2022年12月19日(月) 4限
@研究棟A302

プログラミングA2
第12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

- Python力チェック について

- **デコレータ**
  - 復習：関数をデコレートする
  - クラスによる関数デコレータ
  - ~~クラスをデコレートする~~
  - （メソッドをデコレートする）

- **コンテキストマネージャー**

Python力チェック について

- **日時：2023年1月16日(月) 4限 (,5限)**
- **場所：ココ(研A302)**
- **形式：Google Formによる選択式**
- **持ち物：PC，学生証**
- **持ち込み：何でもOK**
  - インターネット検索，教科書，講義資料，自習メモはOK
  - 人(先生，先輩，友達，SNS)はダメ
- **内容：第1回～第13回+α**
  - 用語，出力，コード穴埋め，エラー訂正
- **配点：40点／問題数：100問／時間：60分** ← 予定なので変更する場合があります
  - 正解率が高い簡単な問題は配点が低い
  - 正解率が低い難しい問題は配点が高い

【第4回資料再掲】デコレータ

デコレータ

組み込みデコレータ：
@classmethod
@staticmethod
@property

既存の関数やメソッドを変更するのではなく，デコレートすることで機能を追加する高階関数をデコレータと呼ぶ．
引数として渡された関数やメソッドを拡張して返す．

デコレータ関数の定義

```
def デコレータ関数(デコレート対象の関数):
    def decorated():
        事前処理
        デコレート対象の関数の呼び出し
        事後処理
        return
    return decorated
```

← 簡単なデコレーション

← 簡単なデコレーション

デコレータの使用方法①

```
def デコレート対象の関数():
    …

デコレート後の関数
= デコレータ(デコレート対象の関数)
```

デコレータの使用方法②

```
@デコレータ名
def デコレート対象の関数():
    …
```

※@デコレータでデコレートすると
デコレート前の状態には戻せない．

コード例：decorator.py

デコレータ
デコレート対象
デコレート中
デコレート後

decorator.py

```
def bossy(func): # 偉そうなデコレータ
    def _func(t):
        print("オレの名前は", end="")
        func(t)
        print("様だ！")
    return _func

def humble(func): # 謙遜なデコレータ
    def _func(t):
        print("わたくしの名前は", end="")
        func(t)
        print("でございます")
    return _func

@bossy
# @humble
def print_name(t): # デコレート対象の関数
    print(t, end="")

print_name("ピカチュウ") # 常にデコレートされた状態
```

実行例

オレの名前はピカチュウ様だ！

実行例

わたくしの名前はピカチュウでございます

練習問題：html.py

HTMLの
…
タグと
…
タグで文字列を囲むデコレータを実装せよ.

[要件]

1. get_title関数
   - Monsterクラスのインスタンスを受け取り,
   - インスタンスからtitle属性を取得し,
   - そのtitle文字列を返す
2. concat_strs関数　　stringsをconcatenateするの意味
   - Monsterクラスのインスタンスがならぶリストを受け取り,
   - 各要素に対してget_title関数を用いて文字列を取得し,
   - それらを結合した文字列を返す
3. li_decorator関数
   - 受け取った関数を実行する前後に
, を付与するデコレータ
4. ul_decorator関数
   - 受け取った関数を実行する前後に
, を付与するデコレータ関数
5. 3のデコレータで1の関数をデコレートする
6. 4のデコレータで2の関数をデコレートする

解答例：html.py

デコレータ
デコレート対象
デコレート中
デコレート後

html.py：デコレート対象の関数

```
5
def get_title(mon):
    return 1
```

```
6
def concat_strs(mon_lst):
    s = []
    for mon in mon_lst:
        s.append(get_title(mon))
    return 2
```

html.py：デコレータ関数

```
def li_decorator(対象):
    def _func(t):
        s1 = "<li>"
        s2 = func(t)
        s3 = "</li>"
        return s1+s2+s3+"¥n"
    return _func


def ul_decorator(対象):
    def _func(t):
        s1 = "<ul>"
        s2 = func(t)
        s3 = "</ul>"
        return s1+s2+s3+"¥n"
    return _func
```

