PYCON JP 2017

OUTPUT & FOLLOW

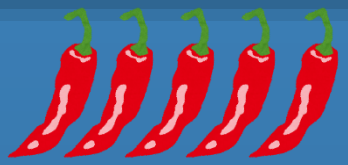
Pythonはどうやってlen関数で長さを手にいれているの？

How does python get length with the len() function?

Takayuki Shimizukawa

おまえ誰よ / Who are you

@shimizukawa（清水川）



- BeProud co, ltd.

- Sphinx committer

- 一般社団法人PyCon JP理事
  Board member of PyCon JP committee

[AD] Python オンライン学習サービス PyQ 機械学習コンテンツをリリース

NEW！

https://pyq.jp/

Pythonはどうやってlen関数で長さを手にいれているの？

How does python get length with the len() function?

Target attendees

The person who think …
以下のように思っている方

- I got into Python, but it doesn't come nicely..
  Pythonは入門したが、なんだかしっくりこない..
- Why len is a function?
  なんでlenは関数なんだろう
- Python is not object-oriented as len() function
  len()関数だなんてPythonはオブジェクト指向じゃないな

アジェンダ

1. len() がオブジェクトの長さを手に入れる方法
   - なんでPythonはlen()関数なの
   - Protocol: オブジェクトの振る舞い
2. if がオブジェクトのTrue/Falseを判断する方法
3. for がオブジェクトの繰り返し要素を取得する方法
4. まとめ
5. References

len() がオブジェクトの長さを手に入れる方法

len()関数に文字列を渡したときに起こること

- len関数で文字列の長さを得ます

```
>>> len(“もじれつ”)
4
```

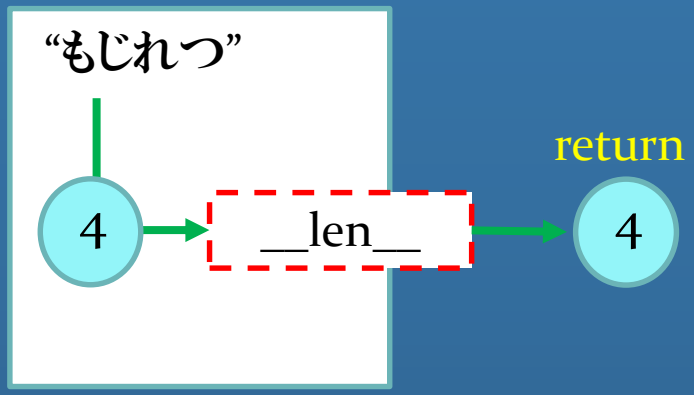
- obj.__len__() メソッドを呼んで文字列の長さを得ます

```
>>> “もじれつ”.__len__()
4
```

- 得られる結果は同じ
- len(obj)は内部で obj.__len__() を実行しています

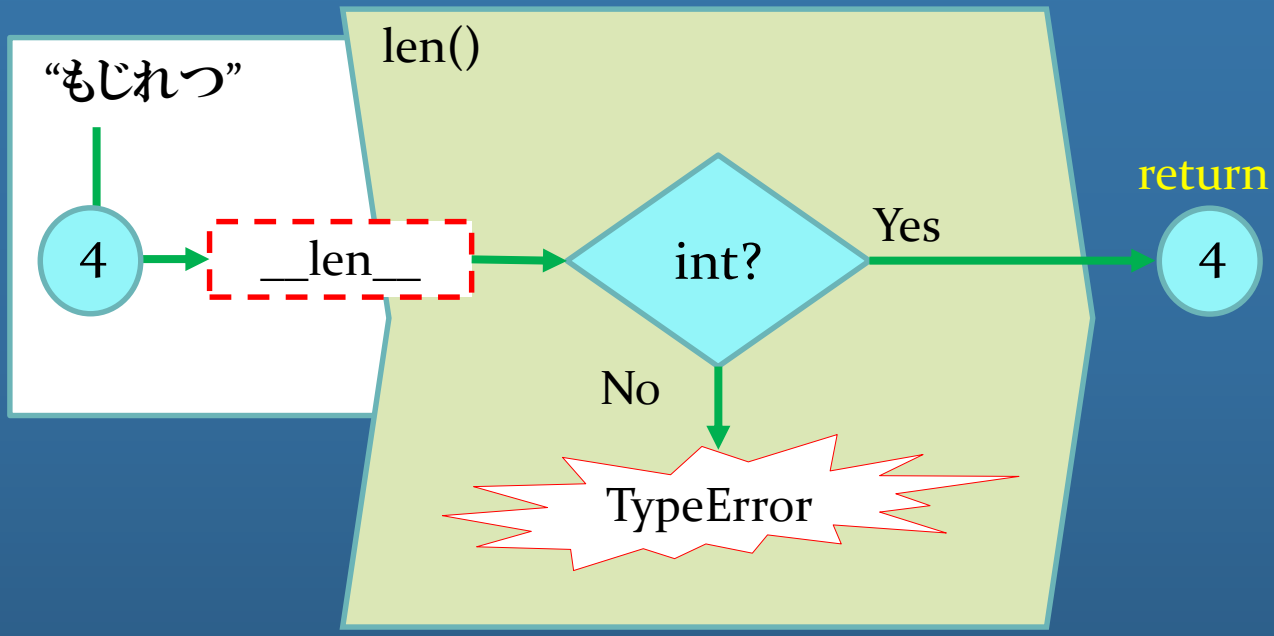
len()要らないのでは？

- obj.__len__() が呼ばれるなら obj.length() でよかったのでは？



len()要らないのでは？

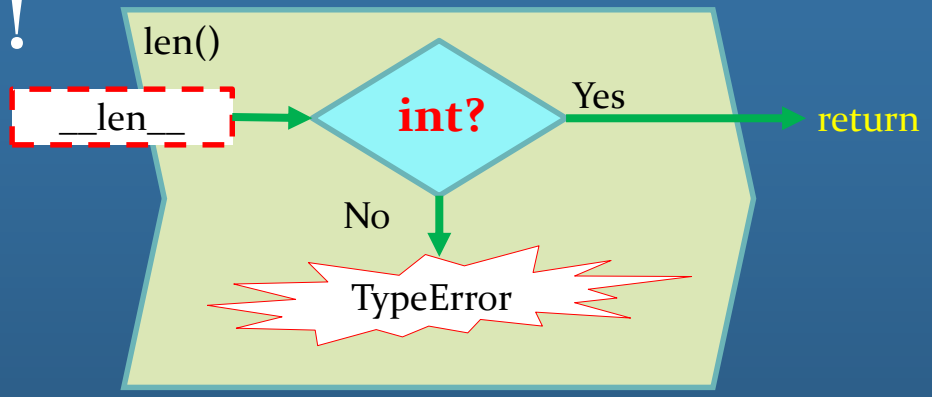
- obj.__len__() が呼ばれるなら obj.length() でよかったのでは？
  - len() はもうちょっと仕事してます
  - __len__() の値がintかチェックしています



__len__() がint以外の値を返すと..

```
>>> class WaruiObj:
...     def __len__(self):
...         return 1.2
...
>>> w = WaruiObj()
>>> len(w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an
integer
```

