2022年10月31日(月) 4限
@研究棟A302

プログラミングA2
第5回

担当：伏見卓恭
連絡先：fushimity@edu.teu.ac.jp
居室：研A1201

プログラミングA2の流れ

第 1回：＜復習編＞関数，ファイル入出力，コンテナデータ型
第 2回：＜復習編＞クラスとオブジェクト
第 3回：＜文法編＞関数の高度な利用法１
第 4回：＜文法編＞関数の高度な利用法２
第 5回：＜文法編＞オブジェクト指向プログラミング
第 6回：＜応用編＞データ構造とアルゴリズム１
第 7回：＜応用編＞データ構造とアルゴリズム２
第 8回：＜実践編＞HTTPクライアント
第 9回：＜実践編＞スクレイピング
第10回：＜実践編＞データベース
第11回：＜実践編＞並行処理
第12回：＜総合編＞総合演習(複合問題)
第13回：＜総合編＞まとめ
第14回：＜総合編＞ Python力チェック ← 確認テストのこと

本日のお品書き

- 前回課題の振り返りとパイソニック**でない**書き方

- オブジェクト指向プログラミング
  - 特殊メソッドと特殊属性のおさらい
  - ディスクリプタ
  - 抽象基底クラス

変数にどんな値が入っている？

- **変数名が何を表しているか不明**

```
for i in self.source.titles:
    if i[0] == 何か
```

- `i` って何やねん？
- 慣習として，`i, j, k` は `int`の値が順に入るような変数として使われる

```
for i in range(10):
```

- `titles`というリストの`1`つの要素なら`title`が無難

```
for title in self.source.titles:
    if title[0] == 何か
```

- そうすると，`title[0]`がタイトルの最初の文字だと一目瞭然

リストが空かどうか？

```
lst = []
```

- notを使う `if not lst:`
- len()を使う `if len(lst) == 0:`
- []を使う `if lst == []:`
  - 上2つの方法は，lstがタプルでも集合でも通用する
  - これは，リストであることも確認している
  - → 真に，空のリストか否かは確認している

※前回，この方法はあまりよくないと言ってしまいました🙇

特殊メソッド
特殊属性

いままで出てきた特殊メソッド

- `__init__()`：クラス名()でインスタンスを生成する時
- `__str__()`：str(), print(), format()に渡す時
- `__repr__()`：repr(), print(), format()に渡す時
- `__iter__()`：iter()に渡す時
  - for-in構文でも，自動でiter()に渡される
- `__next__()`：next()にイテレータを渡す時
- `__len__()`：len()に渡す時
- `__eq__()`：== で比較する時　← 演算子のオーバーロード
- `__gt__()`：> で比較する時　← 演算子のオーバーロード
- `__getitem__()`：[idx]でシーケンスにアクセスする時
- `__call__()`：()でcallableを呼び出す時

いままで出てきた特殊属性

- __name__：
  - そのまま：トップレベルのコードが実行される名前空間
    - モジュールとしてimportされた時：モジュール名
    - CLからスクリプトとして実行された時：__main__
  - __class__.__name__：クラス名
  - 関数オブジェクト.__name__：関数名，メソッド名
- __class__：インスタンスが属するクラス
  - type()関数により取得可能
- __dict__：クラスやインスタンスの**属性辞書**
  - vars()関数により取得可能
- __doc__：docstring（関数の説明など）
  - help()関数により取得可能

__name__を確かめてみよう

pokemon.py

```
print(f"{__file__}の{__name__}(ifの外)") ①

class Monster:
    def __init__(self):
      ② print(f"{__class__.__name__}クラスの{__class__.__init__.__name__}関数")

if __name__ == "__main__":
  ③ print(f"{__file__}の{__name__}(ifの中)")
    m = Monster()
```

selfも可能

special.py

```
from pokemon import Monster

if __name__ == "__main__":
   ④ print(f"{__file__}の{__name__}(ifの中)")
      m = Monster()
```

