2022年11月07日(月) 4限
@研究棟A302

プログラミングA2
第6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

- **データ構造**とアルゴリズム１
  - 集合と辞書と`hashable`
  - 組み込みクラスのサブクラスなど
    - `defaultdict`
    - `array`
    - `namedtuple`
    - `dataclass`
    - `frozenset`
    - `Counter`
  - コレクションの抽象基底クラス

集合と辞書とhashable

集合と辞書

●集合

集合(set)は，順序を持たないオブジェクトの集まりであり，要素は一意である（＝重複がない）．あるオブジェクトが集合に含まれているかを高速にチェックすることができる．

- 要素の要件：hashable

●辞書

辞書(dict)は，任意の数のオブジェクトを格納でき，それらのオブジェクトは一意なキーによって識別される．
辞書のキーを使えば，キーと関連付けられたオブジェクトの検索，削除，新たなオブジェクトの挿入を高速に行うことができる．
※集合と異なり，辞書はキーと値の対が要素になっている．

- キーの要件：hashable
- 値の要件：特になし

hashableなオブジェクト

hashableなオブジェクトとは，①生存期間中変わらないハッシュ値を持ち②他のオブジェクトと比較ができるオブジェクトである．
※①には__hash__が必要であり，②には__eq__が必要である．

- hashableの例：
  - immutableな組み込み型のオブジェクトのほとんど（str，int，floatなど）
  - immutableなコンテナ（tupleやfrozenset）のうち，その要素がimmutableなもの
  - カスタムクラスのインスタンスオブジェクト（デフォルトで）

- 等価なhashableオブジェクトは，同じハッシュ値を持つ必要がある．
  → a == b ならば，hash(a) == hash(b)

オブジェクトのハッシュ値

- **hash(オブジェクト)：**
  - オブジェクトのハッシュ値を返す組み込み関数
  - `オブジェクト.__hash__()` を呼び出す

- **__hash__(self)：**
  - 当該オブジェクト`self`のハッシュ値を返す
  - カスタムクラスの場合は独自に定義できるが，「等価な`hashable`オブジェクトは，同じハッシュ値を持つ必要がある」を満たすように定義すべき

  - `__hash__ = None` とすると`unhashable`なオブジェクトを作れる

ためしてみよう：hashable.py

●組み込み型の場合

hashable.py

```
a = 243
print(hash(a))

a = 172.5
print(hash(a))

a = "fsm"
print(hash(a))

a = (1, 2, 0, 1)
print(hash(a))

a = [1, 2, 0, 1]
print(hash(a))
```

実行例

```
243

115292150460

-60484321243

-24388440842

TypeError:
unhashable type: 'list'
```

← intはhashebla
　値自体がハッシュ値

← floatはhashable

← 文字列はhashable

← タプルはhashable

← リストはunhashable

ためしてみよう：hashable.py

●カスタムクラスの場合

hashable.py

```
class Monster:
    省略

a = Monster("ピカチュウ")
pprint(Monster.__dict__)
print(hash(a))
```

実行例

```
mappingproxy({})
101512823290
```

← __hash__は見えない

hashable.py 青字追記箇所

```
class Monster:
    def __eq__(self, other):
        return self.title
               == other.title

a = Monster("ピカチュウ")
pprint(Monster.__dict__)
print(hash(a))
```

__hash__ = None が追加され，unhashableになった ↓

実行例

```
mappingproxy({__hash__: None})
TypeError:
unhashable type: 'Monster'
```

※ __eq__()メソッドを実装すると，自動で __hash__ = None が追加される

ためしてみよう：hashable.py

hashable.py 青字追記箇所

```
class Monster:
    def __hash__(self):
        return hash(self.title)

dct = dict()
dct[a] = "1匹目"
pprint(dct)
dct[b] = "2匹目"
pprint(dct)


c = Monster("フシギダネ")
dct[c] = "3匹目"
pprint(dct)
```

