ФУНКЦІЇ ВИЩОГО ПОРЯДКУ ТА РОБОТА З КОЛЕКЦІЯМИ ДАНИХ

Питання 11.3

Вбудовані функції вищого порядку (HOF)

- Функція вищого порядку - це функція, яка може приймати в якості аргументу іншу функцію і/або повертати функцію як результат роботи.
  - Так як в Python функції – це першокласні об'єкти, то вони є HOF, ця властивість активно використовується при розробці програмного забезпечення.
- До вбудованих функцій вищого порядку відносяться map і filter.
  - Функція map приймає функцію та ітератор, повертає ітератор, елементами якого є результати застосування функції до елементів вхідного ітератора.

```
a = [1, 2, 3, 4, 5]
print(list(map(lambda x: x ** 2, a)))      #[1, 4, 9, 16, 25]
```

  - Функція filter приймає функцію предикат та ітератор, повертає ітератор, елементами якого є дані з вихідного ітератора, для яких предикат повертає True.

```
print(list(filter(lambda x: x > 0, [-1, 1, -2, 2, 0])))              # [1, 2]
print(list(filter(lambda x: len(x) == 2, ["a", "aa", "b", "bb"])))   # ['aa', 'bb']
```

Вбудовані функції вищого порядку

- Якщо дотримуватися "пітонічного" стилю програмування, то замість map і filter краще використовувати спискові включення (list comprehensions) з круглими дужками.
  - Варіант з функцією map(): `map(lambda x: x**2, [1,2,3])`
    можна замінити на: `(v**2 for v in [1,2,3])`
  - Для варіанту з функцією filter: `filter(lambda x: x > 0, [-1, 1, -2, 2, 0])`
    аналог буде виглядати так: `(v for v in [-1, 1, -2, 2, 0] if v > 0)`
- В екосистемі Python є два модуля: functools і operator, які розвивають ідею використання HOF і надають інструменти для розробки програм в функціональному стилі.

Модуль functools

- Модуль functools надає набір декораторів та HOF функцій, які або приймають інші функції в якості аргументу, або повертають функції як результат роботи.
  - Функція ***partial*** створює частково застосовані функції: якщо у функції є кілька аргументів, то можна створити на базі неї іншу, у якій частина аргументів матимуть наперед задані значення.
  - ***partialmethod*** - інструмент, аналогічний за своїм призначенням функції partial(), застосовується для методів класів.
  - ***reduce*** - згортає передану послідовність за допомогою заданої функції.
  - ***update_wrapper*** - огортає вихідну функцію так, щоб вона виглядала як функція-обгортка.
- Декоратори:
  - ***@cached_property*** - декоратор, який дозволяє створювати властивості класу з підтримкою кешування (починаючи з Python 3.8).
  - ***@lru_cache*** – декоратор для створення кешованої версії функції (методу класу).
  - ***@total_ordering*** – декоратор класу, який автоматично додає методи порівняння, якщо заданий один з методів __lt __(), __le __(), __gt __(), __ge __() і метод __eq __().
  - ***@singledispatch*** - трансформує функцію в single-dispatch функцію. Особливість такої функції полягає в тому, що вибір її реалізації визначається типом першого аргументу. Аналогічним є singledispatchmethod.
  - ***@wraps*** - декоратор для спрощення роботи з функцією update_wrapper ().

Замикання (closures)

- ***Замикання (closure)*** — функція, в тілі якої присутні посилання на змінні, оголошені поза тілом цієї функції в навколишньому коді і не є її параметрами.
  - У мові Python виділяють 4 області видимості для змінних:
    - ***Local*** - цю область видимості мають змінні, які створюються і використовуються всередині функцій.
    - ***Enclosing*** - суть даної області видимості в тому, що всередині функції можуть бути вкладені функції і локальні змінні; локальна змінна функції для її вкладеної функції знаходиться в enclosing області видимості.
    - Global - це глобальні змінні рівня модуля.
    - Built-in - рівень Python-інтерпретатора. У рамках цієї області видимості знаходяться функції open, len і т.п., також туди входять винятки. Максимально широка область видимості.
  - Локальна змінна не буде знищена, якщо на неї десь залишиться "живе" посилання після завершення роботи функції.
  - Це посилання може зберігати вкладена функція. Функції побудовані за таким принципом можуть використовуватися для побудови спеціалізованих функцій, тобто є фабриками функцій.

