2022年10月03日(月) 4限
@研究棟A302

プログラミングA2
第2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

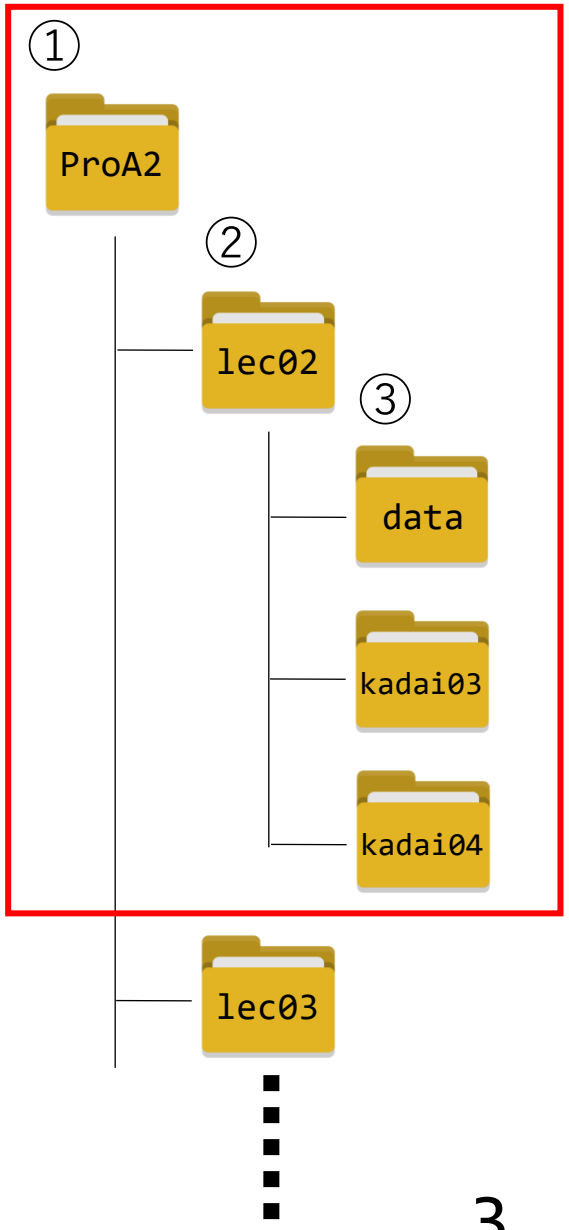
第 1回：＜復習編＞関数，ファイル入出力，コンテナデータ型
第 2回：＜復習編＞クラスとオブジェクト
第 3回：＜文法編＞関数の高度な利用法１
第 4回：＜文法編＞関数の高度な利用法２
第 5回：＜文法編＞オブジェクト指向プログラミング
第 6回：＜応用編＞データ構造とアルゴリズム１
第 7回：＜応用編＞データ構造とアルゴリズム２
第 8回：＜実践編＞HTTPクライアント
第 9回：＜実践編＞スクレイピング
第10回：＜実践編＞データベース
第11回：＜実践編＞並行処理
第12回：＜総合編＞総合演習(複合問題)
第13回：＜総合編＞まとめ
第14回：＜総合編＞Python力チェック　← 確認テ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

準備

1. デスクトップなどの任意の場所に
   プログラミングA2用のフォルダを
   作成する

2. 1.の下に
   第2回授業用のフォルダを
   作成する

3. 2.の下に
   データファイル用のフォルダ，
   課題03，課題04用のフォルダを
   作成する



本日のお品書き

- **クラスとオブジェクト**
  - クラス定義とインスタンス生成
  - インスタンス変数・メソッド
  - クラス変数・メソッド
  - 静的メソッド
  - プロパティ，マングリング
  - objectクラス，特殊メソッド・属性，オーバーライド
  - オブジェクトの同一性・等価性，コピー
  - 継承

クラスとオブジェクト

クラス定義とインスタンス生成

対象となるモノやコト（オブジェクト）の設計図（クラス）を定義し，
定義に基づき実際のオブジェクト（インスタンス）を生成する

クラスの定義

```
class クラス名:
    def __init__(self, パラメータ1, パラメータ2):
        self.属性1 = パラメータ1
        self.属性2 = パラメータ2
        self.属性3 = リテラル
```

