
  - long
    - 範囲は無限
    - 整数リテラルの接尾辞 L
- py3 では int が long 相当に
  - long という型名は廃止
  - 整数リテラルの接尾辞 L は廃止
  - sys.maxint は廃止

基本型の変更：str が unicode に

- py2 の文字列型には str の unicode の 2 種類がある
  - どちらも抽象基底クラス basetring を継承する
- py3 では str が unicode 相当に
  - unicode という型名は廃止
  - basestring という型名は廃止

文字列と整数の型名の変更は、以下のような互換レイヤーで吸収。

```
if sys.hexversion < 0x3000000:
    BYTES = str
    UNICODE = unicode
    BASESTR = basestring
    LONG = long
else:
    BYTES = bytes
    UNICODE = str
    BASESTR = str
    LONG = int
```

math.floor() と math.ceil()

float の関数だが int を返すようになった。

pylint でも futurize でも検出不可

py2

```
>>> math.floor(1.2), math.ceil(1.2)
(1.0, 2.0)
```

py3

```
>>> math.floor(1.2), math.ceil(1.2)
(1, 2)
```

int と long の区別が無くなり、
さらに int と float の区別も py2 より緩くなっているので、
数値の型によって処理を分けるような実装はしてはならない。

もし、過去のコードでそのような箇所があれば要注意。

組み込み関数 round()

int を返すようになっただけでなく、違う結果を返す。

pylint で検出可能

futurize -2 で対応可能

py2

```
>>> [round(x) for x in (-3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5)]
[-4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0]
```

py3

```
>>> [round(x) for x in (-3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5)]
[-4, -2, -2, 0, 0, 2, 2, 4]
```

py2 は「四捨五入」だが、py3 は「Banker's Rounding」になった。端数が 5 のとき、上位桁が偶数なら 0 方向へ、奇数なら∞方向へ丸められる。丸め方向が一定よりも誤差の累積が抑えられる。

future では py3 の round() のバックポートが用意されているが、私の利用範囲では、丸め方法の違いが悪影響を生むことは無いと判断し、型を統一するだけの簡易な互換レイヤーで済ませた。

```
if sys.hexversion < 0x3000000:
    def round(f, ndigits=None):
        return int(fround(f)) if ndigits is None else fround(f, ndigits)
    fround = round
else:
    def fround(f, ndigits=0):
        return round(f, ndigits or 0)
    round = round
```

py3 の round() と同じく、デフォルトだと int を返す

常に float を返せる fround() 関数も追加

Exception

キャッチの古い書き方の廃止。 futurize -1 で変換可能

```
try:
    raise RuntimeError()
except RuntimeError, e:
    print(e)
```

```
try:
    raise RuntimeError()
except RuntimeError as e:
    print(e)
```

例外オブジェクトの message() メソッド廃止。 pylint で検出可能

```
try:
    raise RuntimeError()
except RuntimeError as e:
    print(e.message())
```

```
try:
    raise RuntimeError()
except RuntimeError as e:
    print(str(e))
```

その場での message() 呼び出しは pylint で検出されるが、他の関数に渡してその中で呼び出しているような場合は検出されない。そのような**「静的解析の限界」**というものは随所にある。

例外オブジェクト以外の raise を認めない。 pylint で検出可能

```
raise 'hoge'
```

```
raise RuntimeError('hoge')
```

dict の要素コレクション取得メソッド

Python 2.7 で、list を返す keys/values/items、イテレータを返す iter* に対し、「辞書ビューオブジェクト」を返す view* が追加された。

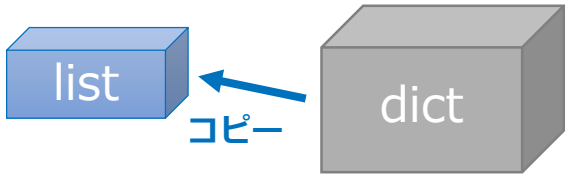
Python 3.x では、通常の keys/values/items がビューになり、他は廃止された。

戻り値の型	2.6 まで	2.7	3.x
list	keys() values() items()	keys() values() items()	n/a
イテレータ	iterkeys() itervalues() iteritems()	iterkeys() itervalues() iteritems()	n/a
辞書ビューオブジェクト	n/a	viewkeys() viewvalues() viewitems()	keys() values() items()

リスト？　ビュー？　イテレータ？　ジェネレータ？

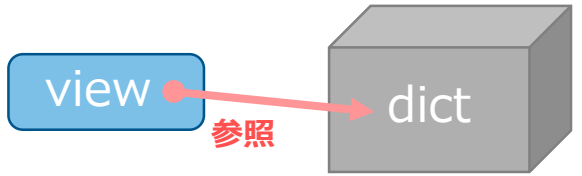
リスト

オブジェクト要素の完全なコピー。



ビュー

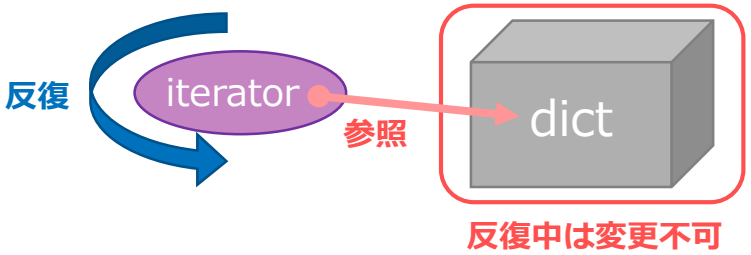
オブジェクトを参照し、変更を常に反映する。



イテレータ

オブジェクトを反復中のインタフェース。

ジェネレータ



イテレータの一種。
よく混同されるが、py3 の collections.abc では、send(), throw(), close() メソッドを持つイテレータとされている。