len関数で、型と値をチェック！

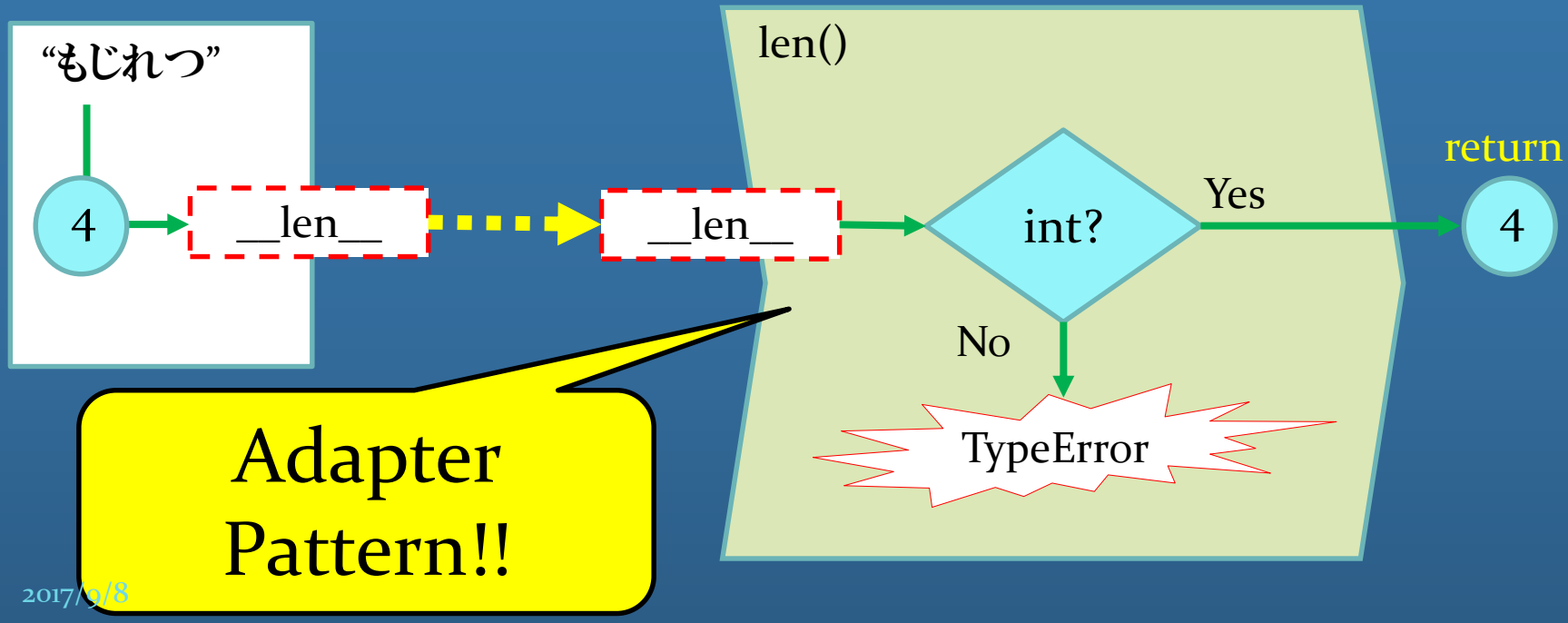


sys.maxsize を超える値を返すと
OverflowError例外を起こします。

— データモデル より

len()関数の役割

- 渡されたオブジェクトの __len__() APIを使って、
- 取得した値をチェック、変換して、
- 呼び出し元に適切な値を返す

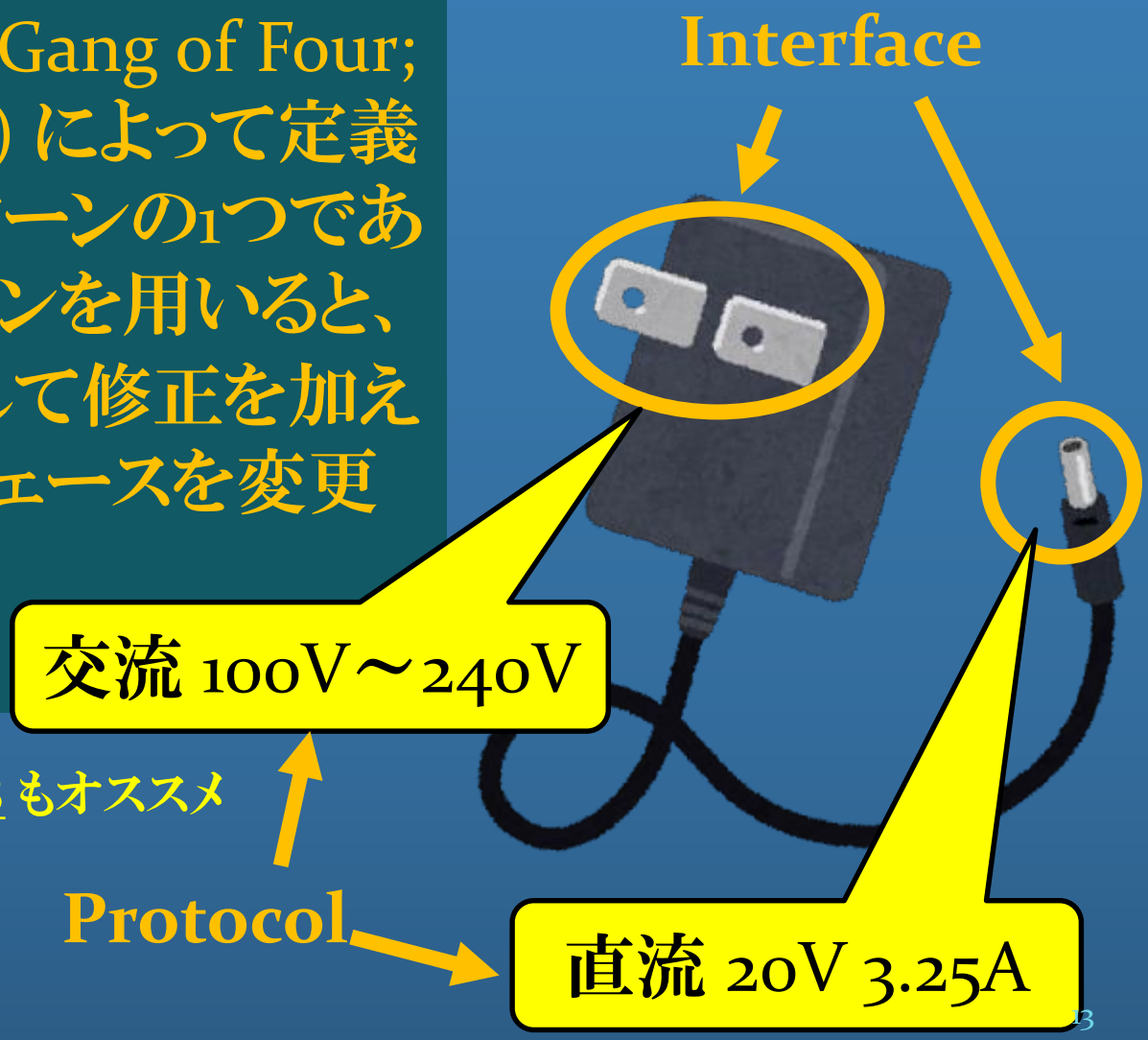


Adapter Pattern とは

Adapter Pattern（アダプター・パターン）とは、GoF (Gang of Four; 4人のギャングたち) によって定義されたデザインパターンの1つである。Adapter パターンを用いると、既存のクラスに対して修正を加えることなく、インタフェースを変更することができる。