インスタンスの生成

```
インスタンス1 = クラス名(引数1, 引数2)
インスタンス2 = クラス名(引数1, 引数2)
              :
```

パラメータ・引数の種類の詳細は
第3回で説明する

ダンダー（double underscore）で囲まれたものは特殊メソッドであり，
イニシャライザ `__init__()` はインスタンス生成後に
属性に値を設定するのに用いられる．
selfを除く位置パラメータの数と，位置引数の数は一致する必要がある

インスタンス変数・メソッド

生成されるインスタンスそれぞれに紐づけられる変数（属性）・メソッド

インスタンスメソッドの定義

```
class クラス名:
    def インスタンスメソッド名(self, パラメータ):
        インスタンス変数に対する処理
        クラス変数に対する処理　など
```

インスタンスメソッドの使用

インスタンス名 . インスタンスメソッド名(引数)

例題

pokemon.py

```
class Pikachu:
    def __init__(self, name):
        self.name = name

    def appear(self):
        print(f"{self.name}が現れた")
```

← インスタンス変数

← インスタンスメソッド

get_monsters.py

```
from pokemon import Pikachu

monsters = [Pikachu(f"野生のピカチュウその{i+1}") for i in range(2)]
for mon in monsters:
    mon.appear()
```

← pokemonモジュールからPikachuクラスをimport

クラス変数・メソッド

個々のインスタンスに**紐づけられない，クラス内で共通**の変数・メソッド

クラス変数の宣言・クラスメソッドの定義

```
class クラス名:
    クラス変数 = 初期値

    @classmethod
    def クラスメソッド名(cls, パラメータ):
        クラス変数に対する処理など
```

← デコレータが必要

デコレータの詳細は
第3or4回で説明する

クラスメソッドの使用

```
クラス名.クラスメソッド名(引数)
```

←インスタンス名.クラスメソッド名(引数)
でも使用できる

インスタンスメソッド内でのクラス変数へのアクセス・クラスメソッドの使用

```
class クラス名:
    def インスタンスメソッド名(self, パラメータ):
        クラス名.クラス変数名
        クラス名.クラスメソッド名(引数)
```

例題

pokemon.py

青字：追記変更箇所

```
class Pikachu:
    num = 0
    title = "ピカチュウ"

    def __init__(self, name):
        Pikachu.num += 1

    @classmethod
    def get_num(cls):
        print(f"{cls.num}匹の{cls.title}がいる")
```

← クラス変数の初期化

← クラス変数へのアクセス

← クラスメソッドの定義

get_monsters.py

青字：追記変更箇所

```
Pikachu.get_num()
monsters = 省略
for mon in monsters:
    mon.appear()
Pikachu.get_num()
```

実行例

```
0匹のピカチュウがいる
野生のピカチュウその1が現れた
野生のピカチュウその2が現れた
2匹のピカチュウがいる
```

静的メソッド

※「静的メソッドを使用しなければならない」という状況はない.

クラスに属するが，そのクラスに依存しないメソッドのこと.
クラスに属さないただの関数と同じ挙動であるが，なんらかの方針で関数をクラスに属させたい場合に使用する.

静的メソッドの定義

```
class クラス名:
    クラス変数 = 初期値

    @staticmethod
    def 静的メソッド名(パラメータ):
        処理
```

← デコレータが必要

静的メソッドの使用

```
クラス名.静的メソッド名(引数)
```

インスタンスメソッド内での静的メソッドの使用

```
class クラス名:
    def インスタンスメソッド名(self, パラメータ):
        クラス名.静的メソッド名(引数)
```

関数外部から属性へのアクセス

●直接アクセス

インスタンス名.属性名

pokemon.py　　青字：追記変更箇所

```
def __init__(self, name):
    省略
    self.level = 5
```