← title文字列のハッシュ値を返すように定義してみた

実行例

```
{ピカチュウ: '1匹目'}
{ピカチュウ: '2匹目'}
```

↑ hashableになり，
等価でハッシュ値も等しいため，
上書きされている

```
{ピカチュウ: '2匹目',
 フシギダネ: '3匹目'}
```

※ __hash__()を独自定義するときには，
__eq__()で定義したオブジェクト間の等価性を考慮すべき

組み込みクラスのサブクラスなど

デフォルト値付き辞書：defaultdict

https://docs.python.org/ja/3/library/collections.html#collections.defaultdict

- **defaultdict(callableオブジェクト)：**
  - 存在しないキーにアクセスした際に，callableオブジェクトを呼び出し，デフォルト値を設定する辞書を作る
  - callableオブジェクト（ほとんどがクラスのコンストラクタ）を引数なしで呼び出した際に生成されるインスタンスがデフォルト値になる
    - int()：0
    - str()：空文字列
    - list()：空のリスト

callableオブジェクト：
()を付けて呼び出すことができるオブジェクト
- 関数，メソッド，ラムダ式
- クラス（のコンストラクタ）

```
from collections import defaultdict

def_dct = defaultdict(list)
print(def_dct["イーブイ"])

sim_dct = dict()
print(sim_dct["イーブイ"])
```

↓ デフォルト値：空のリスト

```
[]
```

↓ 通常のdictの場合はエラー

```
KeyError: 'イーブイ'
```

練習問題：count_char.py

デフォルト値付き辞書を用いて，ポケモンの名前として使われる文字をカウントするコードを実装せよ．

すなわち，文字が**キー**，出現回数が**値**となる．

count_char.py

collectionsモジュール内に定義されたdefaultdictをimportする

```
titles = read_names("poke_names.txt")
chr_counts = 
for title in titles:
    for t in title:
        
pprint(chr_counts)
```

defaultdictクラスのインスタンスを生成する

← 1文字ずつ

defaultdictを参照＋更新

実行例

```
defaultdict(<class 'int'>,
            {'2': 1, '♀': 1, '♂': 1, 'ァ': 2, 'ア': 15, 'ィ': 13,…
```

例題：type_list.py

デフォルト値付き辞書を用いて，ポケモンのタイプを**キー**，
そのタイプを持つポケモンの名前リストを**値**として
持つような辞書を構築するコードを実装せよ

type_list.py

```
from collections import defaultdict

types = read_types("poke_names.txt")
typ_lst = defaultdict(list)
for title, types in types.items():
    for type_ in types:
        typ_lst[type_].append(title)

pprint(typ_lst)
```

← 空のリストをデフォルト値とした辞書

実行例

```
defaultdict(<class 'list'>,
            {'あく': ['ブラッキー', 'ヤミカラス', … 'バンギラス'],
             'いわ': ['イシツブテ', 'ゴローン', 'ゴローニャ',…
```

配列：array

https://docs.python.org/ja/3/library/array.html

- **array(型, iterableオブジェクト)：**
  指定した型のみを要素とすることができるリスト

int_array.py

```
from array import array

lst = [1, 16, 129, 134, 246, 467]
ary = array('I', lst)          ← unsigned int型 'I' の配列

lst.append("国道")             ← 通常のリストは文字列も追加できる
print(lst)

ary.append("国道")             ← int型の配列に文字列は追加できない
print(ary)
```

実行例

```
[1, 16, 129, 134, 246, 467, '道路']

TypeError: an integer is required (got type str)
```

名前付きタプル：namedtuple

https://docs.python.org/ja/3/library/collections.html#collections.namedtuple

- **namedtuple(クラス名，フィールド名シーケンス)：**
  - 要素に名前（フィールド名，属性名）が付いたタプル
  - クラスと辞書とタプルの中間的な存在

named_tuple.py

```
from collections import namedtuple
                                ↓ クラス名      ↓ フィールド名のリスト
Monster = namedtuple("Monster", ["title", "level", "type_"])
fushi = Monster("フシギダネ", 25, "くさ　どく")  ←位置引数で値を設定
                                                  キーワード引数も可

print(fushi)
print(fushi.title)      ← フィールドtitleにアクセス
tit, lev, typ = fushi   ← アンパック
print(tit, lev, typ)

for item in fushi:      ← タプルと同じくiterable
    print(item)

fushi.title = "フシギソウ"  ← タプルと同じくimmutable
```

練習問題：stat_tuple.py

base_stats.txtを読み込み，1匹の種族値を1つの名前付きタプルStatで表し，それらを格納したリストを作成せよ．

つまり，namedtupleが251個並んだリストを作成する．

stat_tuple.py

collectionsモジュール内に定義されたnamedtupleをimportする

```
if __name__ == "__main__":
    stats = read_stats("base_stats.txt")
    field_names = ["H", "A", "B", "C", "D", "S"]
    tpls = []
    for stat in stats:
        tpl = 
        tpls.append(tpl)
    pprint(tpls)
```

namedtupleによるクラスを定義し，同時に値も設定している例

実行例

```
[Stat(H=45, A=49, B=49, C=65, D=65, S=45),
 Stat(H=60, A=62, B=63, C=80, D=80, S=60),
 Stat(H=80, A=82, B=83, C=100, D=100, S=80),…
```