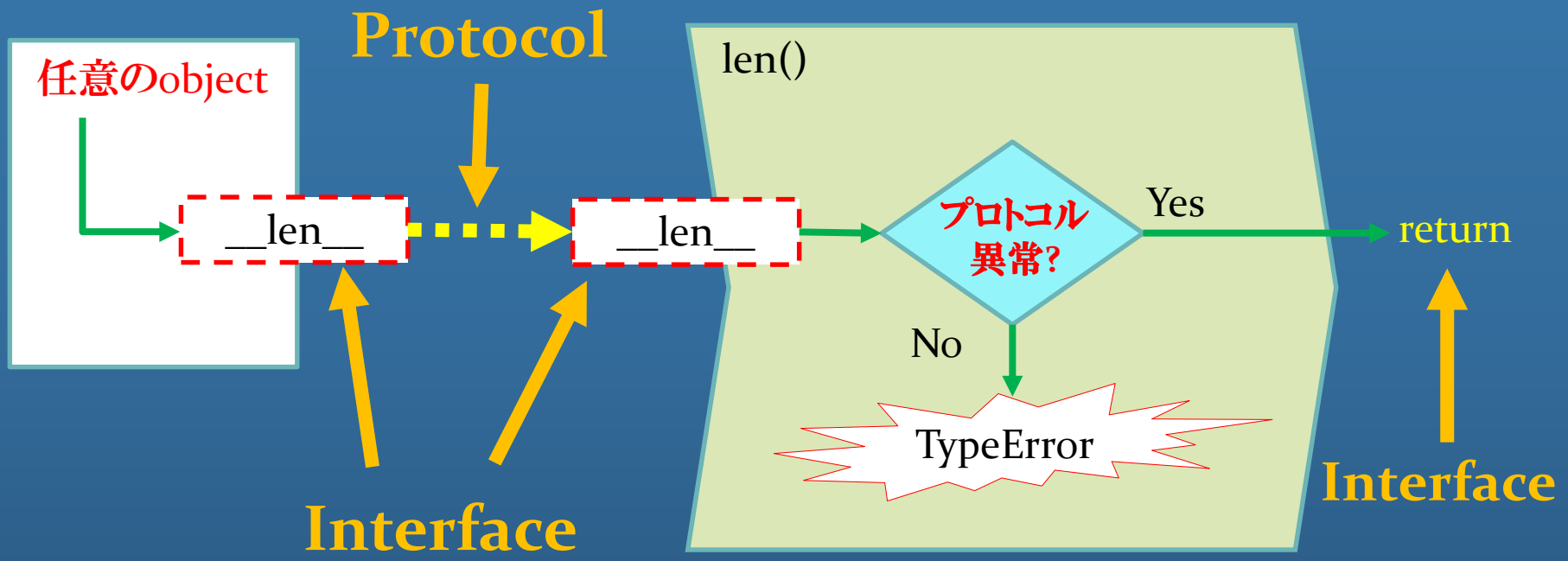
— Wikipedia より

参考: 実践Python3 もオススメ



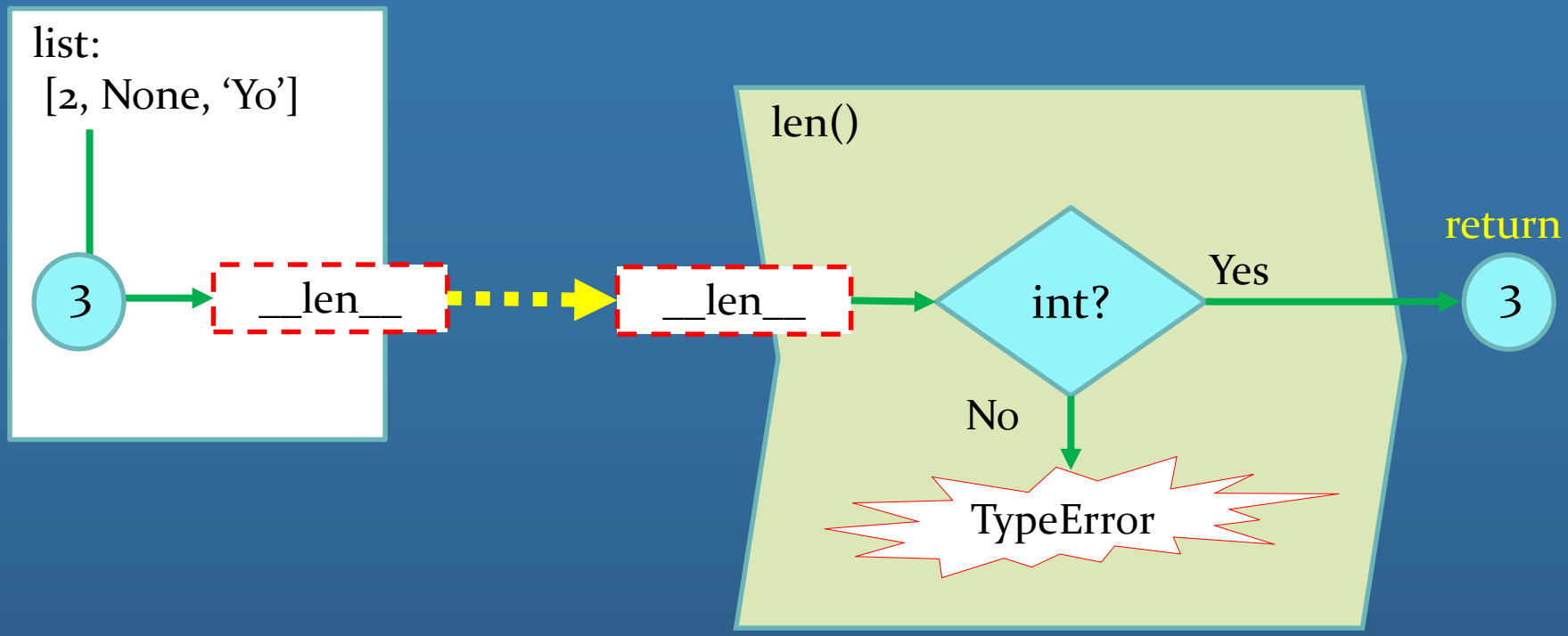
どのオブジェクトにも使えるlen() Adapter

- len() 関数は、任意のオブジェクトに対してAdapterとして作用します。
- len() は、obj.__len__() があれば、どんなオブジェクトにも使えるAdapterです。



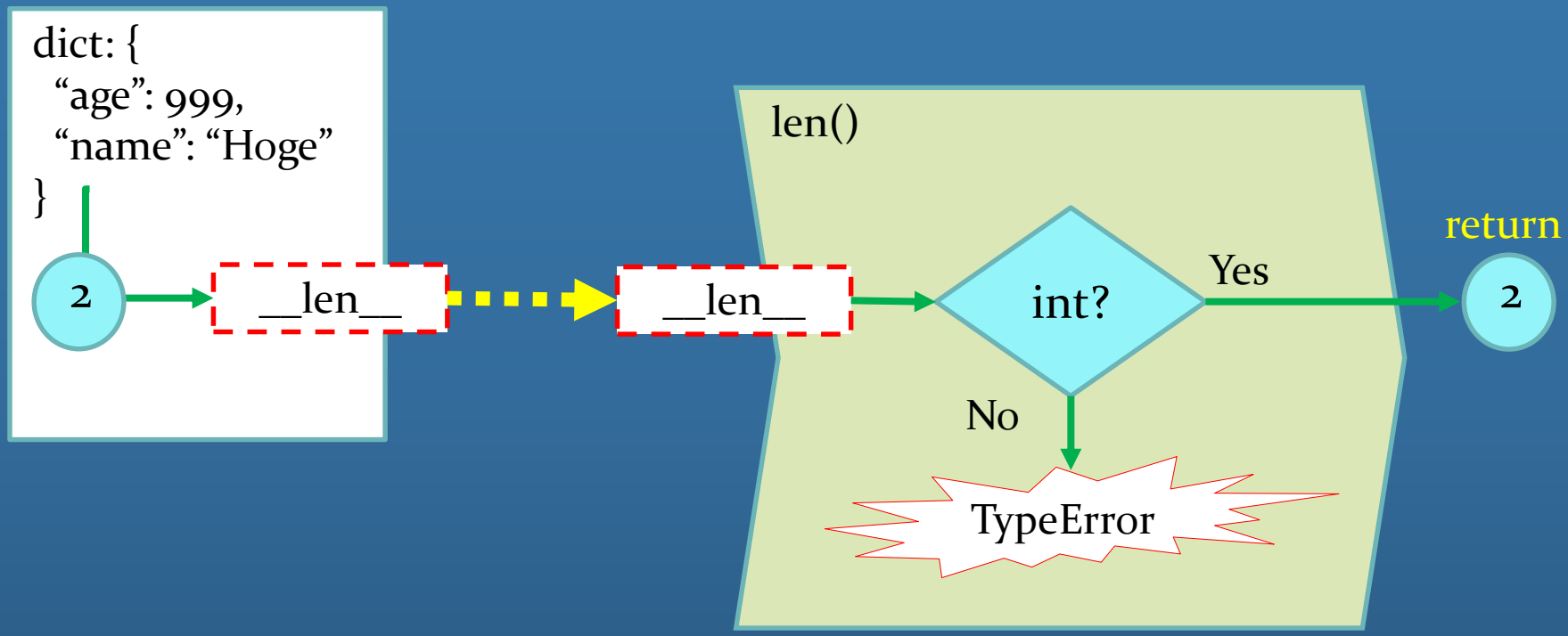
listに len() Adapter

- **list**.__len__() は中に持っている**要素数**を返します。
- 要素数はlist自身が知っています



dictに len() Adapter

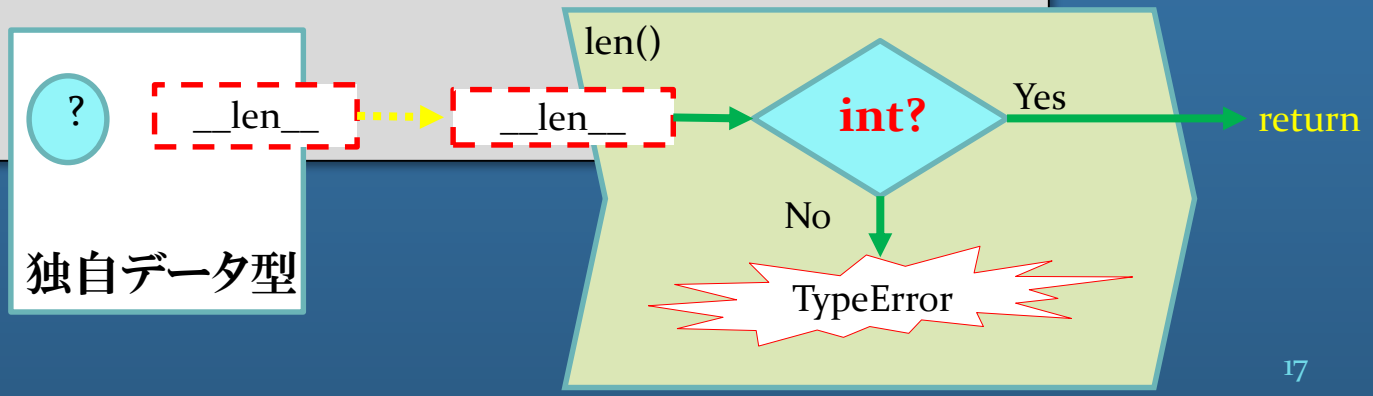
- **dict**.__len__() は中に持っている**キーの数**を返します。
- キーの数はdict自身が知っています