↑ デフォルト値を設定しておく

get_monsters.py

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

↑ 直接アクセス＝ノーチェックなので
不正な値を設定できる

実行例

```
省略

5
-10
```

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 属性名(self):
    return self.属性名
```

値 = インスタンス名.属性名

- セッタsetter：属性に値を設定する場合

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

インスタンス名.属性名 = 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

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
```

名前マングリング

属性名の前にダブルアンダースコア__を付けることで属性を非公開にし，ゲッタとセッタを介してのみ属性にアクセスできるようにすることがある．しかし実際には，_クラス名__属性名に**マングリング（修飾）**しているだけなので完全な非公開ではない．

まず，Pokemonクラス内の変数self.levelをself.__levelに変更してみる

get_monsters.py

```
省略
print(monsters[0].__level)

print(vars(monsters[0]))


monsters[0]._Pikachu__level = -10
print(vars(monsters[0]))
```

← AttributeErrorが出るのでコメントアウト

実行例

```
{'name': '野生のピカチュウその1',
'_Pikachu__level': 5}


{'name': '野生のピカチュウその1',
'_Pikachu__level': -10}
```

メンバの追加・削除

クラス定義時には存在しなかったメンバ（インスタンス変数・メソッド，クラス変数・メソッド）を，**代入文により後から追加**することができる

インスタンス変数・メソッドの追加

インスタンス名.変数名 = 値
インスタンス名.メソッド名 = 関数オブジェクト

インスタンス変数・メソッドの削除

インスタンス名.変数名 = None
インスタンス名.メソッド名 = None

- 便利な側面もあれば，厄介な側面もある

get_monsters.py

```
省略
monsters[0].__level = -20
print(vars(monsters[0]))
```

インスタンス変数__levelが追加された

実行例

```
省略

{'name': '野生のピカチュウその1', '_Pikachu__level': -10, '__level': -20}
```

特殊メソッド(文字列化)

自作クラスを作ると，最上位クラスであるobjectクラスを自動で継承する．
objectクラスに実装されている特殊メソッドを**オーバーライド**することで
自作クラスをカスタマイズする．

- **__str__()：**
  オブジェクトを文字列に変換しようとしたときに動作する

```
def __str__(self):
    return f"{self.属性}"
```

```
print(インスタンス)
print(str(インスタンス))
```

- **__repr__()：**
  インタプリタセッションでオブジェクトを調べたり，オブジェクトを含むコンテナをprintするときに動作する

※__str__()がない場合は，__repr__()が実行される

```
def __repr__(self):
    return f"{self.属性}"
```

```
>>> インスタンス
print([インスタンス])
```

特殊属性

- **__class__：インスタンスの元となるクラスを表す**

```
インスタンス.__class__
self.__class__
```

- **__name__：モジュールの完全修飾名，実行プログラムのエントリーポイントか否か(__main__)，クラス名などを表す**

```
if __name__ == "__main__":
```

↑ 実行プログラムのエントリーポイント（トップレベル）の場合
__name__は__main__という文字列となる

```
インスタンス.__class__.__name__
```
←インスタンスの元となるクラス名

- **__dict__：クラス，インスタンスに含まれる変数の辞書**

```
クラス.__dict__
```

```
インスタンス.__dict__
```

練習問題

以下のprint文を実行すると以下のような出力となる

get_monsters.py

```
省略
print(monsters)
print(monsters[0])
```

オブジェクトが保持されている
メモリ上のアドレス

実行例

```
省略
[<pokemon.Pikachu object at 0x000017030C90FD0>,
<pokemon.Pikachu object at 0x000017030C90F70>]
<pokemon.Pikachu object at 0x000017030C90FD0>
```

以下のような出力となるようにせよ

実行例

```
省略
[
Pikachu({'name': '野生のピカチュウその1', 'types': ['でんき'], '_Pikachu__level': -10, '__level': -20}),
Pikachu({'name': '野生のピカチュウその2', 'types': ['でんき'], '_Pikachu__level': 5})
]
Pikachu({'name': '野生のピカチュウその1', 'types': ['でんき'], '_Pikachu__level': -10, '__level': -20})
```

