

2022年10月17日(月) 4限
@研究棟A302

プログラミングA2 第3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

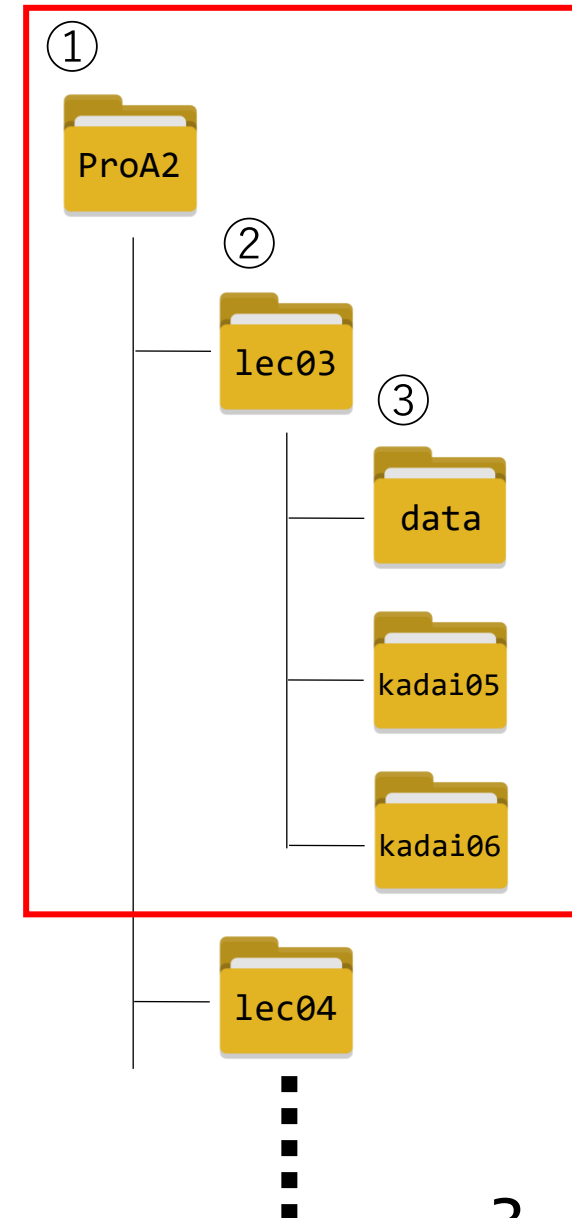
第12回：＜総合編＞総合演習(複合問題)

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3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05、課題06用のフォルダを作成する



本日のお品書き

- 前回課題の振り返りとパイソニックな書き方
- 関数の高度な利用法 1
 - 引数あれこれ
 - 引数・パラメータの種類
 - 可変長引数, アンパック
 - 関数はオブジェクト
 - 高階関数, コールバック関数
 - 関数内関数, クロージャ
 - 無名関数 (lambda)

パイソニックな書き方

●パイソニックでないfor文

```
for i in range(len(monsters)):  
    print(monsters[i])
```

要素を1つずつ
取り出すだけなのに、
i って本当に必要？

●パイソニックなfor文

```
for mon in monsters:  
    print(mon)
```

- 添え字が必要ならenumerateする

```
for i, mon in enumerate(monsters):  
    print(i, mon)
```

パisonニックな書き方

●パisonニックでない書き方

```
flag = False
for i in range(len(monsters)):
    if monsters[i] == target:
        flag = True
if flag == True:
    return "図鑑に登録済み"
else:
    return "図鑑に登録されてない"
```

●パisonニックなfor-else文

```
for mon in monsters:
    if mon == target:
        return "図鑑に登録済み"
else:
    return "図鑑に登録されてない"
```

breakでなく
returnの場合は、
elseも不要である

パイソニックな書き方

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

pika = Monster("ピカチュウ")
puka = Monster("プカチュウ")
```

●パイソニックな特殊メソッドの使い方

```
if pika == puka:
```

↑
self

↑
other

← 等価比較演算子「==」の使用により
自動で特殊メソッド__eq__()が呼び出される

●パイソニックでない特殊メソッドの使い方