独自のデータ型に len() Adapter

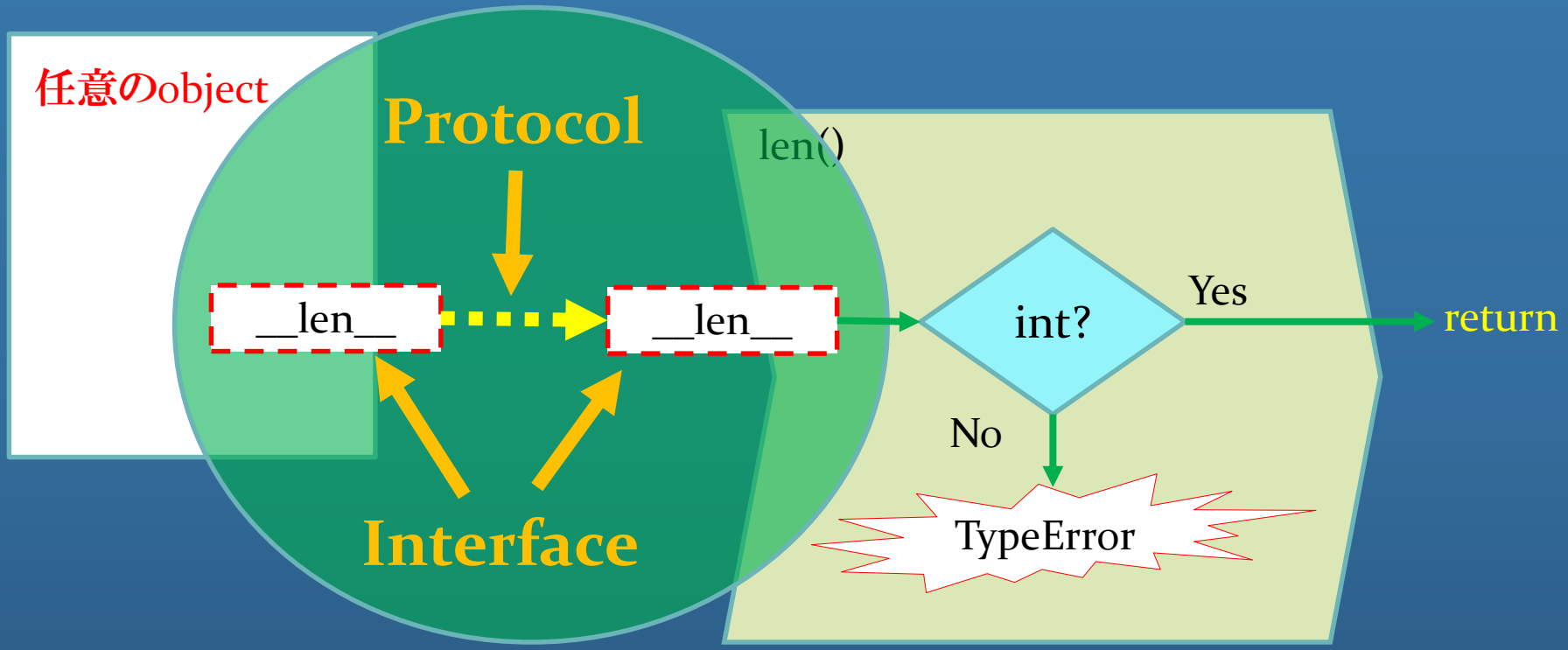
- __len__() メソッドを実装した独自クラスを定義します

```
>>> import random
>>> class Random:
...     def __len__(self):
...         return random.randint(0, 10)
...
>>> r = Random()
>>> len(r)
10
>>> len(r)
0
>>> len(r)
5
```



Protocol: オブジェクトの振る舞い

- len() は、obj.__len__() があれば動作します。
- 言い換えると、長さの概念を持つオブジェクトは、obj.__len__() を実装する必要があります。
- 長さの「**プロトコルを実装する**」と言います。



Protocol ってどこに書いてあるの？

- 実は、Pythonの公式ドキュメントに何度か登場してます

iter(*object*[, *sentinel*]) (原文)

イテレータ (iterator) オブジェクトを返します。 第二引数があるかどうかで、第一引数の解釈は大きく異なります。 第二引数がない場合、 *object* は反復プロトコル (__iter__() メソッド) か、シーケンスプロトコル (引数が 0 から開始する __getitem__() メソッド) をサポートする集合オブジェクトでなければなりません。これらのプロトコルが両方ともサポートされていない場合、 TypeError が送出されます。 第二引数 *sentinel* が与えられているなら、 *object* は呼び出し可能オブジェクトでなければなりません。この場合に生成されるイテレータは、 __next__() を呼ぶ毎に *object* を引数無しで呼び出します。返された値が *sentinel* と等しければ、 StopIteration が送出され、そうでなければ、戻り値がそのまま返されます。

組み込み関数 -Python 標準ライブラリ - Python 3.6.1

クラス -Python チュートリアル - Python 3.6

```
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

イテレータプロトコルの裏にある仕組みを観察していれば、自作のクラ
るのは簡単です。 __next__() メソッドを持つオブジェクトを返す __ite
スが __next__() メソッドを定義している場合、 __iter__() メソッドは

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
```

データモデル - Python言語リファレンス - Python 3.6.1

object.__reversed__(*self*) (原文)

reversed() 組み込み関数が逆方向イテレーションを実装するために、(存在すれ
内の全要素を逆順にイテレートする、新しいイテレータを返すべきです。

__reversed__() メソッドが定義されていない場合、 reversed() 組み込み関数は sequence プロトコル (__len__() と __getitem__()) を使った方法にフォールバックします。 sequence プロトコルをサポートしたオブジェクトは、 reversed() よりも効率のいい実装を提供できる場合にのみ __reversed__() を定義するべきです。

帰属テスト演算子 (in および not in) は通常、シーケンスに渡る反復処理を使って実装されます。しかし、コンテナオブジェクトで以下の特殊メソッドを定義して、より効率的な実装を行ったり、オブジェクトがシーケンスでなくてもよいようにできます。

object.__contains__(*self*, *item*) (原文)

帰属テスト演算を実装するために呼び出されます。 *item* が *self*
には偽を返さなければなりません。マップオブジェクトの場合
帰属テストを考えなければなりません。

container.__iter__() (原文)

イテレータオブジェクトを返します。オブジェクトは後述するイテレータプロトコルをサポートする必要があります。もしコンテナが異なる型の反復処理をサポートするなら、それらの反復処理毎に追加のメソッドを提供しても構いません (複数の形式の反復処理を提供するオブジェクトの例として、幅優先探索と深さ優先探索をサポートする木構造が挙げられます)。このメソッドは Python/C API での Python オブジェクトの型構造体の tp_iter スロットに対応します。

イテレータオブジェクト自体は以下の 2 つのメソッドをサポートする必要があります。これらのメソッドは 2 つ合わせて *iterator protocol* (イテレータプロトコル) を成します:

iterator.__iter__() (原文)

イテレータオブジェクト自体を返します。このメソッドはコンテナとイテレータの両方を for および in 文で使えるようにするために必要です。このメソッドは Python/C API において Python オブジェクトを表す型構造体の tp_iter スロットに対応します。

iterator.__next__() (原文)

コンテナの次のアイテムを返します。もしそれ以上アイテムが無ければ StopIteration 例外を送出します。このメソッドは Python/C APIでのPythonオブジェクトの型構造体の tp_iternext スロットに対応します。