yield を含む関数や、ジェネレータ式が返すオブジェクト。

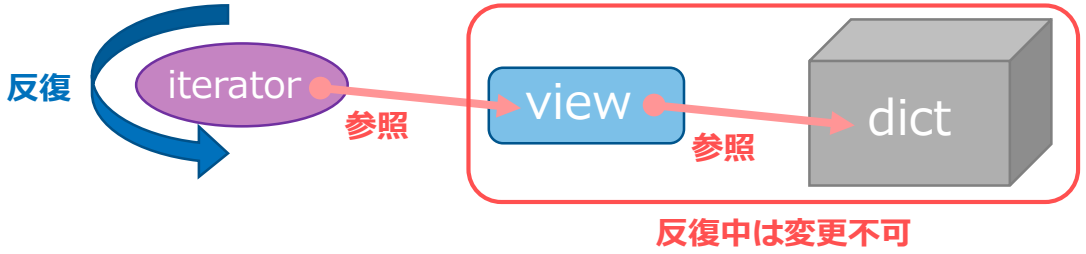
反復中は変更不可のイメージ

リスト

```
# py2 (py3 だとビューになるのでエラー)
for k in dic.keys():
    dic[k] += 1
# py2/3 で問題なし
for k in list(dic):
    dic[k] += 1
```

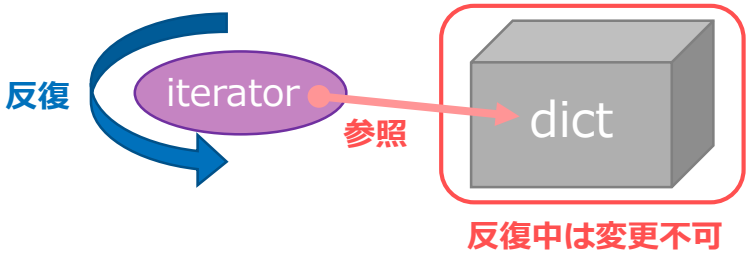
ビュー

```
# py2
for k in dic.viewkeys():
    dic[k] += 1  # エラー
```



イテレータ

```
# py2
for k in dic.iterkeys():
    dic[k] += 1  # エラー
```



互換レイヤーの例

```
if sys.hexversion < 0x3000000:
    dict_get_keys = lambda d: d.keys()
    dict_get_values = lambda d: d.values()
    dict_get_items = lambda d: d.items()

    dict_iterkeys = lambda d: d.iterkeys()
    dict_itervalues = lambda d: d.itervalues()
    dict_iteritems = lambda d: d.iteritems()

    if hasattr(dict, 'viewkeys'):  # 2.7 以降
        dict_keys = lambda d: d.viewkeys()
        dict_values = lambda d: d.viewvalues()
        dict_items = lambda d: d.viewitems()
    else:
        dict_keys = None
        dict_values = None
        dict_items = None

else:
    dict_get_keys = lambda d: list(d)
    dict_get_values = lambda d: list(d.values())
    dict_get_items = lambda d: list(d.items())

    dict_iterkeys = lambda d: iter(d)
    dict_itervalues = lambda d: iter(d.values())
    dict_iteritems = lambda d: iter(d.items())

    dict_keys = lambda d: d.keys()
    dict_values = lambda d: d.values()
    dict_items = lambda d: d.items()
```

リスト

イテレータ

ビュー

リスト

イテレータ

ビュー

keys/values/items を：

- for ループでただ反復しているだけなら py2/3 の違いは問題ない。
- いったん他の変数に受けている場合は、それは list のようには使えないので、pylint で警告される。
  互換レイヤーを用いて、list を得られることを保証する。
- ループ中の内容変更は、pylint では検出できないので、自力で見つける。

その他の「list を返さなくなったもの」

- 組み込み関数 range() が xrange() 相当になり（実際は少し進化している）、xrange() は廃止。
- 組み込み関数 zip(), map(), filter() が itertools の izip(), imap(), ifilter() 相当になり、それらは廃止。
- itertools の izip_longest(), ifilterfalse() が、接頭辞 i を除いた名前に変更。

dict と同様に、pylint でチェック、次のような互換レイヤーを用いて対応をする。

```
if sys.hexversion < 0x3000000:
    lrange = range
    lzip = zip
    lmap = map
    lfilter = filter

    xrange = xrange
    from itertools import izip, imap, ifilter
    if sys.hexversion >= 0x2060000:
        from itertools import izip_longest, ifilterfalse
```

list を返す

```
else:
    lrange = lambda *a: list(range(*a))
    lzip = lambda *a: list(zip(*a))
    lmap = lambda *a: list(map(*a))
    lfilter = lambda *a: list(filter(*a))

    xrange = range
    izip = zip
    imap = map
    ifilter = filter
    from itertools import (
        zip_longest as izip_longest,
        filterfalse as ifilterfalse)
```

list を返す

「list を返さないもの」まとめ

py2	py3	イテレータ	__contains__ 実装	Sized	インデックス	スライス
xrange				✓	✓	
	range		✓	✓	✓	✓
izip	zip	✓				
imap	map	✓				
ifilter	filter	✓				
izip_longest	zip_longest	✓				
ifilterfalse	filterfalase	✓				
dict.iterkeys		✓				
dict.itervalues		✓				
dict.iteritems		✓				
dict.viewkeys	keys		✓	✓		
dict.viewvalues	values			✓		
dict.viewitems	items		✓	✓		

「list を返さないもの」まとめ

py2	py3	イテレータ	__contains__ 実装	Sized	インデックス	スライス
xrange				✓	✓	
	range		✓	✓	✓	✓
izip	zip	✓				
imap	map	✓				
ifilter	filter	✓				
izip_longest	zip_longest	✓				
ifilterfalse	filterfalase	✓				
dict.iterkeys		✓				
dict.itervalues		✓				
dict.iteritems		✓				
dict.viewkeys	keys		✓	✓		
dict.viewvalues	values			✓		
dict.viewitems	items		✓	✓		