実行例：html.py

html.py

```
if __name__ == "__main__":
    monsters = [
        Monster("イーブイ"),
        Monster("シャワーズ"),
        Monster("サンダース"),
        Monster("ブースター"),
        Monster("エーフィ"),
        Monster("ブラッキー"),
        Monster("リーフィア"),
        Monster("グレイシア"),
        Monster("ニンフィア"),
    ]


print(concat_strs(monsters))
```

実行例

```
<ul><li>イーブイ</li>
<li>シャワーズ</li>
<li>サンダース</li>
<li>ブースター</li>
<li>エーフィ</li>
<li>ブラッキー</li>
<li>リーフィア</li>
<li>グレイシア</li>
<li>ニンフィア</li>
</ul>
```

デコレートしなかった場合も確認してみよう.

パラメータを持つデコレータ

デコレータにパラメータを持たせたいときは，
パラメータを**受け取るようなデコレータ生成関数**で
さらにラッピングする必要がある．

パラメータを持つデコレータ関数の定義

```
def デコレータを作る関数(パラメータ)
    def デコレータ(デコレート対象の関数):
        def decorated():
            事前処理
            デコレート対象の関数の呼び出し
            事後処理
            return
        return decorated
    return デコレータ
```

デコレータの使用方法②

```
@デコレータ名(引数)
def デコレート対象の関数():
    …
```

例題：html.py

タグの名前をパラメータとして受け取るデコレータを実装せよ

html.py：デコレート対象の関数

```
@tag_decorator("li")
def get_title(mon):
    return mon.title


@tag_decorator("ul")
def concat_strs(mon_lst):
    省略
```

html.py：デコレータ関数

```
def tag_decorator(tag):
    def _decorator(func):
        def _func(t):
            s1 = "<"+tag+">"
            s2 = func(t)
            s3 = "</"+tag+">"
            return s1+s2+s3+"¥n"
        return _func
    return _decorator
```

_decoratorが実際のデコレータ
tag_decoratorはデコレータを作る関数のような存在

デコレータスタック

複数のデコレータをスタックする（積み重ねる）ことで，
複数の機能をネストさせて追加することができる．

デコレータスタック

```
@デコレータ3
@デコレータ2
@デコレータ1
def デコレート対象の関数():
    …
```

← 内側（下側）のデコレータから動作することに注意

html.py

```
@tag_decorator("html")
@tag_decorator("body")
@tag_decorator("ul")
def concat_strs(mon_lst):
```

実行例

```
<html><body><ul><li>イーブイ</li>
<li>シャワーズ</li>
<li>サンダース</li>
省略
<li>リーフィア</li>
<li>グレイシア</li>
<li>ニンフィア</li>
</ul>
</body>
</html>
```

functoolsモジュール

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

前ページまでの単純なデコレータ定義では,
デコレート対象とは別の関数を定義してreturnしているため,
元の関数オブジェクトが持つ特殊属性__name__や__doc__とは異なる.
functoolsのwraps()やupdate_wrapper()を利用することで
これらの属性情報を保持できる.

decorator.py

```
from functools import wraps
def bossy(func):
    @wraps(func)
    def _func(t):
        省略

@bossy
def print_name(t):
    print(t, end="")


print(print_name.__name__, func)
```

- @wraps()**なし**の場合

実行例

```
_func <function bossy.<locals>._func at 0x…>
```

- @wraps()**あり**の場合

実行例

```
print_name <function print_name at 0x…>
```

【ここから新規】
デコレータ

練習問題：time_measure.py

関数を実行する前後の時刻を測り，差分を求めることで
関数の所要時間を表示するデコレータを実装せよ

time_measure.py

```
@time_measure
def str_concat1(num):
    s = ""
    for i in range(num):
        s += str(i)

@time_measure
def str_concat2(num):
    s = "".join([str(i) for i in range(num)])

if __name__ == "__main__":
    str_concat1(1000000)
    str_concat2(1000000)
```