解答例

pokemon.py

青字：追記変更箇所

```
class Pikachu:
    num = 0
    title = "ピカチュウ"

    def __init__(self, name, types):
        self.types = ["でんき"]

    def __repr__(self):
        return f"{　　クラス名　　}({　　変数一覧　　})"
```

オブジェクトの同一性と等価性

オブジェクトは「型」，「値」，「同一性」を持つ．
同一性は，メモリ上に作成されたオブジェクトのアドレスのことであり，
オブジェクトが削除されるまで変化しない．

- **同一性（is）：オブジェクトそのものが同じものか**

```
id(オブジェクト)
```

← オブジェクト固有のIDを調べる

```
if オブジェクト1 is オブジェクト2:
```

← 同一性のチェック

- **等価性「==」：オブジェクトの値が等しいか**
- **__eq__()：**
  オブジェクト同士を「==」で比較したときに動作する

```
def __eq__(self, other):
    return self.属性 == other.属性
```

← 等価性を自分で定義し
↓ 等価性をチェック

```
if インスタンス1 == インスタンス2:
```

copyモジュール

mutableなオブジェクト（リストや辞書など）のコピー時は
copyモジュールのcopy()関数，deepcopy()関数を使用する

- **copy(obj)：オブジェクトobjの浅いコピーを返す**

  浅いコピーは，新たな複合オブジェクトを作成し，
  その後（可能な限り）元のオブジェクト中に見つかったオブジェクトに対する**参照**を挿入する.

- **deepcopy(obj)：オブジェクトobjの深いコピーを返す**

  深いコピーは，新たな複合オブジェクトを作成し，
  その後元のオブジェクト中に見つかったオブジェクトの**コピー**を挿入する.

例題

クラス変数を用いてインスタンスの等価性を評価することは少ない

①同じtitleのインスタンスは等価であると定義する

②1匹目のピカチュウをmonsters[2]としてappendする

③〃をcopy()したものをmonsters[3]としてappendする

④〃をdeepcopy()したものをmonsters[4]としてappendする

pokemon.py

青字：追記変更箇所

```
class Pikachu:
    省略
①   def __eq__(self, other):
        return self.title == other.title
```

get_monsters.py

青字：追記変更箇所

```
from copy import copy, deepcopy
monsters = [Pikachu(f"野生のピカチュウその{i+1}") for i in range(2)]
②monsters.append(monsters[0])
③monsters.append(copy(monsters[0]))
④monsters.append(deepcopy(monsters[0]))
```

つづき

get_monsters.py

```
つづき
# IDチェック
for mon in monsters: print(id(mon), id(mon.name), id(mon.types))
# 同一性チェック，等価性チェック
for mon in monsters: print(monsters[0] is mon, monsters[0] == mon)
```

実行例

```
省略
2910907994064 2910907939024 2910907969280
2910907993968 2910907939120 2910907969216
2910907994064 2910907939024 2910907969280   ← 代入
2910907991520 2910907939024 2910907969280   ← copy()
2910907991280 2910907939024 2910907606464   ← deepcopy()

True True
False True
True True       ← 代入
False True      ← copy()
False True      ← deepcopy()
```

つづき

⑤1匹目のピカチュウのタイプに"ひこう"を追加する

get_monsters.py

つづき

```
monsters[0].types.append("ひこう")
for mon in monsters: print(mon.name, mon.types)
```