実行例

```
$ python pokemon.py
① pokemon.pyの__main__(ifの外)
③ pokemon.pyの__main__(ifの中)
② Monsterクラスの__init__関数
```

←pokemon.pyがスクリプトとして実行された

実行例

```
$ python special.py
① pokemon.pyのpokemon(ifの外)
④ special.pyの__main__(ifの中)
② Monsterクラスの__init__関数
```

←pokemon.pyがモジュールとしてimportされた
∴③は実行されない

__dict__を確かめてみよう

pokemon.py

```
class Monster:
    num_of_monsters = 0
    def __init__(self, title, type_, level, stats):
        print(f"{__class__.__name__}クラスの{__class__.__init__.__name__}関数")
        self.title = title
        self.type_ = type_
        self.level = level
        self.stats = stats
        __class__.num_of_monsters += 1
```

special.py

```
if __name__ == "__main__":
    print(f"{__file__}の{__name__}(ifの中)")
    m = Monster("ピカチュウ", "でんき", 30, [])
    pprint(Monster.__dict__)
    pprint(m.__dict__)
```

実行例

```
$ python lec05/special.py
mappingproxy({'__dict__': <attribute '__dict__' of 'Monster' objects>,
              '__doc__': None,
              '__init__': <function Monster.__init__ at 0x7f2d661246a8>,
              '__module__': 'pokemon',
              '__weakref__': <attribute '__weakref__' of 'Monster' objects>,
              'num_of_monsters': 1})

{'level': 30, 'stats': [], 'title': 'ピカチュウ', 'type_': 'でんき'}
```

※クラスやインスタンスは辞書形式で属性を管理している
(int, list, strなど, __dict__を持たないクラスもある)

【発展】属性の探索順序

属性へのアクセスがあると，

1. **クラス**の属性辞書から該当する要素を探索する．
   - **__get__()メソッドと__set__()メソッドの両方を持っていれば，__get__()メソッドの戻り値が返される．**
2. **オブジェクト**の属性辞書から該当する要素を探索する．
   - 要素が存在すればその値が返される．
3. **クラス**の属性辞書が探索される．
   - 該当する要素が存在し，その要素が
     - **__get__()メソッドを持っていれば，__get__()メソッドの戻り値が返される．**
     - __get__()メソッドを持っていなければ，その要素そのものが返される．
4. 同様の探索が，継承を辿ってすべての親クラスの属性辞書で行われる．
5. それでも存在しなければ，クラスが__getattr__()メソッドを持っているかチェックされ，
   - __getattr__()メソッドを持っていれば，__getattr__()メソッドの戻り値が返される．
   - __getattr__()メソッドを持っていなければ，AttributeErrorがraiseされる．

属性を操作する組み込み関数

- **getattr(オブジェクト，属性名)：属性値の取得**
  - `オブジェクト.属性` と等価
  - `オブジェクト.__getattr__(self, 属性名)` と等価
- **setattr(オブジェクト，属性名，値)：属性に値を設定**
  - `オブジェクト.属性＝値` と等価
  - `オブジェクト.__setattr__(self, 属性名, 値)` と等価
- **delattr(オブジェクト，属性名)：属性の削除**
  - `del オブジェクト.属性` と等価
  - `オブジェクト.__delattr__(self, 属性名)` と等価
- **hasattr(オブジェクト，属性名)：属性の有無を確認**
- **dir(オブジェクト)：オブジェクトの属性リスト**
  - `オブジェクト.__dir__` と等価

※属性**名**と記述しているものは，文字列です

もしかして知らない人もいるかも

- Pythonでは，すべてのものがオブジェクト
  ＝何らかのクラスのインスタンス

- intクラスのインスタンス
  - 243
- floatクラスのインスタンス
  - 172.5
- listクラスのインスタンス
  - [1, 2, 0, 1]
- strクラスのインスタンス
  - "fsm"
- typeクラスのインスタンス
  - int
  - Monster
- functionクラスのインスタンス
  - lambda : 27
  - def …

instance.py

```
a = 243
print(type(a))

a = 172.5
print(type(a))

a = [1, 2, 0, 1]
print(type(a))

a = "fsm"
print(type(a))

a = int
print(type(a))

a = lambda : 27
print(type(a))
```