```
if pika.__eq__(puka):
```

↑
self

↑
other

その他課題について

- **ファイル名＝モジュール名は指示通りに**
 - モジュール名が異なるとimportできず、採点者が実行できません
- **メソッド名、関数名は指定があれば従うように**
 - 変更できないコード（adventure.pyなど）から呼び出している場合は、そのコードに書かれた名前でなければ呼び出しできない
- **コマンドライン引数を使用する**
 - データファイルのパスなどは、採点者の環境でも実行できるようにCL引数で指定することを要件に書いている
- **不必要なモジュールはimportしない**
 - 採点者の環境にそのモジュールがなければ、ModuleNotFoundErrorが発生し、実行できない
- **5限課題はテストではなく理解を深めるための演習**
 - 相談、教え合うのはよいが、コピペはダメ

引数あれこれ

関数

関数は、呼び出し元から受け取った値（パラメータ、引数）を処理し、処理した結果を呼び出し元に戻す・返す**オブジェクト**

●関数定義

```
def 関数名(パラメータ):  
    処理1  
    処理2  
    return 結果
```

●関数呼び出し

戻り値 = 関数名(引数)

引数として指定した値がパラメータ変数に代入されて、関数の中で処理が行われるため

基本的には引数の数とパラメータの数は一致する。

●ローカル変数

- 関数内で定義された変数
- 関数外からアクセス不可

●グローバル変数

- 関数外で定義された変数
- 関数内で値を変更するには `global` 宣言が必要

●引数(ひきすう)

- 位置引数
- デフォルト引数
- キーワード引数
- 可変長引数

※引数がない関数もある

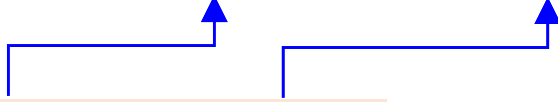
引数(argument)・実引数

実引数は、関数呼び出し時の**指定の仕方**で、位置引数とキーワード引数に分けられる。

●位置引数(positional)： 位置(順番)により紐づけるパラメータを区別する

定義部： `def 関数名(パラメータ1, パラメータ2):`

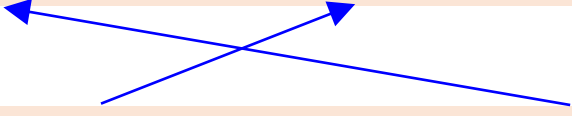
呼出部： `関数名(引数1, 引数2)`



●キーワード引数(keyword)： パラメータ名により紐づけるパラメータを区別する

定義部： `def 関数名(パラメータ1, パラメータ2):`

呼出部： `関数名(パラメータ2=値2, パラメータ1=値1)`



※位置引数とキーワード引数を混在させるときは、**位置引数を先にする** 11

パラメータ (parameter) ・ 仮引数

仮引数は、関数定義時のデフォルト値の有無で、
デフォルト引数 {あり | なし} パラメータに分けられる。

- デフォルト引数あり：関数呼び出し時に引数を省略した場合にデフォルト値がパラメータに代入される
- デフォルト引数なし：関数呼び出し時に対応する引数を必ず指定する必要がある

定義時： `def 関数名(パラメータ1, パラメータ2=デフォルト引数):`

呼出時： `関数名(引数1, 引数2)` または `関数名(引数1)` ←パラメータ2に対応する引数は省略可

※デフォルト引数 {あり | なし} パラメータを混在させるときは、
デフォルト引数なしパラメータを先にする

open関数の引数：

<https://docs.python.org/ja/3.9/library/functions.html?highlight=open#open>

可変長引数・可変長パラメータ

必須ではないオプションな引数を指定したい場合、
可変長引数により**0以上の任意の数**の引数を指定できる

●位置引数の場合

定義時： `def 関数名(パラメータ, *args):`

タプル(引数2, ...)として受け取る

呼出時： `関数名(引数1, 引数2, 引数3, 引数4, ...):`

●キーワード引数の場合

定義時： `def 関数名(パラメータ, **kwargs):`

辞書{パラメータ2:値2, ...}として受け取る

呼出時： `関数名(引数1, パラメータ2=値2, パラメータ4=値4, ...):`

※可変長引数は、一つの関数で一度しか使えない

可変長引数の仕組み

● リスト

変数1, 変数2, *変数3 = リスト

arguments.py

```
a, b, *c = [0, 1, 2, 3, 4]  
print(c)
```

リストとして受け取る

「*」がないと
どうなる？ 🤔

← [2, 3, 4]

● タプル

変数1, 変数2, *変数3 = タプル