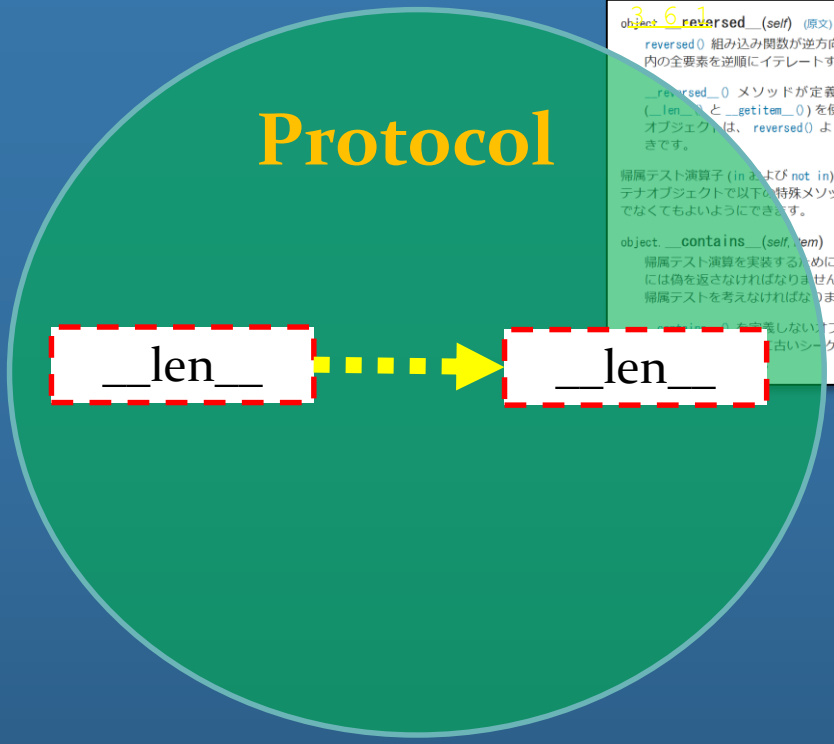
Python では、いくつかのイテレータオブジェクトを定義して、一般のシーケンス型、特殊なシーケンス型、辞書型、その他の特殊な形式に渡って反復をサポートしています。特殊型は、イテレータプロトコルの実装以外では重要ではありません。

イテレータの __next__() メソッドが一旦 StopIteration を送出したなら、以降の呼び出しでも例外を送出し続けなければなりません。この特性に従わない実装は壊れているとみなされます。

4.5.1. ジェネレータ型 (原文)

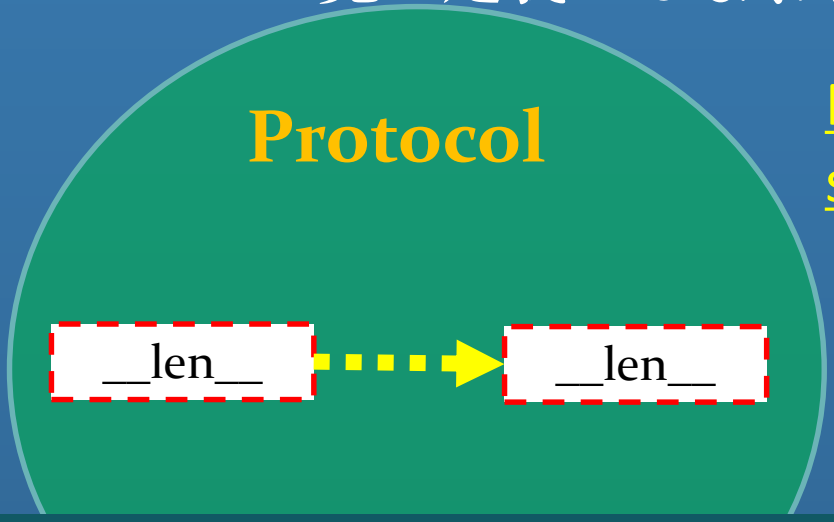
Python における generator (ジェネレータ) は、イテレータプロトコルを実装する便利な方法を提供します。コンテナオブジェクトの __iter__() メソッドがジェネレータとして実装されていれば、そのメソッドは __iter__() および __next__() メソッドを提供するイテレータオブジェクト (厳密にはジェネレータオブジェクト) を自動的に返します。ジェネレータに関する詳細な情報は、 yield 式のドキュメント にあります。

組み込み型 -Python 標準ライブラリ - Python 3.6.1



Protocolの定義はどこにあるの？

- Protocolの一覧などは（現在も）なさそう…
- ドキュメントへの初登場はPython-2.2 (2001年)
- PEP-544 (2017年5月)でPEPに初登場
  - Protocol一覧の定義ではなく、型ヒントのため明確化



PEP 544 -- Protocols: Structural subtyping (static duck typing)

PEPはPython拡張提案(Python Enhancement Proposal)を表しています。PEPはPythonのコミュニティに対して情報を提供したり、Pythonの新機能やプロセス、環境などを説明するための設計書です。PEPは、技術的な仕様と、その機能が必要な論理的な理由を提供しなければなりません。

— PEP1 より

Protocol一覧の代わりに

- http://docs.python.jp/3/library/collections.abc.html

8.4. collections.abc — コレクションの抽象基底クラス (原文)

バージョン *3.3* で追加: 以前はこのモジュールは collections モジュールの一部でした。

ソースコード: Lib/_collections_abc.py

このモジュールは、 抽象基底クラス を提供します。抽象基底クラスは、クラスが特定のインタフェースを提供しているか、例えばハッシュ可能であるかやマッピングであるかを判定します。

8.4.1. コレクション抽象基底クラス (原文)

collections モジュールは以下の ABC (抽象基底クラス) を提供します:

ABC	継承しているクラス	抽象メソッド	mixin メソッド
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	Iterable	`__next__`	`__iter__`
Reversible	Iterable	`__reversed__`	
Generator	Iterator	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized		`__len__`	
Callable		`__call__`	

ここまでのまとめ

len() は Adapter

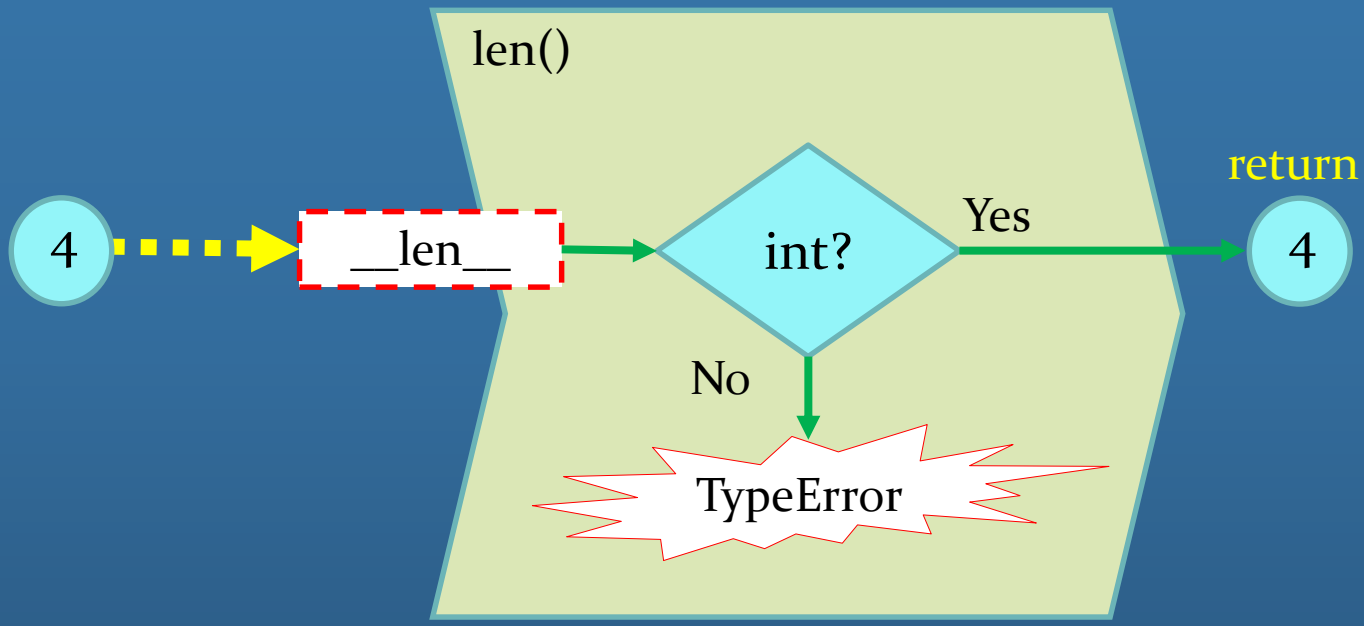
オブジェクトとAdapterが
通信する規約がプロトコル

Adapter、値のチェックしてるだけでしょう？

こういうメリットもあるよ

len() 、 max() 、 min() を組み込み関数として実装することで、それぞれの型のメソッドとして実装するより少ないコードで済みます。

— デザインと歴史 FAQ より



次はもうちょっと複雑な例

(´・ω・`)ノ

if がオブジェクトのTrue/Falseを判断する方法

if 文のルール

- if文の例

```
if obj:
    print(“Trueだ!”)
else:
    print(“Falseだ!”)
```

- 内部では自動的にbool()で変換されます

```
if bool(obj):
```

- はい、bool() 関数 Adapter です。

bool()関数に数値を渡したときに起こること

- bool関数で数値の真偽(True/False)を判別

```
>>> bool(123)
True
```

- obj.__bool__() メソッドでしょ？

```
>>> (123).__bool__()
True
```

- はい(´･ω･`)
- じゃあ次は文字列で。

bool()関数に文字を渡したときに起こること

- bool関数で文字列の真偽(True/False)を判別

```
>>> bool(“もじれつ”)
True
```

- obj.__bool__() メソッド

```
>>> “もじれつ”.__bool__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no
attribute '__bool__'
```