実行例

```
<class 'int'>

<class 'float'>

<class 'list'>

<class 'str'>

<class 'type'>

<class 'function'>
```

組み込み型：https://docs.python.org/ja/3/library/stdtypes.html

もしかして知らない人もいるかも

- Pythonでは，すべてのものがオブジェクト
  ＝何らかのクラスのインスタンス

- Monsterクラスのインスタンス
  - pika
- methodクラスのインスタンス
  - pika.__init__

instance.py

```
class Monster: # 自作クラス
    def __init__(self, title):
        self.title = title

a = Monster
print(type(a)) # 自作クラス
print(type(a.__init__)) # （クラス経由）

a = Monster("ピカチュウ") # 自作クラスのインスタンス
print(type(a))
print(type(a.__init__)) # （インスタンス経由）
```

実行例

```
<class 'type'>
<class 'function'>


<class '__main__.Monster'>
<class 'method'>
```

ディスクリプタ

propertyのデメリット：1つの属性にしか適用できない
→複数の属性，異なるクラスの属性に適用する場合は，
その数の分だけゲッタとセッタを実装し
@propertyを付す必要がある

【再掲】プロパティproperty

●プロパティproperty

プロパティとは，クラス外部からはインスタンス変数のように使用でき，
クラス内部ではメソッドのように実装した属性のこと

- ゲッタgetter：属性の値を取得する場合

```
@property
def 属性(self):
    return self._属性
```

```
値 = インスタンス.属性
```

- セッタsetter：属性に値を設定する場合

```
@属性.setter
def 属性(self, パラメータ):
    self._属性 = パラメータ
```

```
インスタンス.属性 = 値
```

※セッタを定義しなかった場合は
読み取り専用の属性になる

変数自体は，プロパティと別の名前にする
慣習的に，シングルアンダースコア「_」を前につける

【再掲】プロパティによるアクセス

pokemon.py

```
@property
def level(self):
    return self._level

@level.setter
def level(self, val):
    if val < 0:
        print("不正な値のため，設定を中止します")
    else:
        self._level = val
```

← ゲッタ

← セッタ

get_monsters.py

```
省略
print(monsters[0].level)
monsters[0].level = -10
```

実行例

```
省略
5
不正な値のため，設定を中止します
```

ディスクリプタ

複数の属性に対して，同じアクセスロジックを再利用するための方法を提供するクラスである．
ディスクリプタを使うと，属性がアクセスされた際の動作をカスタマイズできる．
Pythonの至るところで使用される非常に強力な多目的のプロトコルでありプロパティ，関数，メソッド，クラスメソッド，静的メソッド，super()の背後でも使われている．

- Pythonの奥義を極めるには、ディスクリプタを理解しなければなりません。- 20章 属性ディスクリプタ - Fluent Python

ディスクリプタクラスの定義

```
class ディスクリプタ:
    def __set_name__(self, owner, attrname):

    def __get__(self, obj, objtype=None):

    def __set__(self, obj, value):

    def __delete__(self, obj):
```

← ディスクリプタがクラスに設置された時に呼び出される

← 対象の属性が参照される時に呼び出される

← 対象の属性が更新される時に呼び出される

← 対象の属性が削除される時に呼び出される

ディスクリプタの利用

ディスクリプタ利用による属性の定義

```
class クラス:
    属性 = ディスクリプタ()
    属性 = ディスクリプタ()
```

← **クラス属性にディスクリプタを設置**
__set_name__(クラス, 属性名)が呼び出される

↑ **属性ごとにディスクリプタを設置**
それぞれのディスクリプタオブジェクトが,
インスタンス変数として属性名を持っていれば,
以下のゲッタとセッタが呼び出される時に属性名がなくても判別できる