実行例

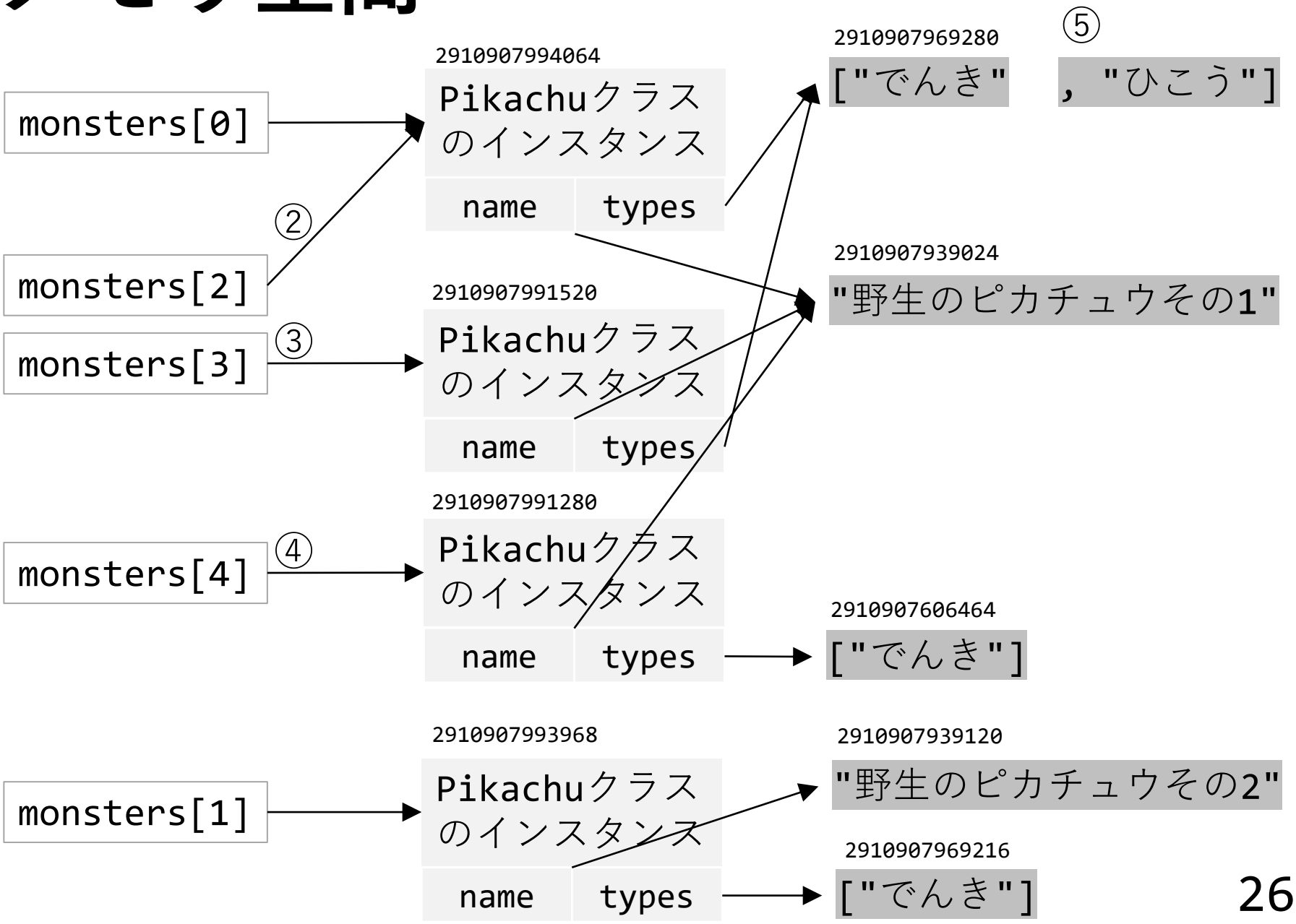
```
野生のピカチュウその1 ['でんき', 'ひこう']
野生のピカチュウその2 ['でんき']
野生のピカチュウその1 ['でんき', 'ひこう']    ← 代入
野生のピカチュウその1 ['でんき', 'ひこう']    ← copy()
野生のピカチュウその1 ['でんき']              ← deepcopy()
```

- 代入：オブジェクトの参照をコピーする→同じものを参照している
- copy()：別のオブジェクトを生成し，属性などの参照をコピーする
  →中身は同じ
- deepcopy()：属性も含め，全く別のオブジェクトが生成される
  →単純計算でメモリ使用量が2倍になる

※ immutableなオブジェクト(例：文字列)は再利用される

メモリ空間

※厳密には，文字列"でんき"も別のアドレスにあり，再利用されている
※このイメージ図では，idをアドレスのように表記している



継承

クラスを定義する際に，既存のクラスを親として指定することで，
親クラスの機能をまるごと引き継ぐことが可能となり便利である

クラスの継承

```
class 親クラス名:
    def __init__(self, パラメータ1):
        self.属性1 = パラメータ1


class 子クラス名(親クラス名):
    def __init__(self, パラメータ1, パラメータ2):
        super().__init__(パラメータ1)
        self.属性2 = パラメータ2
```

← super()関数：
親クラスオブジェクトを返す関数

【よくある役割分担】
親クラスから継承した属性1の設定は親クラスのイニシャライザに任せて
子クラスで新たに定義した属性2の設定は子クラスのイニシャライザで.