データクラス：dataclass

https://docs.python.org/ja/3/library/dataclasses.html

カステムクラスを定義する際，__init__()，__eq__()，__repr__()などお決まりの文（ボイラープレート）をいちいち定義するのは面倒である．メソッドが少ないシンプルなクラスを作る際には，これらの特殊メソッドを自動で追加してくれるデータクラスが便利である．

データクラスの定義

```
from dataclasses import dataclass

@dataclass
class クラス名:
    属性1: 型
    属性2: 型

    def 必要なメソッド1():

    def 必要なメソッド2():
```

通常クラスの定義

```
class クラス名:
    def __init__(self, val1, val2):
        属性1 = val1
        属性2 = val2

    def __repr__(self):
        return ""

    def __eq__(self, other):
        return ""

    def 必要なメソッド1():

    def 必要なメソッド2():
```

↑
デフォルトで，
__init__()，__eq__()，__repr__()は
自動追加されるので，定義しなくてもよい

コード例：vector.py

vector.py

```
from dataclasses import dataclass
@dataclass
class Vector:
    x: int
    y: int

if __name__ == "__main__":
    v1 = Vector(2, 7)
    print(v1)
    print(v1.__dict__)
    pprint(Vector.__dict__)
    v2 = Vector(8, 10)
    print(v2)
    v3 = Vector(2, 7)
    print(v1 == v2, v1 == v3)

    v4 = v1+v2
    print(v4)
    v5 = v4-v1
    print(v5, v5 == v2)
```

__repr__が定義されている

実行例

```
Vector(x=2, y=7)
{'x': 2, 'y': 7}
```

←__init__などがクラスの属性辞書に存在している

```
Vector(x=8, y=10)

False True
```

__eq__が定義されている

```
TypeError: unsupported operand
type(s) for +: 'Vector' and 'Vector'
```

__add__や__sub__は定義されていない

※__eq__()は，デフォルトでは全ての属性の値が等しいければ等価であるとみなす

練習問題：stat_dataclass.py

base_stats.txtを読み込み，1匹の種族値を1つのデータクラスStatで表し，それらを格納したリストを作成せよ．

つまり，dataclassが251個並んだリストを作成する．

stat_dataclass.py

```
dataclassesモジュール内に定義されたdataclassをimportする
@dataclass
class Stat:
    int型の
    属性を
    6つ定義

if __name__ == "__main__":
    stats = read_stats("base_stats.txt")
    dcls = []
    for stat in stats:
        dcl = Statクラスのインスタンスを生成
        dcls.append(dcl)
    pprint(dcls)
```

実行例

```
[Stat(H=45, A=49, B=49, C=65, D=65, S=45),
 Stat(H=60, A=62, B=63, C=80, D=80, S=60),
 Stat(H=80, A=82, B=83, C=100, D=100, S=80),…
```

おまけ問題：stat_dataclass.py

課題10のShuzokuchi.pyのように，__add__()と__sub__()を定義し，Statインスタンス同士を「+」と「-」で算術演算できるようにせよ．

stat_dataclass.py

```
from dataclasses import dataclass
@dataclass
class Stat:
    def __getitem__(self, key):
        return self.__dict__[key]

    def __add__(self, other):
        return __class__(**{
                                        )

    def __sub__(self, other):
        return __class__(**{
                                        )
```

←なぜ定義している？何に必要？

辞書の内包表記
キーは属性名，値は属性の**和**

辞書の内包表記
キーは属性名，値は属性の**差**

クラス自体（＝Stat）↑
インスタンスを生成している

※__getitem__()を定義しなかった場合はどうするのだろう？

データクラスのオプション

デコレータの引数により，自動追加したい特殊メソッドを選択できる

データクラスの定義

```
from dataclasses import dataclass

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,
frozen=False, match_args=True, kw_only=False, slots=False)
class クラス名:
    省略
```

field()関数により，属性のデフォルト値やreprでの表示を設定できる

データクラスの定義

```
from dataclasses import dataclass, field

@dataclass
class クラス名:
    属性1: 型 = 10
    属性2: 型 = field(repr=False, default=10)
```

immutableな集合：frozenset

https://docs.python.org/ja/3.10/library/stdtypes.html#frozenset

- **frozenset(iterableオブジェクト)：**
  - immutableな集合（作成後に要素を変更できない）
  - 要素の検索はできる／追加・削除はできない
  - hashable → 他の集合の要素，辞書のキーになれる
  - 集合とタプルの中間的な存在

frozen_set.py

```
fushi_fset1 = frozenset(["フシギダネ", "フシギソウ", "フシギバナ"])
fushi_fset2 = frozenset(["フシギバナ", "フシギソウ", "フシギダネ"])

dct_fset = defaultdict(int)
dct_fset[fushi_fset1] += 1
dct_fset[fushi_fset2] += 1
pprint(dct_fset)
```