１周イテレーションすると終わりなので特に注意

リスト内包表記の変数の隔離

リスト内包表記の中で使用された一時変数が外のスコープを汚さなくなった。
ジェネレータ式の場合は元々問題なかった。

```
x = None
a = [x for x in range(10)]
print(x)

x = None
a = list(x for x in range(10))
print(x)
```

py2

```
9
None
```

py3

```
None
None
```

pylint は、リスト内包表記の後にその変数にアクセスしているコードを検出し警告してくれる。
しかし、検出されない場合もあり、del 操作は検出されないようだ。

リスト内包表記が使った変数を、ご丁寧に削除しているコードは要注意（py3 でエラー）。

```
a = [x for x in range(10)]
print(x)

a = [y for y in range(10)]
del y
```

pylint で検出可能

pylint でも futurize でも検出不可

比較：cmp 関数の廃止

組み込み関数 cmp() の廃止

pylint で検出可能

futurize -2 で対応可能

cmp(a, b) は、a < b なら -1、a > b なら 1、a == b なら 0 となる。

使わないのが一番良いが、どうしても使いたい場合は、同等の関数を簡単に作ることができる。

```
>>> cmp(1, 1.2)
-1
```

```
>>> cmp = lambda a, b: (a > b) - (a < b)
>>> cmp(1, 1.2)
-1
```

sort 関数/メソッドなどの cmp オプション廃止

pylint で検出可能

```
>>> words = ['foo', 'bar', 'baz']
>>> tbl = dict(foo=2, bar=3, baz=1)
>>> cmpwd = lambda a, b: cmp(tbl[a], tbl[b])
>>>
>>> sorted(words, cmp=cmpwd)
['baz', 'foo', 'bar']
```

key オプションに移行するのが良い。

```
>>> sorted(words, key=lambda x: tbl[x])
['baz', 'foo', 'bar']
```

難しい場合は、cmp 関数を key 関数に変換可能。

```
>>> from functools import cmp_to_key
>>> sorted(words, key=cmp_to_key(cmpwd))
['baz', 'foo', 'bar']
```

比較：大小比較における TypeError

py2 では、何でも大小比較できてしまう。
結果は一貫していて矛盾はないが、その規則性はよくわからない。

```
>>> 3 < '1'
True
>>> None < None
False
>>> None <= None
True
>>> 0. > None
True
>>> zip > map
True
```

py3 では、自然な順序付けができないものは TypeError 。

```
>>> 3 < '1'
TypeError: '<' not supported between instances of 'int' and 'str'
>>> None < None
TypeError: '<' not supported between instances of 'NoneType' and 'NoneType'
>>> None <= None
TypeError: '<=' not supported between instances of 'NoneType' and 'NoneType'
>>> 0. > None
TypeError: '>' not supported between instances of 'float' and 'NoneType'
>>> zip > map
TypeError: '>' not supported between instances of 'type' and 'type'
```

Namespace Package

- py3 では、__init__.py を含まないフォルダもパッケージとしてインポート可能
  - そのようなパッケージを Namespace Package と呼ぶ。
  - __file__ 属性は None になる。
  - __path__ 属性には _NamespacePath オブジェクトが設定され、sys.path 上の同名フォルダが自動的に合成される（通常のパッケージでは __path__ は list で、拡張するには手動で設定するもの）。
- py2/3 相互運用では利用すべきものではないが、通常のモジュールと同名のフォルダが優先パス上にたまたま存在すると、本来のモジュールがインポートできなくなるので注意が必要。

Python 2 と 3 の違い

クラス関連

古いスタイルのクラスの廃止

py2 に残っていた（が使うべきではない）古いスタイルのクラスは完全に廃止。
py3 では親クラス指定を省略したら object 指定になるだけ。

py2

```
>>> class Foo:
...     pass
...
>>> issubclass(Foo, object)
False
>>> isinstance(Foo(), object)
True
>>> Foo.mro()
AttributeError: class Foo has no attribute 'mro'
```

py3

```
>>> class Foo:
...     pass
...
>>> issubclass(Foo, object)
True
>>> isinstance(Foo(), object)
True
>>> Foo.mro()
[<class '__main__.Hoge'>, <class 'object'>]
```

古いスタイルはやめて、常に親クラスを指定する。

py2/3

```
>>> class Foo(object):
...     pass
...
>>> issubclass(Foo, object)
True
>>> isinstance(Foo(), object)
True
>>> Foo.mro()
[<class '__main__.Hoge'>, <class 'object'>]
```

組み込み関数 super()

py3 では、メソッド内では super() の引数を省略できる（メソッド外で使用する場合は不可）。
py2/3 で動くようにするには、従来通り、引数を省略しないようにする。

py3

```
>>> class Foo(object):
...     def a(self):
...         print('Foo.a(%s)' % type(self).__name__)
...
...     @classmethod
...     def b(cls):
...         print('Foo.b(%s)' % cls.__name__)
...
>>> class Bar(Foo):
...     def a(self):
...         print('Bar.a(%s)' % type(self).__name__)
...         super().a()
...
...     @classmethod
...     def b(cls):
...         print('Bar.b(%s)' % cls.__name__)
...         super().b()
...
>>> Bar().a()
Bar.a(Bar)
Foo.a(Bar)
>>> Bar().b()
Bar.b(Bar)
Foo.b(Bar)
>>>
>>> super(Bar, Bar()).a()
Foo.a(Bar)
>>> super(Bar, Bar).b()
Foo.b(Bar)
```

py2/3

```
>>> class Foo(object):
...     def a(self):
...         print('Foo.a(%s)' % type(self).__name__)
...
...     @classmethod
...     def b(cls):
...         print('Foo.b(%s)' % cls.__name__)
...
>>> class Bar(Foo):
...     def a(self):
...         print('Bar.a(%s)' % type(self).__name__)
...         super(Bar, self).a()
...
...     @classmethod
...     def b(cls):
...         print('Bar.b(%s)' % cls.__name__)
...         super(Bar, cls).b()
...
>>> Bar().a()
Bar.a(Bar)
Foo.a(Bar)
>>> Bar().b()
Bar.b(Bar)
Foo.b(Bar)
>>>
>>> super(Bar, Bar()).a()
Foo.a(Bar)
>>> super(Bar, Bar).b()
Foo.b(Bar)
```