※Pythonには多重継承（複数の親クラスを拡張）があるが扱わない

例題：pikachu.py

野生のピカチュウとトレーナーに飼われているピカチュウがバトルする．レベルの大小を比較して，大きい方が勝利する．

①野生のピカチュウ(WildPikachu)クラスを定義する

pikachu.py

```
class WildPikachu():
    num = 0
    title = "野生のピカチュウ"

    def __init__(self, level):
        self.level = level
        self.name = f"{__class__.title}その{__class__.num+1}"
        __class__.num += 1

    def __str__(self):
        return f"{self.name}(Lv.:{self.level})"

    def __gt__(self, other):
        return self.level > other.level
```

野生には名前がないので，
numを使って通し番号を名前とする
例：「野生のピカチュウその2」
↓

文字列化
例：「野生のピカチュウその2(Lv.:89)」
↓

←比較演算のための特殊メソッド（greater_than）

練習問題：pikachu.py

②飼われたピカチュウ(TamePikachu)クラスを定義する

pikachu.py

```
class TamePikachu():
    def __init__(self, name, partner, level):
```

name
partner
levelをインスタンス変数に設定

```
    def
```

文字列化
例：「サトシのピカチュウ(Lv.:56)」　※網掛けはインスタンス変数

```
    def
```

インスタンスの比較演算「>」を定義する：
レベルの大小比較

実行例：battle.py

battle.py

```
from pikachu import *
from random import randint

if __name__ == "__main__":
    wild_pikas = [WildPikachu(randint(1,100)) for i in range(5)]
    pika = TamePikachu("光宙", "サトシ", 32)
    for wp in wild_pikas:
        print(pika, "vs", wp)
        if pika > wp:
            print(pika, "の勝利！")
```

実行例

```
サトシの光宙(Lv.:32) vs 野生のピカチュウその1(Lv.:37)
サトシの光宙(Lv.:32) vs 野生のピカチュウその2(Lv.:23)
サトシの光宙(Lv.:32) の勝利！
サトシの光宙(Lv.:32) vs 野生のピカチュウその3(Lv.:27)
サトシの光宙(Lv.:32) の勝利！
サトシの光宙(Lv.:32) vs 野生のピカチュウその4(Lv.:48)
サトシの光宙(Lv.:32) vs 野生のピカチュウその5(Lv.:31)
サトシの光宙(Lv.:32) の勝利！
```

練習問題：pikachu.py

③WildPikachuクラスとTamePikachuクラスの共通する部分を親クラス(Pikachuクラス)として定義する

pikachu.py

```
class Pikachu:
    title = "ピカチュウ"

    def __init__(self, level):
        levelをインスタンス変数に設定

    def インスタンスの比較演算「>」を定義する：
        レベルの大小比較
```

練習問題：pikachu.py

④Pikachuクラスを継承して，WildPikachuクラスとTamePikachuクラスを定義しなおす

pikachu.py

青字：追記変更箇所

```
class WildPikachu(Pikachu):
    クラス変数　略
    def __init__(self, level):
        親クラスのイニシャライザでlevelを設定
        self.name = f"{__class__.title}その{__class__.num+1}"
        __class__.num += 1

    def __str__(self):  そのまま
    def __gt__(self, other):  削除


class TamePikachu(Pikachu):
    def __init__(self, name, partner, level):
        self.name = name
        self.partner = partner
        親クラスのイニシャライザでlevelを設定

    def __str__(self):  そのまま
    def __gt__(self, other):  削除
```