← 本練習問題で実装するデコレータによりデコレートしている

空文字列sにnum個の数字を1つずつ連結：
新たなstrオブジェクトがnum回生成されるので遅い

← デコレートしないとどうなる？

実行例

```
所要時間：1.09秒
####################
所要時間：0.16秒
####################
```

解答例：time_measure.py

デコレータ
デコレート対象
デコレート中
デコレート後

time_measure.py：デコレータ部分

```
def time_measure(func):
    def _func(*args):
```

デコレート対象関数の引数

デコレート内容
- 開始時刻記録
- 対象関数の実行
- 終了時刻の記録
- 所要時間の出力
- 「#」を20個出力

デコレート後の関数を返す

デコレータは，
デコレート対象の関数funcを引数に取り
デコレータ内で
デコレートした（＝少し改良した）新たな関数を作り返す

クラスによるデコレータの実装

関数に適用するデコレータを**クラスで実装**するには，
__call__()を実装したcallableオブジェクトをラッパー関数として扱う
※ ちなみに，関数はcallableオブジェクトである

デコレータクラス

```
from functools import update_wrapper

class デコレータクラス:
    def __init__(self, 関数オブジェクト):
        self.func = 関数オブジェクト
        update_wrapper(self, self.func)

    def __call__(self, *args, **kwargs):
        事前処理
        self.func(*args, **kwargs)
        事後処理
```

← デコレート対象関数の属性
(__name__や__doc__)
を引き継ぎたい場合に必要

※このクラスのインスタンスは関数オブジェクトだが，まだデコレータされていない．
インスタンス呼び出し時に__call__が呼び出され，その度にデコレート処理される

callableオブジェクトの作成

関数でないオブジェクトにcallableという性質を持たせることにより，
関数のように扱うことができる＝関数のように呼び出すことができる．
特殊メソッド__call__を実装することでcallableなインスタンス
オブジェクトのクラスを定義できる．

呼び出し可能なインスタンスオブジェクトのクラス定義

```
class クラス名:
    def __call__(self, パラメータ):
        インスタンスが関数として
        呼び出された時の処理

インスタンス = クラス名()
インスタンス(引数)
インスタンス.__call__(引数)
```

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

第1回講義資料P.18

※デコレータはcallableである
必要がある

コード例：pokemon_call.py

pokemon_call.py

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

    def __call__(self):
        print(f"僕の名前は{self.name}です")


if __name__ == "__main__":
    fushi = Monster("フシギダネ")
    fushi()
    pika  = Monster("ピカチュウ")
    pika()
```

← fushi.__call__()と同じ

← pika.__call__()と同じ

実行例

```
 fushi()
TypeError: 'Monster'
object is not callable
```

実行例

```
僕の名前はフシギダネです
僕の名前はピカチュウです
```

例題：time_measure2.py

デコレータ
デコレート対象
デコレート中

time_measure.pyのデコレータをクラスで定義せよ

time_measure2.py：デコレータ部分

```
class time_measure:
    def __init__(self, func):
     ①② self.func = func

    def __call__(self, *args):
     ④⑥
        bgn = time.time()
     ⑤⑦ self.func(*args)
        end = time.time()
        print(f"所要時間：{end-bgn:.2f}秒")
        print("#"*20)
```

デコレート対象関数の引数

考えてみよう

コメントアウトしたprint文を実行して予想しよう

- どこで，__init__(func)が呼び出されている？？ → ①②
  ＝time_measureクラスのインスタンスはどこ？なに？
- どこで，__call__()が呼び出されている？？
- なぜ，__call__()が必要？？

time_measure2.py (time_measure.pyと同じ)

```
①  @time_measure
   def str_concat1(num):
     ⑤ s = ""
        for i in range(num):
            s += str(i)

②  @time_measure
   def str_concat2(num):
     ⑦ s = "".join([str(i) for i in range(num)])

③  if __name__ == "__main__":
     ④ str_concat1(1000000)
     ⑥ str_concat2(1000000)
```

← インスタンス = time_measure(str_concat1)

※@デコレータした場合は，対象関数がデコレートされる
つまり，デコレータクラスのインスタンスになる
デコレート前の関数は上書きされる

← インスタンスは，通常呼び出し不可能

メソッドをデコレートする

メソッドをデコレートする時は,
メソッドがどのインスタンスに対して呼び出されたのかを知る必要がある
＝ディスクリプタの__get__()を利用する

デコレートディスクリプタ