Замикання (closures) в Python

```
def mul(a, b):
    return a * b

def mul5(a):
    return mul(5, a)

def mult(a):
    def helper(b):
        return a * b

    return helper

print(mul(5, 7))    # 35
print(mul5(7))      # 35
print(mult(5)(7))   # 35

new_mul5 = mult(5)
print(new_mul5)
# <function mult.<locals>.helper at 0x0000021D87CB93A0>
print(new_mul5(7))  # 35
```

- Розглянемо приклад з множенням двох чисел.
  - Функція mul() перемножує два числа і повертає отриманий результат.
  - Можемо створити нову функцію, яка буде викликати mul(), з п'ятіркою і ще одним числом, яке вона буде отримувати в якості свого єдиного аргументу.
  - Вже краще, але все ще недостатньо гнучко: наступного разу, коли потрібно буде побудувати помножувач на сім, нам доведеться створювати нову функцію.
- Викликаючи new_mul5(2), ми фактично звертаємося до функції helper(), яка знаходиться всередині mul().
  - Змінна **a**, є локальною для mul(), і має область enclosing в helper().
  - Незважаючи на те, що mul() завершила свою роботу, змінна **a** не знищується, тому що на неї зберігається посилання у внутрішній функції, яка була повернута в якості результату.

Замикання (closures) в Python

```
def fun1(a):
    x = a * 3

    def fun2(b):
        nonlocal x
        return b + x

    return fun2

test_fun = fun1(4)
print(test_fun(7))    # 19
```

- Ще один приклад:
  - У функції fun1() оголошена локальна змінна **x**, значення якої визначається аргументом **a**.
  - У функції fun2() використовуються ця ж змінна **x**, nonlocal вказує на те, що значення змінної буде взято з найближчої області видимості, в якій існує змінна з таким же ім'ям.
  - Тут це enclosing-область, в якій цій змінній **x** присвоюється значення **a** * **3**.
  - Також як і в попередньому випадку, на змінну **x** після виклику fun1(4), зберігається посилання, тому вона не знищується.
- Для замикання застосовується математична властивість замкненості: множина замкнена відносно операції, якщо результати цієї операції над елементами множини є елементами цієї ж множини.
  - Ось як визначається "властивість замикання" в книзі "Структура та інтерпретація комп'ютерних програм" Айбельсона Х., Сассман Д.Д .:
  - "В загальному випадку, операція комбінування об'єктів даних має властивість замикання в тому випадку, якщо результати з'єднання об'єктів за допомогою цієї операції самі можуть з'єднуватися цієї ж операцією".

Замикання (closures) в Python

```
tpl = lambda a, b: (a, b)

a = tpl(1, 2)
print(a)          # (1, 2)

b = tpl(3, a)
print(b)          # (3, (1, 2))

c = tpl(a, b)
print(c)          # ((1, 2), (3, (1, 2)))
```

- Ця властивість дозволяє будувати ієрархічні структури даних.
  - Створимо функцію tpl (), яка на вхід приймає два аргументи і повертає кортеж.
  - Ця функція реалізує операцію "об'єднання елементів у кортеж".
  - Таким чином, у прикладі кортежі виявилися замкнені щодо операції об'єднання tpl.