object の __new__ と __init__ の引数

py2 では、object の __new__ や __init__ に、本来は不要な引数を渡しても問題なかったが、py3 だとエラーになるようになった。

pylint でも futurize でも検出不可

```python
class Foo(object):
    def __new__(cls, *args):
        print('Foo.__new__' + repr(args))
        return super(Foo, cls).__new__(cls, *args)

    def __init__(self, *args):
        print('Foo.__init__' + repr(args))
        super(Foo, self).__init__(*args)

class Bar(Foo):
    def __new__(cls, *args):
        print('Bar.__new__' + repr(args))
        return super(Bar, cls).__new__(cls, *args)

    def __init__(self, *args):
        print('Bar.__init__' + repr(args))
        super(Bar, self).__init__(*args)

Bar(1,2,3)
```

```python
class Foo(object):
    def __new__(cls, *args):
        print('Foo.__new__' + repr(args))
        return super(Foo, cls).__new__(cls)

    def __init__(self, *args):
        print('Foo.__init__' + repr(args))
        super(Foo, self).__init__()

class Bar(Foo):
    def __new__(cls, *args):
        print('Bar.__new__' + repr(args))
        return super(Bar, cls).__new__(cls, *args)

    def __init__(self, *args):
        print('Bar.__init__' + repr(args))
        super(Bar, self).__init__(*args)

Bar(1,2,3)
```

メソッド名変更

- __nonzero__ -> __bool__
- next -> __next__

どちらも
pylint で検出可能

どちらも
futurize -2 で対応可能

py2

```
>>> class Half(object):
...     def __init__(self, val):
...         self._it = iter(val)
...
...     def __iter__(self):
...         return self
...
...     def __nonzero__(self):
...         return True
...
...     def next(self):
...         return next(self._it) * .5
...
>>> a = list(range(5))
>>> i = Half(a)
>>> list(i)
[0.0, 0.5, 1.0, 1.5, 2.0]
```

py2/3

```
>>> class Half(object):
...     def __init__(self, val):
...         self._it = iter(val)
...
...     def __iter__(self):
...         return self
...
...     def __bool__(self):
...         return True
...     __nonzero__ = __bool__
...
...     def __next__(self):
...         return next(self._it) * .5
...     next = __next__
...
>>> a = list(range(5))
>>> i = Half(a)
>>> list(i)
[0.0, 0.5, 1.0, 1.5, 2.0]
```

コピー
するだけ

ただの名前変更なので、メソッドを別名でコピーすれば良いだけ。
python-future では py3 の挙動に変更した object から派生させる対応をするため、Stage 2 でないと書き換えてくれない。py3 風にした object は MRO が一段増えたラッパーなので微妙。

割り算の特殊メソッド

div 系は廃止され truediv 系に

pylint で検出可能

- __div__ -> __truediv__　… self / other
- __idiv__ -> __itruediv__　… self /= other
- __rdiv__ -> __rtuediv__　… other / self
- py2 では__future__.division がインポートされると truediv 系が呼ばれるように切り替わる。
- コピーして両方用意しておけば OK 。

floordiv 系の仕様には変更なし

- py2/3 とも、// 演算子で floordiv 系が呼ばれる。
- __floordiv__　… self // other
- __ifloordiv__　… self //= other
- __rfloordiv__　… other // self

比較演算子の特殊メソッド

__cmp__ が廃止されるので、
py2/3 共通の比較演算子ごとのメソッドを実装する。

pylint で検出可能

- __eq__ … self == other
- __ne__ … self != other
- __le__ … self <= other
- __lt__ … self < other
- __ge__ … self >= other
- __gt__ … self > other

__eq__ を実装すると、
py3 では __ne__ は省略できるが、
py2 だとできないので両方実装する

py2

```
class Foo(object):
    def __init__(self, v):
        self._v = v

    def __cmp__(self, other):
        if isinstance(other, Foo):
            return cmp(self._v, other._v)
        return 1
```

py2/3

```
class Foo(object):
    def __init__(self, v):
        self._v = v

    def __eq__(self, other):
        return isinstance(other, Foo) and self._v == other._v

    def __ne__(self, other):
        return not isinstance(other, Foo) or self._v != other._v

    def __lt__(self, other):
        if isinstance(other, Foo):
            return self._v < other._v
        return NotImplemented

    def __le__(self, other):
        if isinstance(other, Foo):
            return self._v <= other._v
        return NotImplemented
```

もう一方の __lt__ と __le__ で解決
できるので __ge__ と __gt__ を省略

比較不可の場合は
NotImplement を返す

自動的な unhashable 化

py3 では __eq__ をオーバーライドすると __hash__ もオーバーライドしない限り unhashable になる。（自動的に __hash__ に None がセットされる）

```
>>> class Foo(object):
...     def __init__(self, a):
...         self._a = a
...
...     def __eq__(self, x):
...         return isinstance(x, Foo) and self._a == x._a
...
...     def __hash__(self):
...         return hash(self._a)
...
>>> class Bar(Foo):
...     def __init__(self, a, b):
...         super(Bar, self).__init__(a)
...         self._b = b
...
...     def __eq__(self, x):
...         return super(Bar, self).__eq__(x) and self._b == x._b
...
>>> f = Foo(1)
>>> b = Bar(2, 3)
>>> {f: 100, b: 200}
TypeError: unhashable type: 'Bar'
```

pylint で検出可能

```
>>> class Foo(object):
...     def __init__(self, a):
...         self._a = a
...
...     def __eq__(self, x):
...         return isinstance(x, Foo) and self._a == x._a
...
...     def __hash__(self):
...         return hash(self._a)
...
>>> class Bar(Foo):
...     def __init__(self, a, b):
...         super(Bar, self).__init__(a)
...         self._b = b
...
...     def __eq__(self, x):
...         return super(Bar, self).__eq__(x) and self._b == x._b
...
...     __hash__ = Foo.__hash__
...
>>> f = Foo(1)
>>> b = Bar(2, 3)
>>> {f: 100, b: 200}
{<__main__.Foo object at 0x……>: 100, <__main__.Bar object at 0x……>: 200}
```