実行例

```
defaultdict(<class 'int'>,
{frozenset({'フシギソウ', 'フシギダネ', 'フシギバナ'}): 2})
```

多重集合：Counter

https://docs.python.org/ja/3.10/library/collections.html#collections.Counter

- **Counter(iterableまたはmapping)：**
  - hashableの出現回数をカウントするdictのサブクラス
  - インスタンスメソッド
    - update(iterableまたはmapping)：インスタンスを更新する
    - most_common(件数)：出現回数が多い順に件数分の要素を返す

counter.py

```
from collections import Counter

counter = Counter("フシギダネ")
print(counter)
counter.update("フシギソウ")
print(counter)
counter.update("フシギバナ")
print(counter)


print(counter.most_common(3))
```

実行例

```
Counter({'フ': 1, 'シ': 1, 'ギ': 1, 'ダ': 1, 'ネ': 1})
Counter({'フ': 2, 'シ': 2, 'ギ': 2, 'ダ': 1, 'ネ': 1, 'ソ': 1, 'ウ': 1})
Counter({'フ': 3, 'シ': 3, 'ギ': 3, 'ダ': 1, 'ネ': 1, 'ソ': 1, 'ウ': 1, 'バ': 1, 'ナ': 1})

[('フ', 3), ('シ', 3), ('ギ', 3)]
```

練習問題：type_counter.py

poke_names.txtからタイプを読み込み，
タイプのペアをカウントするコードを実装せよ．

タイプが1つの場合もカウントすること．

そして，出現回数が多い上位5件を表示せよ．

[要件]

- ファイルを読み込み，タイプの集合を要素としたリストを返すread_types()関数を定義し，使用する
- 上記のリストに基づき，Counterインスタンスを生成する
- Counterクラスのインスタンスメソッドを用いて上位5件を表示する

実行例

```
[(frozenset({'みず'}), 25),
 (frozenset({'ノーマル'}), 23),
 (frozenset({'ほのお'}), 16),
 (frozenset({'でんき'}), 12),
 (frozenset({'ノーマル', 'ひこう'}), 10),…
```

解答例：type_counter.py

type_counter.py

```
from pprint import pprint
```

collectionsモジュール内に定義されたCounterをimportする

```
def read_types(file_path):
    types = list()
    with open(file_path, "r", encoding="utf8") as rfo:
        for row in rfo:
            _, _, typ, *_ = row.rstrip().split("¥t")
            types.append(
    return types #タイプ文字列のfrozensetが並ぶリスト


if __name__ == "__main__":
    types = read_types("lec06/data/poke_names.txt")
    typ_cnt =
    pprint(typ_cnt.
```

タイプのfrozensetをappendする

)

Counterクラスのインスタンスを生成

上位5件を抽出

)

組み込みクラスの継承

組み込みクラスを継承することで，少し性質の異なるリストやタプルを自作できる．

even_list.py

```
class EvenList(list):
    def __setitem__(self, idx, value: int):
        if value%2 != 0:
            raise ValueError(f"奇数は設定できません：{value}")
        super().__setitem__(idx, value)

    def append(self, value):
        if value%2 != 0:
            raise ValueError(f"奇数は設定できません：{value}")
        super().append(value)

if __name__ == "__main__":
    eve_lst = EvenList()
    for i in range(10):
        try:
            eve_lst.append(i)
        except Exception as e:
            print(e)
    print(len(eve_lst), eve_lst)
```

←[idx]=valueで発動する特殊メソッド

←必ずlistクラスの__setitem__()を使用して値を設定

実行例

```
奇数は設定できません：1
奇数は設定できません：3
奇数は設定できません：5
奇数は設定できません：7
奇数は設定できません：9
5 [0, 2, 4, 6, 8]
```

← listクラスを継承しているので，
- __len__()を定義しなくてもlen()が使える
- __repr__()を定義してなくてもprint()で表示できる

コレクションの抽象基底クラス

https://docs.python.org/ja/3/library/collections.abc.html#collections-abstract-base-classes