```
class デコレートディスクリプタ:
    def __init__(self, 対象メソッド):
        self.func = 対象のメソッド

    def __get__(self, obj, objtype=None):
        def _func(*args, **kwargs):
            事前処理
            self.func.__get__(obj, objtype)(*args, **kwargs)
            事後処理

        return _func
```

対象メソッドをデコレート

```
class 対象クラス:
    @デコレートディスクリプタ
    def 対象メソッド(self):
```

対象メソッドをデコレート

```
class 対象クラス:
    デコレート後メソッド =
    デコレートディスクリプタ(対象メソッド)
```

※**メソッド**がアクセスされると**ディスクリプタ**の__get__が呼び出される

コード例：time_measure3.py

time_measure3.py：デコレータ部分

```
class time_measure:
    def __init__(self, func):
    ①② self.func = func

    def __get__(self, obj, objtype=None):
     ④  def _func(*args):
            ⑥ bgn = time.time()
            ⑦ self.func.__get__(obj, objtype)(*args)
              end = time.time()
              print(f"所要時間：{end-bgn:.2f}秒")
              print("#"*20)
     ⑤  return _func
```

メソッドに紐づくインスタンス **s**

`self.func`は，対象クラスString内で定義された関数オブジェクトを表す
＝インスタンスに紐づけられていないメソッド
`self.func.__get__(obj, objtype)`により，メソッドを呼び出したインスタンスobjを`self.func`に紐づけてから，引数*argsを伴って呼び出す

コード例：time_measure3.py

time_measure3.py

```
class String:
 ① @time_measure
    def str_concat1(self, num):
      ⑦ s = ""
        for i in range(num):
            s += str(i)

 ② @time_measure
    def str_concat2(self, num):
        s = "".join([str(i) for i in range(num)])

③ if __name__ == "__main__":
    s = String()
 ④ s.str_concat1(1000000) ⑥
    s.str_concat2(1000000)
```

④インスタンスsの属性str_concat1にアクセス
→ すると，属性に設置されたディスクリプタの__get__が呼び出される
→ __get__は関数オブジェクト_funcを作成し，⑤返す

コンテキスト
マネージャ

コンテキストマネージャ

コンテキストマネージャとは，**with文の実行時にランタイムコンテキストを定義するオブジェクト**であり，**コードブロックを実行するために必要な**入り口および出口の処理を扱う．

コンテキストマネージャの作り方

```
class コンテキストマネージャ:
    __enter__(self):
        ラップしている処理開始前の処理
        return self                          ← コンテキストをreturn

    __exit__(self, exc_type, exc_value, traceback):   ← 送出した例外を捕捉する
        ラップしている処理終了後の処理


コンテキスト = コンテキストマネージャ()
with コンテキスト:
    処理
```

__enter__が呼び出される

__exit__が呼び出される

または

```
with コンテキストマネージャ() as コンテキスト:
    処理
```

__enter__が呼び出される

コード例：file_obj.py

file_obj.py

```
if __name__ == "__main__":
    rfo = open("lec12/data/poke_names.txt")
    print(rfo, rfo.closed)
    print(dir(rfo))

    with rfo:
        for i, row in enumerate(rfo):
            row = row.rstrip()
            print(row)
            if i == 9: break

    print(rfo, rfo.closed)
```

← open関数は
コンテキストマネージャ
の役割を果たす
← ファイルオブジェクトrfo
はコンテキスト

実行例

```
<_io.TextIOWrapper name='poke_names.txt' mode='r' encoding='utf-8'> False
[…, '__enter__', '__eq__', '__exit__', '__iter__', …]
<_io.TextIOWrapper name='poke_names.txt' mode='r' encoding='utf-8'> True
```

対比：file_obj0.py

- with構文+コンテキストマネージャを使わない場合

file_obj0.py