ディスクリプタを介した属性へのアクセス

```
インスタンス = クラス()

print(インスタンス.属性)



インスタンス.属性 = 値
```

← **属性が参照される**
__get__(インスタンス, クラス)が呼び出される

← **属性が更新される**
__set__(インスタンス, 値)が呼び出される

コード例①

Monsterクラスのインスタンス
Monsterクラスの属性
属性に設定する値
ディスクリプタ
属性名文字列

●属性をディスクリプタで記述する対象のクラス

descriptor1.py：ディスクリプタ設置部

```
class Monster:
    title = PokeDesc()
    level = PokeDesc()

    def __init__(self, title):
        self.title = title
```

← クラス属性として

← イニシャライザで名前(title)だけ設定
レベルは後で設定

●ディスクリプタクラス

descriptor1.py：ディスクリプタ定義部

```
class PokeDesc:
    def __get__(self, obj, objtype=None):
        print("参照")
        return self.value

    def __set__(self, obj, value):
        self.value = value
        print("更新")
```

← 各クラス属性に紐づいたディスクリプタが
内部変数valueにその値を保持

コード例①

Monsterクラスのインスタンス
Monsterクラスの属性
属性に設定する値
ディスクリプタ
属性名文字列

descriptor1.py：属性参照・更新部

```
if __name__ == "__main__":
    fushi = Monster("フシギダネ")
    pika  = Monster("ピカチュウ")
    fushi.level = 25
    print("-"*20)
    pprint(fushi.__dict__)
    print(fushi.title, fushi.level)


    print("-"*20)
    pprint(pika.__dict__)
    print(pika.title)
```

実行例

```
$ python lec05/descriptor1.py
更新
更新
更新
----------
{}
参照
参照
ピカチュウ 25
----------
{}
参照
ピカチュウ
```

クラス属性にディスクリプタを設置
← インスタンスの属性辞書は空

← fushiインスタンスからtitleを見ても"ピカチュウ"

※各Monsterインスタンスに紐づく属性（インスタンス変数）ではなく，Monsterクラス共通の属性に"フシギダネ"や"ピカチュウ"が代入された

※どのインスタンスに紐づく属性に値を代入するのかを判別する必要がある

コード例②

Monsterクラスのインスタンス
Monsterクラスの属性
属性に設定する値
ディスクリプタ
属性名文字列

objにより，どのポケモンインスタンスなのか
attrnameにより，どの属性なのかを判別している例

descriptor2.py：ディスクリプタ定義部

```
class PokeDesc:
    def __init__(self, attrname):
        self.attrname = attrname

    def __get__(self, obj, objtype=None):
        return obj.__dict__[self.attrname]

    def __set__(self, obj, value):
        obj.__dict__[self.attrname] = value
```

descriptor2.py：ディスクリプタ設置部

```
class Monster:
    title = PokeDesc("title")
    level = PokeDesc("level")
```

↑ クラス属性として

```
    def __init__(self, title):
        self.title = title
```

descriptor2.py：属性参照・更新部

```
if __name__ == "__main__":
    fushi = Monster("フシギダネ")
    pika  = Monster("ピカチュウ")
    fushi.level = 25
    print("-"*20)
    pprint(fushi.__dict__)
    print(fushi.title, fushi.level)


    print("-"*20)
    pprint(pika.__dict__)
    print(pika.title)
```

実行例

```
$ python lec05/descriptor2.py
更新：obj.__dict__[self.attrname]='フシギダネ'
更新：obj.__dict__[self.attrname]='ピカチュウ'
更新：obj.__dict__[self.attrname]=25
--------------------
{'level': 25, 'title': 'フシギダネ'}
参照：obj.__dict__[self.attrname]='フシギダネ'
参照：obj.__dict__[self.attrname]=25
フシギダネ 25
--------------------
{'title': 'ピカチュウ'}
参照：obj.__dict__[self.attrname]='ピカチュウ'
ピカチュウ
```