__eq__ を実装したら __hash__ も再実装するか、親クラスからコピーする。

hashable について整理

hashable object とは :

- dict のキーになれたり、set に格納できる。
- 生存期間中変わらないハッシュ値を持ち (__hash__ メソッドが必要)、他のオブジェクトと比較ができる (__eq__ メソッドが必要) オブジェクト。
- 等価なオブジェクト同士は、必ず同じハッシュ値を持つ必要がある（同じハッシュ値であっても、必ずしも等価である必要はない）。

__eq__ を再定義したということは、__hash__ も再定義しないとハッシュ値が妥当でなくなる可能性がある。

```
>>> class Foo(object):
...     def __init__(self, a):
...         self._a = a
...
...     def __eq__(self, x):
...         return isinstance(x, Foo) and self._a == x._a
... ...
>>> d = {Foo(1): 100}
>>> d[Foo(1)]
KeyError: <__main__.Foo object at 0x……>
```

pylint で検出可能

```
>>> class Foo(object):
...     def __init__(self, a):
...         self._a = a
...
...     def __eq__(self, x):
...         return isinstance(x, Foo) and self._a == x._a
...
...     __hash__ = None
...
>>> d = {Foo(1): 100}
TypeError: unhashable type: 'Foo'
```

object は __hash__ 実装を持つ（id() より得ている）ので、py2 では全てのクラスが意図しない hashable になり得る。
py3 ではそれが抑制されるが、py2/3 で同じ動作にするには __hash__ を実装するか None をセットする。

メタクラス指定方法の変更

クラスにメタクラスを指定するときの書き方が py2 と py3 で異なる。

py2

```
>>> from weakref import ref as _wref
>>>
>>> class Singleton(type):
...     def __call__(cls, *args, **kwargs):
...         ref = getattr(cls, '_ref', None)
...         obj = ref and ref()
...         if obj is None:
...             obj = type.__call__(cls, *args, **kwargs)
...             cls._ref = _wref(obj)
...         return obj
...
>>> class Foo(object):
...     __metaclass__ = Singleton
...
>>> a = Foo()
>>> b = Foo()
>>> a is b
True
```

pylint で検出可能

py3

```
>>> from weakref import ref as _wref
>>>
>>> class Singleton(type):
...     def __call__(cls, *args, **kwargs):
...         ref = getattr(cls, '_ref', None)
...         obj = ref and ref()
...         if obj is None:
...             obj = type.__call__(cls, *args, **kwargs)
...             cls._ref = _wref(obj)
...         return obj
...
>>> class Foo(object, metaclass=Singleton):
...     pass
...
>>> a = Foo()
>>> b = Foo()
>>> a is b
True
```

文法が異なるので、書き分けることはできないし、仮にできたとしてもクラス実装丸ごと分岐は厳しい…。

```
if sys.hexversion < 0x3000000:
    class Foo(object):
        __metaclass__ = Singleton

else:
    class Foo(object, metaclass=Singleton):
        pass
```

py2 で文法エラー

C/C++ なら #ifdef とか（プリプロセッサ）でやるんだけど…

ところで、メタクラスって何？

Python では、ほとんど全てのものが「オブジェクト」。
ただの数値も、関数も、クラスも、みんなオブジェクト。print は文だったがオブジェクトになった。
オブジェクトとは、クラス（型）のインスタンス（実体）。

```
>>> class Foo(object):
...     pass
...
>>> obj = Foo()
>>> type(obj)
<class '__main__.Foo'>
>>> type(Foo)
<class 'type'>
>>> type(type)
<class 'type'>
```

obj は Foo 型（クラス）のオブジェクト

Foo は type 型（クラス）のオブジェクト

type も type 型（クラス）のオブジェクト ……

メタクラスとは type の派生クラスで、クラスのクラスのふるまいをカスタマイズするもの。

```
>>> class Singleton(type):
...     def __call__(cls, *args, **kwargs):
...         ref = getattr(cls, '_ref', None)
...         obj = ref and ref()
...         if obj is None:
...             obj = type.__call__(cls, *args, **kwargs)
...             cls._ref = _wref(obj)
...         return obj
...
```

type の派生クラス（メタクラス）

```
>>> class Foo(object, metaclass=Singleton):
...     pass
...
>>> type(Foo)
<class '__main__.Singleton'>
>>> obj = Foo()
>>> type(obj)
<class '__main__.Foo'>
```

通常のクラスは type 型だがメタクラス指定で変えられる

Foo クラスは Singleton 型のオブジェクト

obj は Foo 型のオブジェクト

with_metaclass

python-future や six の with_metaclass を使うと共通の書き方ができる。

py2/3

```
from future.utils import with_metaclass

class Foo(with_metaclass(Singleton, object)):
    pass
```

futurize -2 で対応可能

python.future や six を使わない場合でも、短いので引用しやすい。

```
# Function from jinja2/_compat.py. License: BSD.
def with_metaclass(meta, *bases):
    class metaclass(meta):
        __call__ = type.__call__
        __init__ = type.__init__
        def __new__(cls, name, this_bases, d):
            if this_bases is None:
                return type.__new__(cls, name, (), d)
            return meta(name, bases, d)
    return metaclass('temporary_class', None, {})
```

Retrieved from python-future 0.18.2 (from jinja2 / MIT LIcense)

```
def with_metaclass(meta, *bases):
    class metaclass(meta):
        def __new__(cls, name, this_bases, d):
            return meta(name, bases, d)
    return type.__new__(metaclass, 'temporary_class', (), {})
```

Retrieved from six 1.12.0 (BSD License)