【再掲】オブジェクトの性質

- **iterable：forで繰り返し可能なオブジェクト**
  - 例：コンテナやファイルオブジェクト
- **sized：len()で要素数を返すオブジェクト**
- **mutable：作成後に値を変更できるオブジェクト**
  - 例：リスト，集合，辞書など
  - 文字列，タプルはmutableでない（＝immutable）
  - ※再代入は，別オブジェクトへの変数名の付け替えなので可能
  - ※immutableなものはhashable
  - ※hashableなものだけが，辞書のキー，集合の要素とすることが可能
- **callable：()を付して呼び出せるオブジェクト**
  - 例：クラス，関数，ラムダ式
  - ジェネレータはcallableでない

コレクションの抽象基底クラス

抽象基底クラスは,
- 複数のクラスを抽象クラスでまとめる
- 継承先クラスのインスタンス生成時にメソッドの存在有無をチェック

に用いられる.
つまり，クラスが特定のインターフェース（実体を持たず，動作の一覧だけを列挙したもの）を提供しているか判定するのに用いられる.
※Javaなどの他言語でいうインタフェースと完全に一致する概念はない

- **Container：__contains__()を提供**
- **Iterable：__iter__()を提供**
- **Sized：__len__()を提供**
- **Hashable：__hash__()を提供**
- **Callable：__call__()を提供**

サブクラス，インスタンスの確認

- **issubclass(あるクラス，別のクラス)：**
  あるクラスが別のクラスのサブクラスならTrueを返す
  ※直接的，間接的，仮想的な継承でもよい

abc_check.py

```
print(issubclass(Zukan, Pokemon))
print(issubclass(Zukan, list))
print(issubclass(Zukan, Iterable))
```

実行例

```
False
False
True
```

ZukanクラスはIterableクラスを直接継承していないがIterableクラスが持つ抽象メソッドを実装している
←

- **isinstance(インスタンス，クラス情報)：**
  インスタンスがあるクラスのインスタンスならTrue

abc_check.py

```
print(isinstance(zukan, Zukan))
print(isinstance(zukan, list))
print(isinstance(zukan, Iterable))
print(isinstance(zukan, Iterator))
print(isinstance(zukan, Sized))
print(isinstance(zukan, Sequence))
print(isinstance(zukan, Container))
```

実行例

```
True
False
True
False
True
False
False
```

zukanインスタンスの属するクラスZukanはSizedクラスを直接継承していないがSizedクラスが持つ抽象メソッドを実装している
←

属性有無の確認

- **hasattr(オブジェクト，属性名)：** ※属性**名**は文字列です
  オブジェクトが属性名の属性を持っていたらTrueを返す

abc_check.py

```
print("__iter__", hasattr(Zukan, "__iter__"))
print("__next__", hasattr(Zukan, "__next__"))
print("__len__", hasattr(Zukan, "__len__"))
print("__getitem__", hasattr(Zukan, "__getitem__"))
print("__contains__", hasattr(Zukan, "__contains__"))
```

実行例

```
__iter__ True
__next__ False
__len__ True
__getitem__ True
__contains__ False
```

動的型付け言語と型チェック

Pythonのような動的型付け言語では，ある1つの変数にあらゆる型のオブジェクトを紐づける（代入する）ことができるため，
安全のためにそのオブジェクトの型を事前にチェックする必要がある．

- **LBYL（Look Before You Leap：転ばぬ先の杖）**

abc_check.py

```
#zukan = 5
if isinstance(zukan, (list, tuple, dict, set, range, str, Zukan)):
    for item in zukan:
        print(item)
```

↑
エラーが出る前にif文で型チェック

- **EAFP（Easier to Ask for Forgiveness than Permission：許可をもらうより，後で謝った方が楽）**

abc_check.py

```
#zukan = 5
try:
    for item in zukan:
        print(item)
except TypeError as e:
    print(e)
```

← とりあえずtryして
エラー出たらexceptへ

ダック・タイピング

直訳：「もしもそれがアヒルのように歩き、アヒルのように鳴くのなら、それはアヒルに違いない」
Python訳：「オブジェクトがアヒルのように歩き、アヒルのように鳴くメソッドを実装しているなら、**そのオブジェクトのクラスが何であれ**、アヒルと同じように機能する」

●型チェックよりも，持っている属性＝機能で判断

abc_check.py

```
#zukan = 5
if hasattr(zukan, "__iter__"):
    for item in zukan:
        print(item)
```

← __iter__だけならいいけど
たくさんの条件となる属性が必要な場合
不格好になる

●hasattrよりも，抽象基底クラス

abc_check.py

```
#zukan = 5
if isinstance(zukan, Iterable):
    for item in zukan:
        print(item)
```