コード例③

Monsterクラスのインスタンス
Monsterクラスの属性
属性に設定する値
ディスクリプタ
属性名文字列

objにより，どのポケモンインスタンスなのか
attrnameにより，どの属性なのかを判別している例

descriptor3.py：ディスクリプタ定義部

```
class PokeDesc:
    def __set_name__(self, owner, attrname):
        self.attrname = attrname
        print(f"設置：{owner.__name__}の{self.attrname}")

    def __get__(self, obj, objtype=None):
        print(f'参照：{obj.__dict__[self.attrname]=}')
        return obj.__dict__[self.attrname]

    def __set__(self, obj, value):
        obj.__dict__[self.attrname] = value
        print(f'更新：{obj.__dict__[self.attrname]=}')
```

self：ディスクリプタ
owner：設置されたクラス
attrname：対応する属性名

descriptor3.py：ディスクリプタ設置部

```
class Monster:
    title = PokeDesc()
    level = PokeDesc()

    def __init__(self, title):
        self.title = title
```

←__set_name__()が呼び出される

←__set__()が呼び出される

実行例

```
$ python lec05/descriptor3.py
設置：Monsterのtitle
設置：Monsterのlevel
```

コード例③

Monsterクラスのインスタンス
Monsterクラスの属性
属性に設定する値
ディスクリプタ
属性名文字列

descriptor3.py：属性参照・更新部

```
print("main開始")
fushi = Monster("フシギダネ")
  Monster.__dict__["title"].__set__(fushi, "フシギダネ")
pika  = Monster("ピカチュウ")
  Monster.__dict__["title"].__set__(pika,  "ピカチュウ")
fushi.level = 25
  Monster.__dict__["level"].__set__(fushi, 25)
print("-"*20)
pprint(fushi.__dict__)
print(fushi.title,
      fushi.level)

  Monster.__dict__["title"].__get__(fushi, Monster)
  Monster.__dict__["level"].__get__(fushi, Monster)
```

実行例

```
main開始
更新：obj.__dict__[self.attrname]='フシギダネ'
更新：obj.__dict__[self.attrname]='ピカチュウ'
更新：obj.__dict__[self.attrname]=25
--------------------
{'level': 25, 'title': 'フシギダネ'}
参照：obj.__dict__[self.attrname]='フシギダネ'
参照：obj.__dict__[self.attrname]=25
フシギダネ 25
```

※クラス内で共通して参照されるクラス属性の値を更新するのではなく
クラス属性に紐づくディスクリプタオブジェクトの内部変数を更新するのでもなく
ディスクリプタを介して対応するインスタンスの属性辞書を更新している

ちなみに，こんなことしちゃうと

descriptor3.py：属性参照・更新部

```
print(Monster.title)
```

```
Monster.__dict__["title"].__get__(None, Monster)
```

```
def __get__(self, None, Monster):
    return None.__dict__["title"]
```

実行例

```
Traceback (most recent call last):
  File "descriptor3.py", line 41, in <module>
    print(Monster.title)
  File "descriptor3.py", line 7, in __get__
    return obj.__dict__[self.attrname]

AttributeError: 'NoneType' object has no attribute '__dict__'
```

アンスコ付けて，内部変数に

__get__()が参照する，__set__()が更新する属性名の前にアンダースコア「_」を付けることで，みなし内部変数にする

descriptor3.py：ディスクリプタ定義部

```
class PokeDesc:
    def __set_name__(self, owner, attrname):
        self.attrname = "_"+attrname

    def __get__(self, obj, objtype=None):
        return obj.__dict__[self.attrname]

    def __set__(self, obj, value):
        obj.__dict__[self.attrname] = value
```

← ここだけ変更

descriptor3.py：属性参照・更新部

```
print(fushi.__dict__)
print(fushi.title, fushi.level)
```