```
if __name__ == "__main__":
    rfo = open("lec12/data/poke_names.txt")
    print(rfo, rfo.closed)
    print(dir(rfo))

    try:
        for i, row in enumerate(rfo):
            row = row.rstrip()
            print(row)
            if i == 9: break
    finally:
        rfo.close()

    print(rfo, rfo.closed)
```

← コードが長くなる

← 例外が発生してもしなくても
← 必ずclose()する

例題：game_manager.py

ゲームの開始から終了までを管理するGameクラスを
コンテキストマネージャとして実装せよ

game_manager.py

```
class GameManager:
    def __enter__(self):
        print(f"=== ゲーム「{self.title}」開始 ===")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("=== ゲーム終了 ===")
        if exc_value is None:
            print("正常終了")
        else:
            print(f"{self.cnt}回目プレイ中に例外発生", exc_type, exc_value)

    def play(self):
        print(f" = {__class__.cnt}回目：プレイ開始")
        score = random.randint(-3, 5)
        if score < 0:
            raise ValueError(f"スコア異常 {score}")
```

← 書かないとどうなる？

例外クラス　例外の値
↓　↓

実行例：game_manager.py

game_manager.py

```
if __name__ == "__main__":
    with GameManager("逃げろ！こうかとん", 5) as game:
        for _ in range(game.num):
            game.play()
```

実行例

```
=== ゲーム「逃げろ！こうかとん」開始 ===
5回チャレンジします
 = 1回目：プレイ開始
 = 1回目：プレイ終了
 = 2回目：プレイ開始
 = 2回目：プレイ終了
 = 3回目：プレイ開始
 = 3回目：プレイ終了
 = 4回目：プレイ開始
 = 4回目：プレイ終了
 = 5回目：プレイ開始
 = 5回目：プレイ終了
=== ゲーム終了 ===
スコア履歴：defaultdict(<class 'int'>,
{4: 3, 0: 2})
正常終了
```

実行例

```
=== ゲーム「逃げろ！こうかとん」開始 ===
5回チャレンジします
 = 1回目：プレイ開始
 = 1回目：プレイ終了
 = 2回目：プレイ開始
=== ゲーム終了 ===
スコア履歴：defaultdict(<class 'int'>, {5: 1})
2回目プレイ中に例外発生 <class 'ValueError'>
スコア異常 -2
```

練習問題：table_creator.py

sqliteのデータベースにテーブルを構築するクラスを
コンテキストマネージャとして実装せよ

table_creator.py

```
class TableCreator:
    def __init__(self, db_path):
        self.db_path = db_path

    def __enter__(self):
        データベースに接続する
        接続したことをprintする
        カーソルオブジェクトを構築する
        など

    def __exit__(self, exc_type, exc_value, traceback):
        エラーなくテーブルが構築できたら
        正常にSQLが実行されたことをprintする
        データベースから切断する　など
        else:
            エラーが発生した場合でも
            必ずデータベースから切断する　など
```

← データベース名

実行例：table_creator.py

table_creator.py

```
    def create(self, tbl_name, col_lst):
        self.tbl_name = tbl_name
        self.columns = ",".join(col_lst)
        sql = f"CREATE TABLE {self.tbl_name} ({self.columns})"
        self.cur.execute(sql)
        print(f"{self.tbl_name}を構築しました")

if __name__ == "__main__":
    db_path = "lec12/data/pokemon.db"
    tbl_name = "names"
    col_lst = ["id", "name", "types", "evolvs"]
    with TableCreator(db_path) as tc:
        tc.create(tbl_name, col_lst)
```

← テーブルを構築するメソッド

← テーブル名
← カラムリスト

← テーブルを構築する

- 正常にテーブルを構築できた場合

実行例

```
lec12/data/pokemon.dbに接続しました
namesを構築しました
正常にSQLが実行されました
切断しました
```

- 正常にテーブルを構築できなかった場合でもDBからの切断は必ず実行される

実行例

```
lec12/data/pokemon.dbに接続しました
正常にSQLが実行されませんでした：<class
'sqlite3.OperationalError'> table names
already exists
切断しました
```