Каррування (каррінг, currying)

```
def faddr(x, y, z):
    return x + y + z

def curr_faddr(x):
    def tmp_a(y):
        def tmp_b(z):
            return x+y+z
        return tmp_b
    return tmp_a

print(faddr(1, 2, 3))
print(curr_faddr(1)(2)(3))
# частково застосовані функції
p_c_faddr = curr_faddr(1)(2)
print(p_c_faddr(3))
```

Ідея повернення функції як результату знайшла застосування в побудові замикання і каррінгу.

- Каррування — перетворення функції багатьох аргументів на набір функцій, кожна з яких є функцією одного аргументу.
- Суть в тому, щоб перейти від виду f(x, y, z) до виду f(x)(y)(z).
- Каррінг дозволяє будувати частково застосовані функції.
- У прикладі p_c_faddr робить наступне: 1 + 2 + x, невідоме значення x приймає в якості аргументу.

Загалом можна скласти список властивостей, якими повинні володіти функції:

- може бути збережена в змінній або структурі даних;
- може бути передана в іншу функцію як аргумент;
- може бути повернута з функції як результат;
- може бути створена під час виконання програми;
- не повинна залежати від іменування.

- Сутність, яка задовольнить перерахованим вище вимогам, називається ***першокласним об'єктом (або об'єктом першого порядку)***.

Функції partial() та partialmethod()

```
from functools import *

def faddr(x, y, z):
  return x + 2 * y + 3 * z

# частково застосована функція
p_faddr = partial(faddr, 1, 2)
print(p_faddr(3))         # 14
# частково застосована функція
# із заздалегідь заданими значеннями
p_kw_faddr = partial(faddr, y=3, z=5)
print(p_kw_faddr(2))      # 23

class Math:
  def mul(self, a, b):
    return a * b
  x10 = partialmethod(mul, 10)

m = Math()
print(m.mul(2,3))       # 6
print(m.x10(5))         # 50
```

- Прототип: partial(func, /, *args, **keywords)
  - ***func*** – функція, для якої потрібно побудувати частково застосований варіант.
  - ***args*** – позиційні аргументи функції.
  - ***keywords*** – іменовані аргументи функції.
- partialmethod - інструмент, аналогічний до функції partial(), застосовується для методів класів.
  - Прототип: class partialmethod(func, /, *args, **keywords)
  - ***func*** – метод класу, для якої потрібно побудувати частково застосований варіант.
  - ***args*** – позиційні аргументи метода.
  - ***keywords*** – іменовані аргументи метода.

Функція reduce()

- Прототип: reduce(function, iterable[, initializer])
  - ***function*** – функція для згортки початкової послідовності, повинна приймати два аргументи.
  - ***iterable*** – послідовність для згортки (итератор).
  - ***initializer*** – початкове значення, яке буде використовуватися для звірки. Якщо значення не задано, то в якості початкового буде обраний перший елемент з ітератора.

```
from operator import add
from functools import reduce

print(add(1, 2))                        # 3
print(reduce(add, [1, 2, 3, 4, 5]))     # 15
```

Функція update_wrapper()

```
from functools import update_wrapper

def x10(a):
    return a * 10

def some_mul(a: int) -> int:
    """a * some value"""
    return a * 1

wrapped_mul = update_wrapper(x10, some_mul)
print(wrapped_mul(3))            # 30
print(wrapped_mul.__name__)      # some_mul
print(wrapped_mul.__annotations__)
# {'a': <class 'int'>, 'return': <class 'int'>}
print(wrapped_mul.__doc__)       # a * some value
```

- Огортає початкову функцію так, щоб вона виглядала як функція-обгортка.
- Прототип: update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)
  - ***wrapper*** – початкова функція.
  - ***wrapped*** – функція-обгортка.
  - ***assigned*** – кортеж з атрибутами, які необхідно замістити в функції, яка огортається, атрибутами функції-обгортки. Значення за умовчанням: WRAPPER_ASSIGNMENTS – будуть заміщені атрибути __module__, __name__, __qualname__, __annotations__, __doc__.
  - ***updated*** - кортеж з атрибутами, які необхідно замістити у функції-обгортки атрибутами функції, яка огортається. Значення за умовчанням: WRAPPER_UPDATES – буде заміщений атрибут __dict__.