どちらも Maya®2022 同梱のバージョンより引用。
若干異なるが、どちらもほぼ同じことをしているので six の方がすっきりしている。

with_metaclass の考察

どちらの場合もメタクラスを解決したダミー親クラスを返すので、それを継承することに。
jinja2 版のコメントには「ダミー親クラスが MRO に含まれない点で six より優れる」とあるが、それは過去の話。

six 1.6.1 の場合：

```
def with_metaclass(meta, *bases):
    return meta('DummyClass', bases, {})
```

```
>>> class Foo(with_metaclass(Singleton, object)):
...     pass
...
>>> Foo.mro()
[<class '__main__.Foo'>, <class '__main__.DummyClass'>, <class 'object'>]
```

ダミーの親クラスが含まれてしまう

six 1.12.0 や python-future の場合：

```
def with_metaclass(meta, *bases):
    class DummyMeta(meta):
        def __new__(cls, name, this_bases, d):
            return meta(name, bases, d)
    return type.__new__(DummyMeta, 'DummyClass', (), {})
```

```
>>> class Foo(with_metaclass(Singleton, object)):
...     pass
...
>>> Foo.mro()
[<class '__main__.Foo'>, <class 'object'>]
```

本来のメタクラスを継承する DummyMeta クラスを生成し、
その DummyMeta によって DummyClass を生成して返す。
派生クラスが作られる際、DummyMeta の __new__ が呼ばれる。

それは「 DummyClass を親とする DummyMeta 型のクラス」を作るという指示になるが、それを無視して、本来の親と、本来のメタクラスによってクラスを生成する。

2021年5月現在 最新の six 1.16.1 では、
もう少し補強された実装になっているので要チェック！

unbound method の廃止

クラスをインスタンス化する前のメソッドは、
py2 だと unbound method というが、py3 だとただの関数になった。

それを受けて MethodType の第３引数が廃止された。ただし、もともと不要ともいえる。

py2

```
>>> from types import MethodType
>>>
>>> class Foo(object):
...     def foo(self):
...         return self, 'foo'
...
>>> f = Foo()
>>> f.foo()
(<__main__.Foo object at 0x……>, 'foo')
>>>
>>> # クラスにメソッドを追加。
>>> def bar(self):
...     return self, 'bar'
...
>>> Foo.bar = MethodType(bar, None, Foo)
>>> f.bar()
(<__main__.Foo object at 0x……>, 'bar')
>>>
>>> # インスタンスにメソッドを追加。
>>> def baz(self):
...     return self, 'baz'
...
>>> f.baz = MethodType(baz, f)
>>> f.baz()
(<__main__.Foo object at 0x……>, 'baz')
```

py2/3

```
>>> from types import MethodType
>>>
>>> class Foo(object):
...     def foo(self):
...         return self, 'foo'
...
>>> f = Foo()
>>> f.foo()
(<__main__.Foo object at 0x……>, 'foo')
>>>
>>> # クラスにメソッドを追加。
>>> def bar(self):
...     return self, 'bar'
...
>>> Foo.bar = bar
>>> f.bar()
(<__main__.Foo object at 0x……>, 'bar')
>>>
>>> # インスタンスにメソッドを追加。
>>> def baz(self):
...     return self, 'baz'
...
>>> f.baz = MethodType(baz, f)
>>> f.baz()
(<__main__.Foo object at 0x……>, 'baz')
```

クラスへのメソッド追加は
関数を代入するだけ。

インスタンスへのメソッド追加は
MethodType() が必要。

クラスの属性に関数を代入したときの挙動

つまり、クラスの属性に関数を代入すると、
py2 だと勝手に unbound method が作られるが、py3 だと元の関数のまま。

py2

```
>>> class Foo(object):
...     pass
...
>>> def bar(self):
...     return self, 'bar'
...
>>> bar, type(bar)
(<function bar at 0x……>, <type 'function'>)
>>> Foo.bar = bar
>>> Foo.bar, type(Foo.bar)
(<unbound method Foo.bar>, <type 'instancemethod'>)
>>> Foo.bar is bar
False
>>> f = Foo()
>>> f.bar, type(f.bar)
(<bound method Foo.bar of <__main__.Foo object at 0x……>>, <type 'instancemethod'>)
```

function から unbound method が作られる

別もの

py3

```
>>> bar, type(bar)
(<function bar at 0x……>, <type 'function'>)
>>> Foo.bar = bar
>>> Foo.bar, type(Foo.bar)
(<function bar at 0x……>, <type 'function'>)
>>> Foo.bar is bar
True
>>> f = Foo()
>>> f.bar, type(f.bar)
(<bound method bar of <__main__.Foo object at 0x……>>, <type 'method'>)
```

元の関数のまま

同じもの

メソッドの型名が変わった

Python 2 と 3 の違い

文字列と IO

文字列の扱いの違いはなかなか厄介

	Python 2 の型名	Python 3 の型名	リテラル接頭辞
バイト列	str	bytes	b
ユニコード	unicode	str	u
ネイティブ文字列	str	str	なし