← ディスクリプタの__get__を呼び出している

実行例

```
{'_level': 25, '_title': 'フシギダネ'}
フシギダネ 25
```

← titleとlevelは存在しない

練習問題：pokemon_dsc1.py

Monsterクラスには，ポケモンの名前を表すtitleと，レベルを表すlevelを属性として持っている．

title属性に対して，クラスの内外から
インスタンス.title = 名前 の形式で設定しようとした際，
__set__()メソッドを経由して値を設定するように
ディスクリプタクラスTitleを作成せよ．

level属性に対して，クラスの内外から
インスタンス.level = レベル の形式で設定しようとした際，
__set__()メソッドを経由して値を設定するように
ディスクリプタクラスLevelを作成せよ．

Levelディスクリプタの__get__()メソッドでは，
常に10を返すようにせよ．

解答例：pokemon_dsc1.py

pokemon_dsc1.py

```
class Title:
    def __set_name__(self, owner, attrname):

    def __get__(self, obj, objtype=None):

    def __set__(self, obj, value):
        属性に値を設定


class Level:
    def __set_name__(self, owner, attrname):

    def __get__(self, obj, objtype=None):
        常に10を返す

    def __set__(self, obj, value):
        属性に値を設定
```

実行例：pokemon_dsc1.py

pokemon_dsc1.py

```
class Monster:
    title = Title()
    level = Level()

    def __init__(self, title, level):
        self.title = title
        self.level = level


if __name__ == "__main__":
    fushi = Monster("フシギダネ", 20)
    print(fushi)
    pika = Monster("ピカチュウ", -15)
    print(pika)
    muu = Monster("ミュースリー", 100)
    print(muu)
```

実行例

```
属性titleのディスクリプタを設置
属性levelのディスクリプタを設置
フシギダネ(Lv.10)
ピカチュウ(Lv.10)
ミュースリー(Lv.10)
```

↑
levelとしてどんな値を渡しても，
10が返される

試しに，__dict__もprintしてみよう
属性辞書のlevelには，どんな値が設定されているだろうか？

練習問題：pokemon_dsc2.py

以下の点を**検証する**validate()メソッドを追加せよ．
条件を満たす場合，属性に値を設定するように修正せよ．

- ポケモンの名前(title)は，2文字以上5文字以下である
- ポケモンのレベル(level)は，1以上100以下である

これらの条件を満たさない値が渡された時，値を設定せずにValueErrorをraiseする．

pokemon_dsc2.py

```
class Monster:
    title = Title(2, 5)
    level = Level(1, 100)

if __name__ == "__main__":
    fushi = Monster("フシギダネ", 20)
    print(fushi)
    pika = Monster("ピカチュウ", -15)
    print(pika)
    muu = Monster("ミュースリー", 100)
    print(muu)
```

← 文字数の規定範囲を指定
← レベルの規定範囲を指定

実行例

```
属性titleのディスクリプタを設置
属性levelのディスクリプタを設置
フシギダネ(Lv.10)
ValueError: レベル範囲規定違反
ValueError: 文字数範囲規定違反
```

解答例：pokemon_dsc2.py

pokemon_dsc2.py　　青字追加箇所

```
class Title:
    イニシャライザで規定範囲の設定

    def __set__(self, obj, value):
        self.validate(value)
        obj.__dict__[self.attrname] = value

    def validate(self, value):
        if not (self.minval <= len(value) <= self.maxval):
            例外の送出

class Level:
    イニシャライザで規定範囲の設定

    def __set__(self, obj, value):
        self.validate(value)
        obj.__dict__[self.attrname] = 10

    def validate(self, value):
        if not (self.minval <= value <= self.maxval):
            例外の送出
```

← validate(値)で値を検証
※規定範囲外の場合は例外がraiseされる

← validate(値)で値を検証
※規定範囲外の場合は例外がraiseされる

例題：pokemon_dsc3.py

ディスクリプタクラスTitleとLevelには，共通のメソッド__set_name__()，__get__()，__set__()を持つので，
これらを持つValidatorクラスを定義し，
TitleクラスとLevelクラスはValidatorクラスを継承するように修正せよ.