- あれっ？

数値や文字を bool() に変換するRule

偽と見なされる条件

クラスが __bool__() または __len__() メソッドを定義していれば、それらのメソッドが整数 0 または bool 値 False を返すとき。

真と見なされる条件

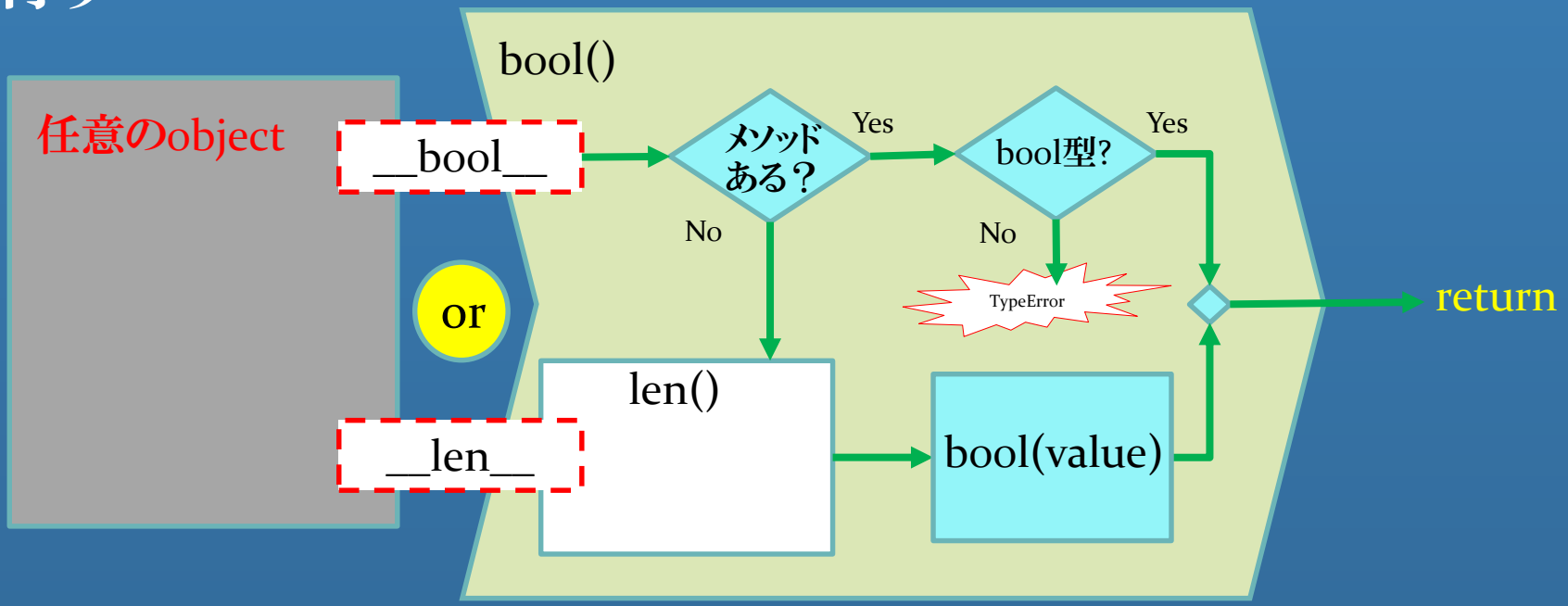
偽じゃないやつ

— 真偽値判定 より

bool() は len() よりも仕事してそう

bool() Adapter

- __bool__ がない場合は、bool(len(obj)) 相当の処理を行う



CPython での bool() の実装コード
https://github.com/python/cpython/blob/master/Objects/typeobject.c#L6081-L6127

独自のデータ型に bool() Adapter

- __bool__() メソッドを実装した独自クラスを定義します

```
>>> class PositiveInt(int):
...     def __bool__(self):
...         return self > 0
...
>>> bool(PositiveInt(10))
True
>>> bool(PositiveInt(-3))  # 0以下の値はFalse
False
>>> bool(-3)  # 本来のintはマイナス値もTrue
True
```

- これは、正の整数ならTrueと判定されるint型です

さらにレベル上げていくよー

(`・ω・´)

for がオブジェクトの繰り返し要素を取得する方法

for 文のルール

- for 文の例

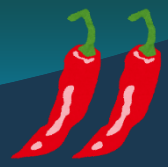
```
for o in obj:
    print(o)
```

- 内部では自動的にiter()で変換されます

```
for o in iter(obj):
```

- はい、iter() 関数 **Adapter** です。

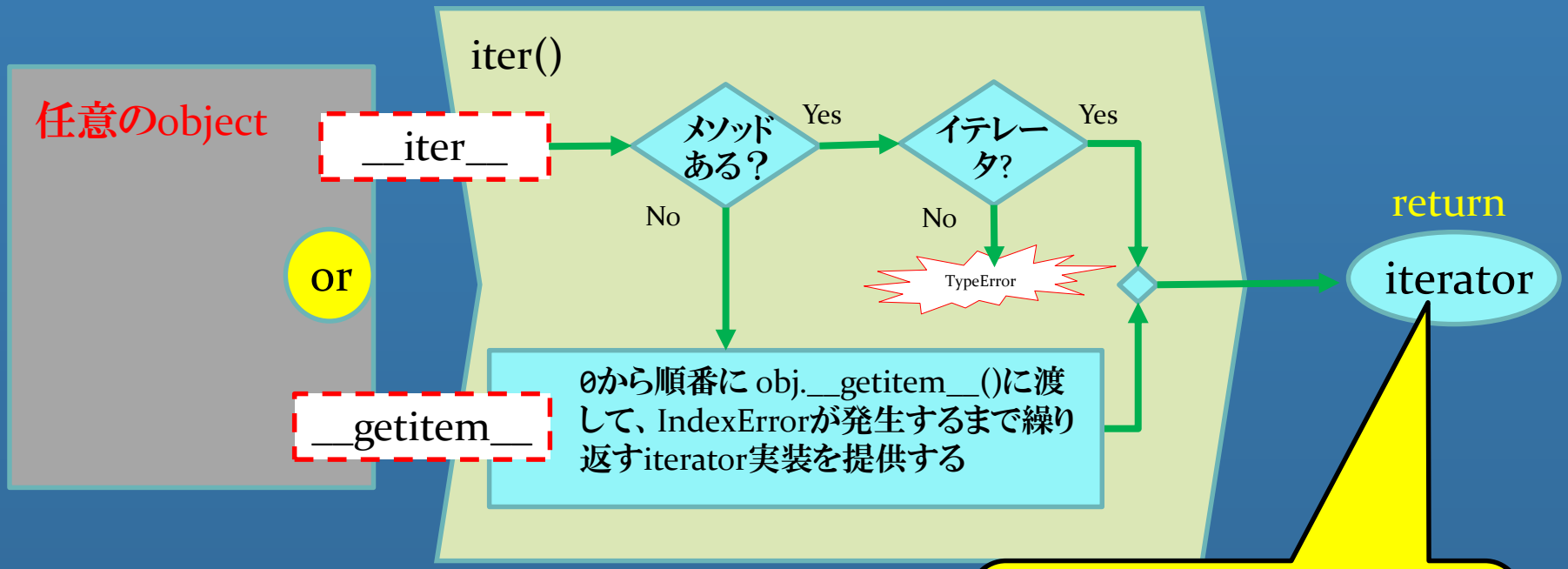
object を iter() に変換するルール



> iter(object) は イテレータ (iterator) オブジェクトを返します。object は反復プロトコル (__iter__() メソッド) か、シーケンスプロトコル (引数が 0 から開始する __getitem__() メソッド) をサポートする集合オブジェクトでなければなりません。これらのプロトコルが両方ともサポートされていない場合、TypeError が送出されます。
>
> — 組み込み関数 iter() より

bool() よりずっと大変そう

iter() Adapter

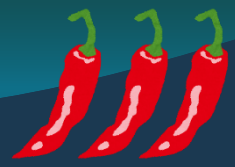


参考: Wikipedia

参考: 実践Python3

Iterator Pattern!!