Декоратор @cached_property

```
import functools


class DataProc:
    def __init__(self, data_set):
        self._data_set = data_set

    @functools.cached_property
    def mean(self):
        return sum(self._data_set) / len(self._data_set)


d = DataProc([1, 2, 3, 4, 5])
print(d.mean)                          # 3.0
```

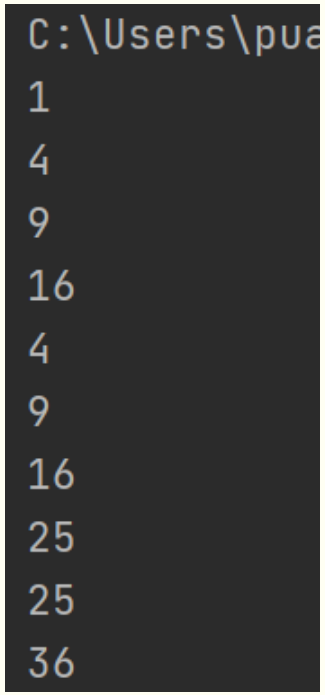
- Декоратор, який дозволяє створювати властивості класу з підтримкою кешування.
  - Це може бути корисно, якщо звернення до властивості є дорогою операцією з точки зору витрати ресурсів.
  - Доступний, починаючи з Python 3.8.
- Прототип: @cached_property(func)
  - ***func*** – декорована функція.
- Реалізуємо клас, який буде зберігати набір даних і надавати властивість mean – середнє арифметичне.
  - Для того, щоб кожного разу не обчислювати це значення при зверненні до властивості, його можна оголосити з декоратором @cached_property.

Декоратор @lru_cache

```
import functools

@functools.lru_cache(maxsize=16)
def square(value):
    return value**2

for v in [1, 2, 3, 4, 2, 3, 4, 5, 5, 6]:
    print(square(v))
```



- Прототип: @lru_cache(user_function)
  - Приклад: @lru_cache(maxsize=128, typed=False)
  - ***maxsize*** – кількість результатів роботи функції, які запам'ятовуються, для різних наборів значень аргументів. Значення за умовчанням: 128.
  - ***typed*** - якщо параметр дорівнює True, то при кешуванні також буде враховуватися тип аргументів.
- Декоратор забезпечує зберігання результатів роботи функції на різних аргументах в кількості до maxsize штук.
  - Декоратор @lru_cache створює функцію ***cache_info()***, за допомогою якої можна оцінити наскільки ефективно працює LRU кеш.
  - Вона повертає іменований кортеж з 4-ма полями: hits (кількість влучань), misses (кількість промахів), maxsize (максимальний розмір кешу) і currsize (поточний розмір кеша).
  - Для очищення кешу використовуйте функцію ***cache_clear()***.
  - Якщо параметр typed = True, то при роботі з кешем буде враховуватися тип аргументу, в цьому випадку func (10) і func (10.0) будуть розпізнаватися як виклики на різних аргументах (будуть закешовані окремо).

Декоратор @total_ordering

- Декоратор класу, який автоматично додає методи порівняння, якщо задані один з методів __lt __ (), __le __ (), __gt __ (), __ge __ (), а також метод __eq __ ().
- Прототип: @total_ordering
- Створимо клас для роботи з раціональними (дробовими) числами:

```
class Rational:
  def __init__(self, a, b):
    self.num = a
    self.den = b
  def __lt__(self, other):
    return (self.num / self.den) < (other.num / other.den)
  def __eq__(self, other):
    return self.num * other.den == self.den * other.num

a = Rational(1, 2)
b = Rational(3, 4)
print(a < b)      # True
print(a == b)     # False
print(a > b)      # False
# print(a <= b)    # TypeError: '<=' not supported between instances of 'Rational' and 'Rational'
```