例題：pokemon_dsc3.py

pokemon_dsc3.py

青字追加箇所

```
class Validator:
    def __init__(省略):
        省略
    def __set_name__(省略):
        省略
    def __get__(省略):
        省略
    def __set__(self, obj, value):
        self.validate(value)
        obj.__dict__[self.attrname] = value


class Title(Validator):
    def validate(self, value):
        if not (self.minval <= len(value) <= self.maxval):
            raise ValueError("文字数範囲規定違反")


class Level(Validator):
    def validate(self, value):
        if not (self.minval <= value <= self.maxval):
            raise ValueError("レベル範囲規定違反")
```

← TitleとLevelに共通するメソッドを移動

← validate()メソッドは
Validatorクラスでは定義されていない
※Validatorクラスのインスタンスは
validate()メソッドを使えない

← 子クラスにない属性（メソッド）は，
親クラスのものを使用する

検証：pokemon_dsc3.py

pokemon_dsc3.py　　青字変更箇所

```
class Monster:
    title = Validator(2, 5)
    level = Validator(1, 100)

if __name__ == "__main__":
    fushi = Monster("フシギダネ", 20)
    print(fushi)
    pika = Monster("ピカチュウ", -15)
    print(pika)
    muu = Monster("ミュースリー", 100)
    print(muu)
```

validate()メソッドは
Validatorクラスでは定義されていない
※Validatorクラスのインスタンスは
validate()メソッドを使えない

実行例

```
AttributeError: 'Validator' object has no attribute 'validate'
```

抽象基底クラス(Abstract Base Class)

抽象基底クラスは，それ自体がインスタンス化することはなく，
他のクラスに継承されることを前提とした基底クラス(親クラス)である．
抽象基底クラスを使うことで，サブクラスのメソッド名を共通化できる．
@abstractmethodが付されたメソッドは，抽象メソッドという．

- 抽象メソッドをもつ抽象基底クラスは，インスタンス化できない．
- 抽象メソッドをもつ抽象基底クラスを継承したクラスは，
  必ず抽象メソッドをオーバーライドしなければならない．

→サブクラスに，あるメソッドを持つことを強要できる

抽象基底クラスの作り方

```
from abc import ABC, abstractmethod
class クラス名(ABC):
    @abstractmethod
    def 抽象メソッド名(self, パラメータ):
        pass
```

← 中身は空

検証：pokemon_dsc4.py

【間違った例】pokemon_dsc4.py　赤字追加箇所

```
from abc import ABC, abstractmethod

class Validator(ABC):
    @abstractmethod
    def validate(self, value):
        pass

class Monster:
    title = Validator(2, 5)
    level = Validator(1, 100)
```

【正しい例】pokemon_dsc4.py　青字修正箇所

```
from abc import ABC, abstractmethod

class Validator(ABC):
    @abstractmethod(
    def validate(self, value):
        pass

class Monster:
    title = Title(2, 5)
    level = Level(1, 100)
```

実行例

```
TypeError: Can't instantiate abstract class Validator with
abstract methods validate
```

※抽象メソッド validate を持つ抽象クラス Validator は
インスタンス化できない

検証：pokemon_dsc4.py

【間違った例】pokemon_dsc4.py　赤字追加箇所

```
from abc import ABC, abstractmethod

class Validator(ABC):
    @abstractmethod(
    def validate(self, value):
        pass

    @abstractmethod
    def hoge(self):
        pass
```

実行例

```
TypeError: Can't instantiate abstract class Title with abstract
method hoge
```

※抽象メソッド hoge を持つ抽象クラス Title は
インスタンス化できない
→具象化していない(つまり抽象メソッド) hoge を持っている

試しに，Title.__dict__もprintしてみよう