iter() が返すIteratorとは



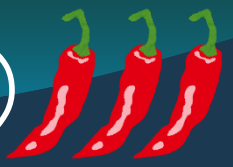
イテレータ(iterator)は、データの流れを表現するオブジェクトです。イテレータの __next__() メソッドを繰り返し呼び出す (または組み込み関数 next() に渡す) と、流れの中の要素を一つずつ返します。データがなくなると、代わりに StopIteration 例外を送出します。

— 組み込み関数 iter() より

イテレータオブジェクト自体は以下の 2 つのメソッドをサポートする必要があります。これらのメソッドは 2 つ合わせて iterator protocol: (イテレータプロトコル) を成します ... __next__(), __iter__()

— 用語集 iterator より

for 文のルール（もうちょっと正確に）



- for 文の例

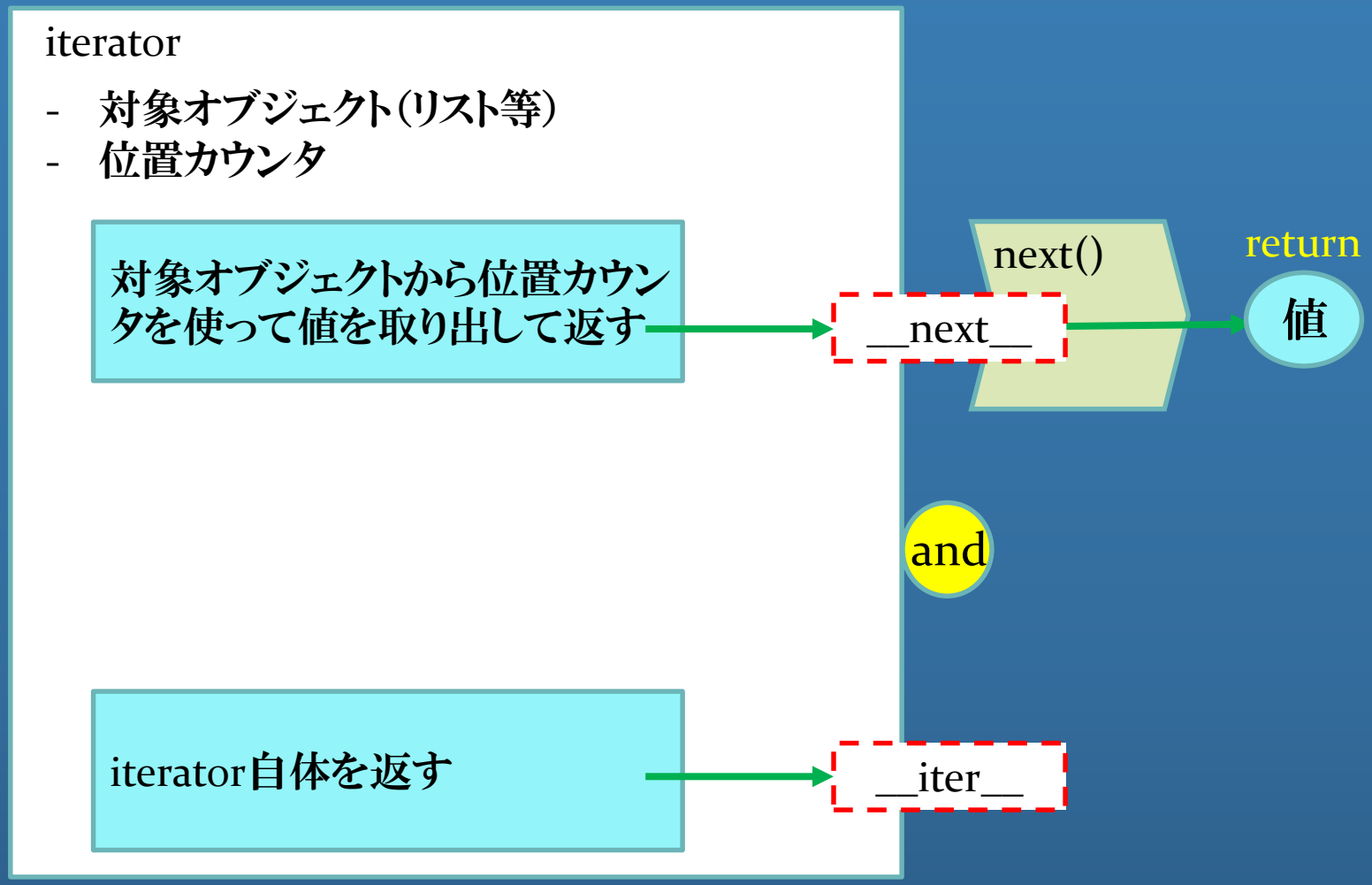
```
for o in obj:
    print(o)
```

- 内部ではこう解釈されます

```
it = iter(obj)
while True:
    try:
        o = next(it)
    except StopIteration:
        break
    print(o)
```

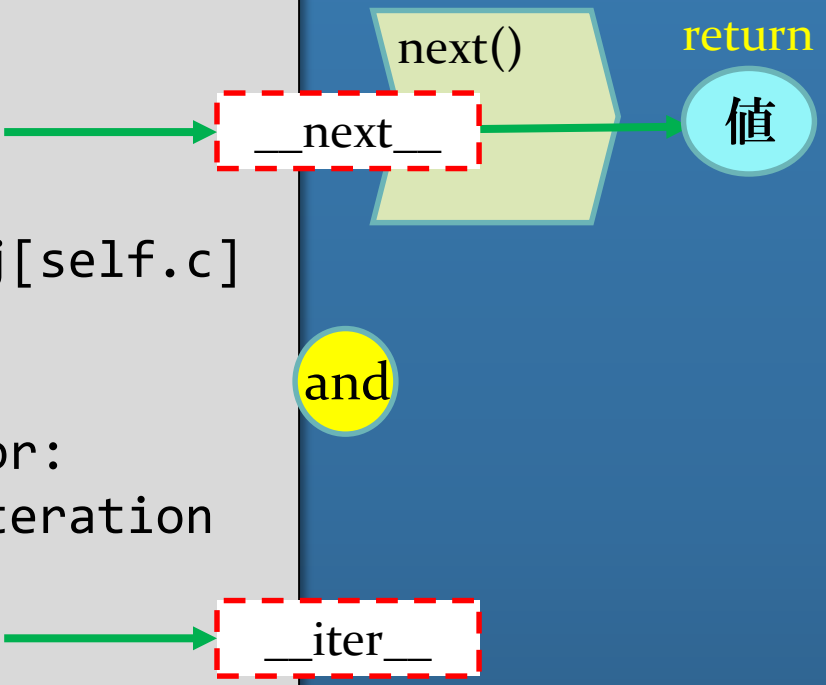
はい、iter() ~~関数~~ **Adapter** と next() ~~関数~~ **Adapter**です。