実際この通りではあるが、これほど単純明快ではない。 なぜなら…、

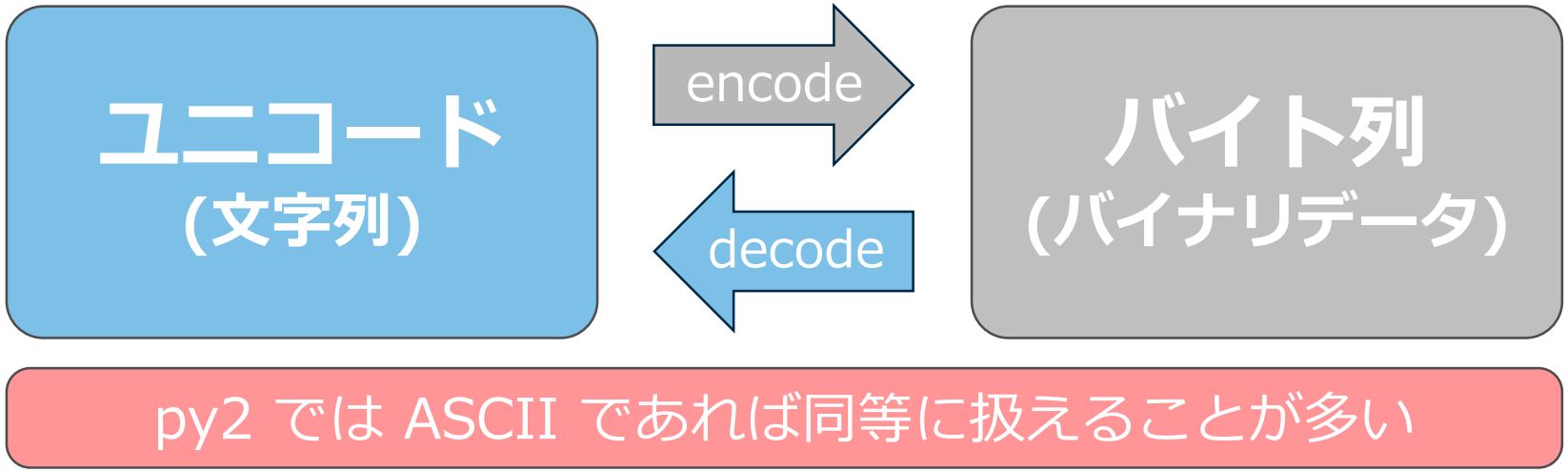
- **py2 では str（バイト列）も unicode も「文字列」**

  どちらも basestring の派生クラスで、どちらかが要求される場面でも、どちらでも受け付けてもらえることが多い。どちらかというと str が文字列（ネイティブ）。

- **py3 では str（ユニコード）のみが「文字列」**

  bytes は py2 の str 相当ではあるが、扱いとしては「バイナリデータ」。
  共通の抽象基底クラスは無く、全くの別もの。

encode と decode にまつわる違い



py2

```
>>> 'foo'.encode('ascii')
'foo'
>>> u'foo'.decode('ascii')
u'foo'
>>> unicode(b'foo')
u'foo'
>>> str(u'foo')
'foo'
```

無意味ではあるが、
decode() と encode() を
両者とも持っている

ASCII なら、
双方向キャストが可能

py3

```
>>> b'foo'.encode('ascii')
AttributeError: 'bytes' object has no attribute 'encode'
>>> u'foo'.decode('ascii')
AttributeError: 'str' object has no attribute 'decode'
>>> str(b'foo')
"b'foo'"
>>> bytes(u'foo')
TypeError: string argument without an encoding
```

str 評価で repr() になっている

ファイル IO

py2 の open()

- file オブジェクトを返す。py3 では廃止される。
- C ランタイムのファイルIOに近い低レベルなもの。
- text モードでも binary モードでも、バイト列で読み書きする。
  ただし、write に ASCII エンコード可能な unicode を渡しても問題ない。

io.open() / py3 の open()

- io.open() は py2/3 共通。py3 の組み込み関数 open() と同じもの。
- binary モードの入力では BufferedReader、出力では BufferedWriter オブジェクトを返し、バイト列で読み書きする。
  py2 の write には ASCII エンコード可能な unicode を渡しても問題ない。
- text モードでは TextIOWrapper オブジェクトを返し、ユニコードで読み書きする。
  py2 の wirte でも unicode しか受け付けない。

py2 の組み込み関数 open()

binary (b) モード

ダイレクトに読み書きする。

text (t) モード

改行コード変換がされる。

- read: プラットフォーム依存の改行コードが ‘¥n’ に変換して読む。
- write: ‘¥n’ をプラットフォーム依存の改行コードに変換して書く。

universal newline (U) モード (read のみ)

3種類の改行コードが認識され ‘¥n’ に変換される。

モード指定を省略した場合のデフォルトは、通常の Python では text だが、
Windows の Maya®（mayapy なら maya initialize 後）では binary となる点に注意！

Windows API の設定を変更しているようで、C の file() でもデフォルトが変わる。

io モジュールの open()

binary (b) モード

バイト列をダイレクトに読み書きする。

text (t) モード

ユニコードで読み書き

encoding オプションを指定可能で、デフォルトはプラットフォーム依存の locale.getpreferredencoding() で得られるもの（日本語 Windows では cp932）。

改行コード変換

- read では universal newline 処理がされる。
- write では newline オプションで指定可能で、デフォルトの None はプラットフォーム依存の改行コード。空文字列 ’’ を指定すると変換しない。

ファイル IO まとめ

open 関数	py2 open()			io.open() / py3 open()	
モード	b (binary)	t (text)	U (universal newline)	b (binary)	t (text)
モード指定省略時のデフォルト	Windows Maya®	通常のPython	n/a	n/a	常時デフォルト
対応する型	バイト列	バイト列	バイト列	バイト列	ユニコード
write 時 py2 型変換	ASCII unicode 指定可	ASCII unicode 指定可	n/a	ASCII unicode 指定可 (py3では不可)	unicode のみ
read 時 改行コード変換	n/a	プラットフォーム依存	universal newline (識別して変換)	n/a	universal newline (識別して変換)
write 時 改行コード変換	n/a	プラットフォーム依存	n/a	n/a	プラットフォーム依存だが newline オプションで指定可能
encode / decode	n/a	n/a	n/a	n/a	プラットフォーム依存だが encoding オプションで指定可能
クラス	file	file	file	BufferedReader / BufferedWriter	TextIOWrapper

ファイルIO の py3 対応はどうするか

バイナリの入出力

組み込み関数 open() を使っていて問題ない。

テキストの入出力

組み込み関数 open() を io.open() に移行する。

改行コード処理の改善

- read: ‘U’ を指定しなくても universal newline になる。
- write: newline オプションを指定してプラットフォーム依存を無くせる。

ASCII テキスト

- read: py2 で unicode になっても str と区別なく扱えるのでほぼ気にする必要はない。
- write: ASCII であっても py2 str は渡せないので注意。