```
a, *b, c = (0, 1, 2, 3, 4)  
print(b)
```

別の変数に
「*」を付けたら
どうなる？ 🤔

← [1, 2, 3]

● 文字列

変数1, 変数2, *変数3 = 文字列

```
*a, b, c = "ABCDE"  
print(a)
```

右辺の要素数を
少なくしたら
どうなる？ 🤔

← ['A', 'B', 'C']

※ 「*」が付いた変数は、一つの文で一つだけ

例題：pokemon.py

配布した「1ec03/data/poke_names.txt」は、
1列目：番号， 2列目：名前， 3列目：タイプ，
4列目以降：進化先がタブ区切りで書かれている。

- 1, 2, 3列目：必須／4列目以降：進化しない場合は空白
- 3列目：最大2つのタイプがあり， 2つの場合はスペース区切り
- 4列目以降：0以上の進化先がある
2つ以上の場合：タブ区切り（例：133行目のイーブイ）

このファイルを読み込み，名前のリスト**names**，タイプリストのリスト**types**，進化先辞書のリスト**evols**を作成し，3つのリストを返す関数を実装せよ。

例題：pokemon.py

pokemon.py

```
def read_names(file_path):  
    names, types, evols = [], [], []  
    with open(encoding="utf8", file=file_path, mode="r") as rfo:  
  
        for row in rfo:  
            _, nam, typ, *evo = row.rstrip().split("¥t")  
            names.append(nam) # 名前の文字列をappend  
  
            types.append(typ.split(" ")) # タイプのリストをappend  
  
            evols.append([e for e in evo]) # 進化先のリストをappend  
  
    return names, types, evols
```

キーワード引数として指定した場合は順番は関係ない

0～8個の値の代入に対応すべく「*」構文を利用

← 3つのリストのタプルをreturn

練習問題：pokemon.py

Monsterクラスのインスタンス変数`type1`と`type2`に値を設定するメソッド`set_types()`を追加せよ.

[要件]

- `type1`は**必須**で、必ず実引数で値を指定させるようパラメータを用意する
- `type2`は**オプション**で、デフォルト値を`None`としてパラメータを用意する
- 新たなインスタンス変数`type_`を作る
 - `type_ = type1+" "+type2` とする
 - ただし、`type2`が`None`なら、`type_ = type1` とする

※ちなみに、`type_`末尾のシングルアンダースコアは、`type()`関数との名前衝突回避のためである

解答例：pokemon.py

pokemon.py

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

    def set_types(self, パラメータ):
        self.type1 = t1
        self.type_ = t1
        if t2:
            self.type2 = t2
            type_を作るのつづき
```

アンパック：

右辺や実引数において、シーケンスに「*」を付けると、要素が展開される

evolution.py

```
for mon, typ in zip(monsters, types):
    mon.set_types(*typ)

for mon in monsters:
    print(mon.type_)
```

- ・ アンパックすると2つの文字列
mon.set_types("くさ", "どく")
- ・ アンパックしないと1つのリスト
mon.set_types(["くさ", "どく"])

例題

※可変長引数としてリストをアンパックしたものを渡すのはよい例ではない。
リストを引数として渡し、
リストのまま受け取ればよいからである。

0以上の進化先ポケモンの名前を受け取る可変長パラメータをもつメソッド`set_evolts()`を実装せよ

`pokemon.py`

```
class Monster:
    省略
    def set_evolts(self, *args):
        self.evolts = args
```

`evolution.py`

青字：追記変更箇所

```
for mon, typ, evo in zip(monsters, types, evols):
    mon.set_types(*typ)
    mon.set_evolts(*evo)

for mon in monsters:
    print(mon.evolts)
```

アンパック：
右辺や実引数において、
シーケンスに「*」を付けると、
要素が展開される

・アンパックすると2つの文字列
`mon.set_evolts("ラフレシア", "キレイハナ")`
・アンパックしないと1つのリスト
`mon.set_evolts(["ラフレシア", "キレイハナ"])`

関数はオブジェクト

関数はオブジェクト

```
def greeting(greet):  
    return f"{greet} world!"  
  
print( greeting("hello") )
```

●変数に代入することができる

```
aisatsu = greeting  
print( aisatsu("hello") )
```

●コンテナの要素とすることができる

```
funcs = [greeting, str.upper, str.capitalize]  
for f in funcs:  
    print( f("hello") )
```