Декоратор @total_ordering

```
from functools import total_ordering
@total_ordering
class Rational:
  def __init__(self, a, b):
    self.num = a
    self.den = b
  def __lt__(self, other):
    return (self.num / self.den) < (other.num / other.den)
  def __eq__(self, other):
    return (self.num == other.num) and (self.den == other.den)

a = Rational(1, 2)
b = Rational(3, 4)
print(a < b)      # True
print(a == b)     # False
print(a > b)      # False
print(a <= b)     # True
print(a >= b)     # False
```

- Якщо при оголошенні класу додати декоратор @total_ordering, то отримаємо в розпорядження весь набір операторів порівняння.

Модуль *operator*

- Містить ефективні реалізації функцій, які часто використовуються як аргументи функцій вищих порядків.
  - Наприклад, функція reduce з модуля functools, розглянутого вище, першим аргументом приймає функцію, яку буде використовувати для згортки.
  - Її можна створити десь заздалегідь або за місцем (у вигляді лямбди), або скористатись функцією з модуля operator:

```
from operator import mul
from functools import reduce

print(reduce(lambda a, b: a * b, [1, 2, 3, 4, 5]))   # 120
print(reduce(mul, [1, 2, 3, 4, 5]))                  # 120
```

Таблиця з mapping-функціями з модуля operator

Операція	Синтаксис	Функція
Додавання	*a + b*	*add(a, b)*
Конкатенація	*seq1 + seq2*	*concat(seq1, seq2)*
Тест на входження	*obj in seq*	*contains(seq, obj)*
Ділення	*a / b*	*truediv(a, b)*
Ділення	*a // b*	*floordiv(a, b)*
Побітове І	*a & b*	*and_(a, b)*
Побітове виключне АБО	*a ^ b*	*xor(a, b)*
Бітова інверсія	*~ a*	*invert(a)*
Побітове АБО	*a | b*	*or_(a, b)*
Піднесення до степені	*a ** b*	*pow(a, b)*
Перевірка того, що a є b	*a is b*	*is_(a, b)*
Перевірка того, що a не є b	*a is not b*	*is_not(a, b)*
Присвоєння значення елементу за його індексом	*obj[k] = v*	*setitem(obj, k, v)*
Видалення елемента за його індексом	*del obj[k]*	*delitem(obj, k)*
Отримання елемента за його індексом	*obj[k]*	*getitem(obj, k)*
Зсув вліво	*a << b*	*lshift(a, b)*
Остача від ділення	*a % b*	*mod(a, b)*
Множення	*a * b*	*mul(a, b)*
Множення матриць	*a @ b*	*matmul(a, b)*
Отримання від'ємної версії числа	*– a*	*neg(a)*
Логічне НЕ	*not a*	*not_(a)*
Отримання додатної версії числа	*+ a*	*pos(a)*
Зсув вправо	*a >> b*	*rshift(a, b)*
Присвоєння значень зрізу послідовності	*seq[i:j] = values*	*setitem(seq, slice(i, j), values)*
Видалення зрізу елементів	*del seq[i:j]*	*delitem(seq, slice(i, j))*
Отримання зрізу	*seq[i:j]*	*getitem(seq, slice(i, j))*
Форматування рядка	*s % obj*	*mod(s, obj)*
Віднімання	*a – b*	*sub(a, b)*
Перевірка істинності	*obj*	*truth(obj)*
Операція порядку	*a < b*	*lt(a, b)*
Операція порядку	*a <= b*	*le(a, b)*
Перевірка рівності	*a == b*	*eq(a, b)*
Перевірка нерівності	*a != b*	*ne(a, b)*
Операція порядку	*a >= b*	*ge(a, b)*
Операція порядку	*a > b*	*gt(a, b)*

ДЯКУЮ ЗА УВАГУ!

Наступна тема: Конкурентне програмування мовою Python