ASCII 以外のテキスト

encoding オプションに注意。
旧 open() なら、呼び出し側で encode/decode をするものだが、そうではなくなる。

StringIO

- 文字列バッファを、ファイルと同じように読み書きするクラス。
- py2 では 3 種類あったが、py3 で 1 種類になった。
- py2/3 共通の io モジュールに移行しなければならないが、扱える型がユニコードのみになる。

モジュール	cStringIO (C実装)	StringIO	io
Python 3	廃止	廃止	2 / 3
対応する型	バイト列	ハイブリッド	ユニコード
write 時 py2 型変換	ASCII unicode 指定可	n/a	unicode のみ
read 時の型	バイト列	1 度でもユニコードが writeされたらユニコード、そうでなければバイト列	ユニコード
コンテキスト対応	no	no	yes

pickle

- C実装のモジュール名変更: cPickle -> _pickle
- シリアライズ: dump(), dumps()
- デシリアライズ: load(), loads()
- dump() と load() で入出力するのはバイナリのIOストリーム。
  （open() でも io.open() でも可）
- dumps() と loads() で入出力するのはバイト列。
  （py2 では文字列だが py3 ではバイナリデータ）

自作クラスも含む Python のあらゆるデータ（オブジェクト）をシリアライズできる
非常に強力な仕組みだが、それだけに言語仕様への依存が強い。

py2/3 の相互運用においては py2 と py3 の違いの影響を強く受ける。

pickle: プロトコルの問題

- dump() や dumps() では、プロトコル（データフォーマット）を指定する。
- py2 では 0～2 が選択可能で、デフォルトは 0 。
- py3 では 0～4 が選択可能で、デフォルトは 3 (py3.8 から 4 になる）。
- py2/3 共通にするには 0～2 を明示する必要がある。
  そのうち、データを ASCII 文字列として扱えるのは 0 のみ。

py2/3 互換の pickle 活用例 :

```
try:
    import cPickle as _pickle
else:
    import _pickle

pkl = _pickle.dumps(data, protocol=0).decode('ascii')
cmds.setAttr('foo.bar', pkl)

cmds.getAttr('foo.bar')
data = _pickle.loads(pkl.encode('ascii'))
```

プロトコルを明示

データをシリアライズして Maya®アトリビュートに保存

Maya®アトリビュートに保存されたデータを復元

pickle: py2 データのロードにおける問題

py2 名のリマップ

- モジュール名などの変更に対応。デフォルトで有効（fix_imports=True）。
- py3 で保存されたものを py2 でロードする際のリマップはできない。

py2 str を py3 str として復元

- デフォルトで有効。
  bytes.decode() と同じオプション encoding='ASCII' と errors='strict' 。
- ASCII テキスト以外のバイト列が含まれると UnicodeDecodeError となる。
- 特殊なコーデック 'bytes' を指定すると処理を無効化できる。
  エラーは回避できるが、もちろん py2 の str は bytes のままとなる（辞書のキーなども！）。
- numpy array や datetime などが含まれる場合は 'latin-1' を指定する必要があるらしい。
- errors='ignore' でエラー発生箇所を無視できる（空文字列となる）。
- 「エラー発生箇所だけ bytes のままとする」ような選択肢は標準では無い。
- py2 で py3 データを読むと str は unicode になるが、それは問題ないものと考えるしかない。

subprocess.Popen()

- 基本的にバイト列での入出力。

  py2 だと文字列だけど py3 ではバイト列…。

- py3 にのみ、ユニコードで入出力する text モードがある。
  - encoding, errors, universal_newlines などのオプション
  - encoding のデフォルトはプラットフォーム依存（日本語Windows では ‘cp932’）

この辺でやめときます・・・

Python C API

Python C API とは

- C で Python オブジェクトを扱うための API
  - Python モジュール（pyd|so）の開発。
  - ctypes 等の何らかのインタフェースを通じた Python オブジェクトの入出力。
- Maya®にも付属
  - lib/Python27.(lib|so), lib/Python37.(lib|so)
  - include/Python27/, include/Python37/
- C で API を通じて Python を書く感覚
  - とにかく手数が非常に多くなるが、使用感は Python を書くことと同じ。
  - C API 固有の概念はほとんどなく、Python と C の知識があれば使える。
  - 全てのオブジェクトは PyObject* で扱われる。
  - 所有オブジェクトの参照カウントは Py_INCREF() と Py_DECREF() で自前で管理。

py2 と py3 の違い

モジュール初期化

- Single-phase

  py2 とは少し書き方が変わるが基本は同じ。
  唯一のモジュールステートを DLL 内 static 変数で管理。

- Multi-phase (PEP 489)

  py3 で追加。モジュールステートを動的に管理。Python で書くモジュールと同等の振る舞いをし、サブインタプリタにも対応できる。
  まだ発展途上で、Python 3.7 ではクラスの弱参照サポートを設定できなかった。

クラス定義

- 型定義構造体 PyTypeObject の内容の変更や追加。

ネイティブ型の変更

- int の取り扱い： PyInt_???() が廃止され PyLong_???() のみに。
- str の取り扱い： PyString_???() が廃止され PyUnicode_???() のみに。

まとめ

まとめ

- これまでの py2 コードを py3 用に変換してダブルメンテナンスするのではなく、py2/3 のどちらでも動く形にするのは現実的に困難ではない。
- どのようにコードを移行させるかの戦略は、状況や人それぞれの考え方によって異なる。あなたの戦略を立案しよう。
- 全てを手で書き換えるのは得策ではなく、ツールを上手に使うのが省力化につながる。
- ツールに完全に依存するくらい活用しても良いが、それでも対応できないことは少なからずある。
- ツールを有効活用し、ツールで対応できないことも克服するには py2/3 の違いについての知識が必要。本セミナーではそれをお伝えした。

ご清聴ありがとうございました。

佐々木 隆典

ryusukes@square-enix.com

SQUARE ENIX ADVANCED TECHNOLOGY DIVISION