●他の関数に引数として渡すことができる

```
def hoge(f, *args):  
    s = []  
    for arg in args:  
        s.append( f(arg.capitalize()) )  
    return " and ".join(s)  
  
print( hoge(greeting, "hello", "good morning", "goodbye") )
```

引数として渡される関数のことを **コールバック関数** と呼ぶ

他の関数を引数として受け取れる・戻り値として返せる関数のことを **高階関数** と呼ぶ

脱線：メソッドと関数

「インスタンスオブジェクトが関数の第1引数として渡されます。
例では、**x.f()**という呼び出しは、**MyClass.f(x)**と厳密に等価…」

参照：<https://docs.python.org/ja/3/tutorial/classes.html#method-objects>

インスタンスメソッドの2通りの呼び出し方

```
class MyClass:  
    def f(self):  
        pass
```

```
x = MyClass()  
x.f()
```

← インスタンスオブジェクト x を生成
← x にbindされた(bound) メソッド f() を実行
<bound method MyClass.f of <__main__.MyClass object at ...>>

```
MyClass.f(x)
```

← MyClassの中で定義された 関数 f() の
位置パラメータ self に引数 x を渡して実行
<function MyClass.f at 0x7efdca909a60>

- インスタンスオブジェクトの属性にメソッドが代入されている
- クラスオブジェクトの属性に関数が代入されている

例題：

Monsterクラスに定義された3つの関数に対して、

- ①別の名前の変数に代入し、
- ②それらを並べたリストを生成し、
- ③pikaインスタンスを引数として順に呼び出せ。

pokemon.py

```
class Monster:
```

省略

```
def evolve(self):
```

省略

```
def attack(self):
```

省略

```
def defense(self):
```

省略

evolution.py

```
pika = monsters[24]
```

① shinka, kougeki, bougyo = Monster.evolve,
Monster.attack,
Monster.defense

② funcs = [shinka, kougeki, bougyo]

③ for f in funcs:
f(pika)

実行例

ライチュウにevolve
attackできません
defenseできません

関数内関数（関数のネスト）

- 関数の中に別の関数を定義できる
- 関数を戻り値として返すことができる

greeting.py

```
def make_greeting(time):  
    def inner1(text):  
        return f"Good morning {text}!"  
  
    def inner2(text):  
        return f"Hello {text}!"  
  
    def inner3(text):  
        return f"Good evening {text}!"  
  
    if 5 < time < 10:  
        return inner1  
    if 10 < time < 17:  
        return inner2  
    else:  
        return inner3  
  
print( make_greeting(16)("world") )
```

関数make_greeting()の中で定義された関数たちは、局所変数のように関数外部では使用できない

❌ inner1("world")

しかし、内部関数を戻り値として返すことで、外部（呼び出し元）でも使用できる

f = make_greeting(16)
print(f("world")) でもOK

練習問題：pokemon.py

pokemon.pyで定義されている

`read_stats()`関数と`read_names()`関数を内側に持つ

`read_files()`関数を定義せよ。

`read_files()`関数は、2つのファイルのパスを受け取り、内側の関数にそれぞれ渡して、実行して得られる4つのリストをタプルにして返す関数である。

なお、`read_stats()`は1つの2次元リストを返し、

`read_names()`は3つのリストをタプルに返す。

今回作成する外側の関数`read_files()`は、これら4つのリストをタプルにして返す点に注意すること。

stats		names	types	evols
251匹分		フシギダネ		
		フシギソウ		
	:	:	:	:
		セレビィ		

解答例：pokemon.py

pokemon.py

```
class Monster:
```

省略

```
def read_files(2つのファイルパス):
```

```
    def read_stats(ファイルパス):
```

省略

```
        return stats
```

```
def read_names(ファイルパス):
```

省略

```
    return names, types, evols
```

```
return 外側関数の戻り値：4つのリストのタプル
```

定義であり呼び出しではない

定義であり呼び出しではない

呼び出して、戻り値を得る

(stats, (names, types, evols))
にならないように

クロージャ

クロージャとは、関数がネストされている状況において、**外側の関数の状態（変数の値）**を記憶している**関数オブジェクト**のこと

counter.py

```
def make_counter():  
    cnt = 0  
    print("カウンタ関数を作成します")  
    def _counter():  
        nonlocal cnt  
        cnt += 1  
        return f"回数：{cnt}"  
    return _counter
```