next() Adapter と iterator



iteratorの実装例

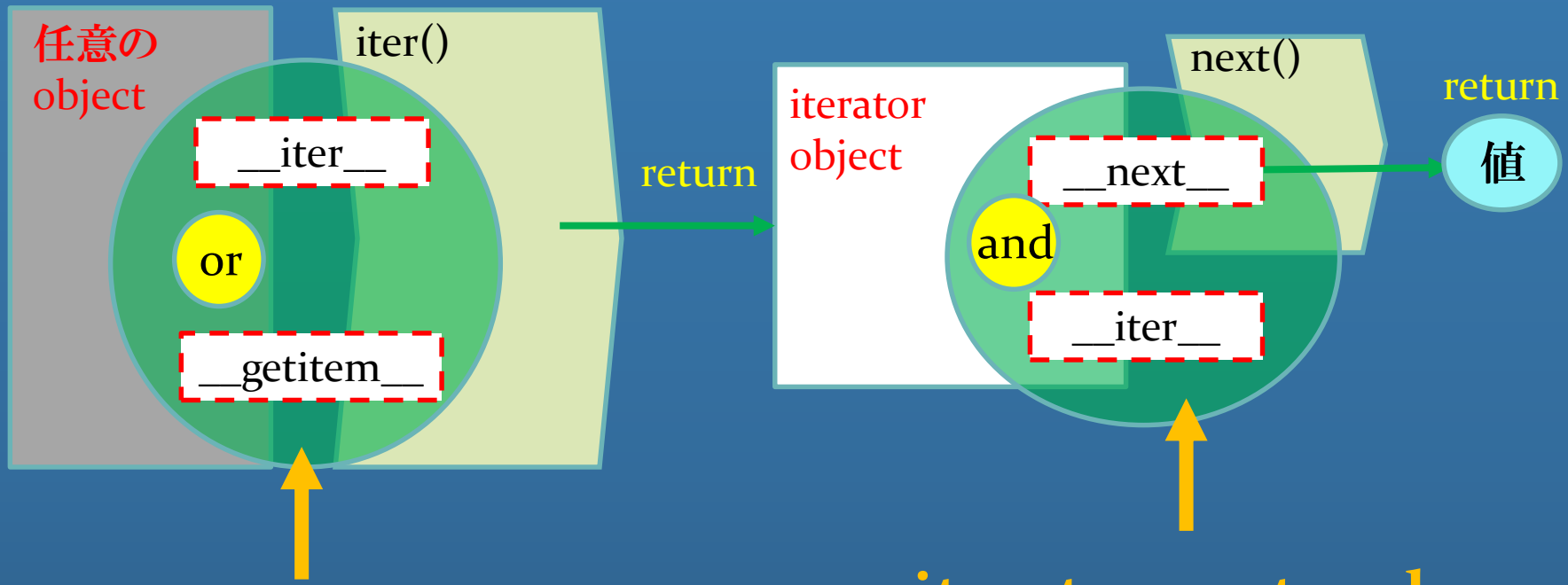
```
class MyIterator:
    def __init__(self, obj):
        self.obj = obj
        self.c = 0

    def __next__(self):
        try:
            r = self.obj[self.c]
            self.c += 1
            return r
        except IndexError:
            raise StopIteration

    def __iter__(self):
        return self
```



for 文の1行目で起こっていること

```
for o in obj:
```

この1行で色々起きてます



Protocol?

特に名称はなさそう(´・ω・`)

iterator protocol

組み込み型 イテレータ型 より

独自のデータ型に iter(), next() Adapte

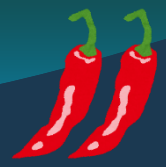
- __iter__() メソッドを実装した独自クラスを定義します

```
class MyContainer:
    def __init__(self, mapping):
        self.keys = sorted(mapping) # ソートして保持
        self.mapping = mapping      # 値返し用

    def __iter__(self):             # for文で呼ばれる
        return MyIterator(self)

    def __getitem__(self, idx):     # MyIteratorから呼ばれる
        return self.mapping[self.keys[idx]]
```

- このコンテナをforに与えると、辞書のキーのアルファベット順に、そのキーの値が繰り返されます

```
>>> list(MyContainer({'foo': 1, 'bar': 2, 'poke': 3, 'ah': 4}))
[4, 2, 1, 3]
```

Iterator Protoclの抽象基底クラス



- http://docs.python.jp/3/library/collections.abc.html

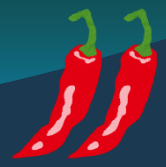
8.4.1. コレクション抽象基底クラス (原文)

collections モジュールは以下の ABC (抽象基底クラス) を提供します:

ABC	継承しているクラス	抽象メソッド	mixin メソッド
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	Iterable	__next__	__iter__
Reversible	Iterable	__reversed__	

- 「抽象基底クラス」を継承して、Protocol実装を強制できます
- abc はそのためのモジュール

継承によるProtocolの強制



- 継承によるInterfaceの強制

```
from collections.abc import Iterator

class MyIterator(Iterator):
    pass

>>> MyIterator()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MyIterator
with abstract methods __next__
```

- collection.abcを継承すれば実装忘れは防げる
- 対応Protocolを明示したい場合にも良い

Explicit is better than implicit.

— PEP 20 -- The Zen of Python より

(´・ω・)おつかれさま(・ω・`)

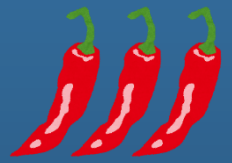
まとめ

まとめ

- len() は Adapter Pattern
  - って言われると納得感あるよね（あるよね？）
- len() ひとつ見ても、多くの経緯と議論の歴史がある
  - len() vs .length の議論を頭良い人達がしてないわけない
  - 歴史に学ぼう
- 初級から中級へ進むには
  - ひたすら情報を読む、歴史を追う
  - 自分なりに解釈する
  - Pythonで色々実装してみる

公式リファレンスに
多くの情報が載っている

原典を調べよう



PEPを読んでみよう

References: Python公式リファレンス

- デザインと歴史 FAQ - Python 3.6.1 ドキュメント - 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？ https://docs.python.jp/3/faq/design.html#why-does-python-use-methods-for-some-functionality-e-g-list-index-but-functions-for-other-e-g-len-list

References: Python公式リファレンス

- Python-2.4 ライブラリリファレンス イテレータ型 http://docs.python.jp/2.4/lib/typeiter.html
  - バージョン 2.2 で追加
- 8.3. collections — 高性能なコンテナ・データ型 - Python 2.6ja2 documentation http://docs.python.jp/2.6/library/collections.html#abcs-abstract-base-classes
  - バージョン 2.6 で変更: 抽象基底クラス (abstract base class) の追加
- 2. 組み込み関数 - Python 3.6.1 ドキュメント http://docs.python.jp/3/library/functions.html#iter
- 4. 組み込み型 - Python 3.6.1 ドキュメント http://docs.python.jp/3/library/stdtypes.html#iterator-types
- 8.4. collections.abc — コレクションの抽象基底クラス - Python 3.6.1 ドキュメント http://docs.python.jp/3/library/collections.abc.html

References: PEP

- PEP 1 -- PEP Purpose and Guidelines | Python.org
  http://sphinx-users.jp/articles/pep1.html (和訳)
  https://www.python.org/dev/peps/pep-0001

- PEP 3119 -- Introducing Abstract Base Classes | Python.org
  https://www.python.org/dev/peps/pep-3119/

- PEP 544 -- Protocols: Structural subtyping (static duck typing) | Python.org
  https://www.python.org/dev/peps/pep-0544/

- PEP 20 -- The Zen of Python
  http://d.hatena.ne.jp/nishiohirokazu/20120317/1331989155 (和訳)
  https://www.python.org/dev/peps/pep-0020/

References: blog等

- len が関数になっている理由 - methaneのブログ
  http://methane.hatenablog.jp/entry/20090702/1246556675
- len が py3k でも 関数のままである理由 - methaneのブログ
  http://methane.hatenablog.jp/entry/20090721/1248195293
- Solid Snakes or: How to Take 5 Weeks of Vacation - Hynek Schlawack
  https://hynek.me/talks/reliability/
- オブジェクト指向と20年戦ってわかったこと – Qiita
  http://qiita.com/shibukawa/items/2698b980933367ad93b4
- 新人プログラマに知っておいてもらいたい人類がオブジェクト指向を手に入れるまでの軌跡 – Qiita
  http://qiita.com/hirokidaichi/items/591ad96ab129388878fe1
- Python を支える技術 ディスクリプタ編 #pyconjp – Qiita
  http://qiita.com/knzm/items/a8a0fead6e1706663c22
- The History of Python.jp: ユーザ定義クラスのサポートの追加
  http://python-history-jp.blogspot.jp/2009/04/blog-post_30.html
- 仮想継承とsingledispatch – atsuoishimoto's diary
  http://atsuoishimoto.hatenablog.com/entry/2016/08/04/095641
- The Zen of Python 解題 - 前編 – atsuoishimoto's diary
  http://atsuoishimoto.hatenablog.com/entry/20100920/1284986066

References: CPython code

- bool()実装
  https://github.com/python/cpython/blob/1f06a680d/Objects/typeobject.c#L6081-L6127
- 真偽判定実装
  https://github.com/python/cpython/blob/1f06a680d/Objects/object.c#L1314-L1336
- len()実装
  https://github.com/python/cpython/blob/1f06a680d/Objects/typeobject.c#L5920-L5944
- len()のint値判定で呼ばれる実装
  https://github.com/python/cpython/blob/1f06a680d/Objects/abstract.c#L1238-L1275

References: その他

- 実践 Python3
  http://amzn.to/2vK2uHl
  - Adapter Pattern, Iterator Pattern, Protocol, ...
- 結城浩にインタビュー Java言語で学ぶデザインパターン入門
  http://www.hyuki.com/dp/interview.html
- かわいいフリー素材集 いらすとや
  http://www.irasutoya.com/

Questions?

@shimizukawa

Grab me anytime :)

Break, Party, Sprint

Thanks :)