緑枠の中・青枠の外
の状態を記憶している

基本：関数の内側から外側の変数へは
「参照」のみ可能
nonlocal：関数の内側から外側の変数を
「更新」可能になる

```
cnt = 20  
counter1 = make_counter()  
print(counter1())  
print(counter1())  
print(counter1())  
  
counter2 = make_counter()  
print(counter2())  
print(counter1())
```

実行例

カウンタ関数を作成します
回数：1
回数：2
回数：3

カウンタ関数を作成します
回数：1
回数：4

練習問題

現在の進化状態を表す変数`target`の値を記憶する **クロージャ** を定義せよ.

`target`には, `Monster("フシギダネ")`, `Monster("フシギソウ")`, `Monster("ピカチュウ")`などが代入される.

evolution.py

```
def make_evolution(target):  
    def _evolve():  
        nonlocal target  
        evo_name = target.evolve()  
  
        for mon in monsters:  
            if mon.title == evo_name:  
                target = mon  
  
    return _evolve
```

← `evolve`メソッドを呼び出し
進化先ポケモンの名前文字列を取得

← 全ポケモンの中から
進化先と同じ名前のインスタンスを探し
`target`を更新

← クロージャを返す

練習問題

フシギダネmonsters[0]を引数としてクロージャを生成せよ

evolution.py

```
if __name__ == "__main__":  
    省略  
    fushi = monsters[0] # フシギダネ  
    print(f"{fushi} : ")  
  
    evolution = クロージャの生成(fushi)
```

クロージャを
3回実行

実行例

フシギダネ :

フシギソウにevolve
フシギバナにevolve
evolveできません

ラムダ式（無名関数）

ラムダ式は、**シンプル**な関数オブジェクトを簡易的に記述する方法で、再利用しないなどの理由で**名前が不要な関数**を定義するときを使う。

ラムダ式による無名関数の定義

lambda パラメータ: returnする値

```
add = lambda x, y: x+y  
add(5, 2)
```

```
(lambda x, y: x+y)(5, 2)
```

← addという名前を付けている例

← 名前を付けずに定義、呼び出している例

比較：通常関数の定義方法

```
def 関数名(パラメータ):  
    パラメータを用いた処理  
    return 処理結果
```

```
def add(x, y):  
    return x+y  
  
add(5, 2)
```

※ラムダ式も関数オブジェクトと同様、変数に代入、コンテナの要素とする、別の関数に渡すなどできる

例題

進化先の数, こうげき技の数, ぼうぎょ技の数の合計を返すラムダ式を定義し, **Monster**クラスのインスタンスをソートせよ

evolution.py

```
res = sorted(monsters,  
              key=lambda mon: (len(mon.evolvs)  
                                パラメータ +len(mon.attacks)  
                                +len(mon.defenses)),  
              reverse=True  
)
```

戻り値

比較: 通常関数定義による

```
def sum_params(mon): パラメータ  
    return (len(mon.evolvs)  
            +len(mon.attacks)  
            +len(mon.defenses))
```

戻り値

sortメソッドやsorted関数のkeyパラメータの実引数としてラムダ式を指定する場合は、ソート対象のリストの各要素（この場合Monsterクラスのインスタンス）がラムダ式のパラメータ（引数）になる

```
res = sorted(monsters, key=sum_params, reverse=True)
```

例題

2つのリスト間のユークリッド距離 $d_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{h=1}^H |x_h - y_h|^2}$ を計算する関数をラムダ式で定義せよ.

比較：通常関数定義による

```
def eucl_dist(lst_x, lst_y):  
    return sum([(x-y)**2 for x, y in zip(lst_x, lst_y)])**0.5  
  
fushi1 = [45, 49, 49, 65, 65, 45]  
fushi2 = [60, 62, 63, 80, 80, 60]  
print(eucl_dist(fushi1, fushi2))
```

distance.py

青字：ラムダ式／緑字：呼び出し

```
print(  
    (lambda lst_x, lst_y:  
        sum([(x-y)**2 for x, y in zip(lst_x, lst_y)])**0.5)  
    (fushi1, fushi2)  
)
```

実行例

```
35.566838487557476  
35.566838487557476
```