# ФУНКЦІЇ, ІТЕРАТОРИ ТА ГЕНЕРАТОРИ

Питання 11.2

# Потреба у чистих (pure) функціях

- Будь-який метод, який отримує доступ до стану екземпляру, щоб визначити, який результат повертати, не є чистою функцією.
- До переваг чистих функцій та коду без побічних ефектів можна віднести набагато простіше налагодження та тестування.
  - Callables, які вільно змішують роботу зі станом (statefulness) з їх результатами повернення, не можуть розглядатись незалежно від контексту, в якому вони працюють.
  - Наприклад, модульний тест може спрацювати в одному контексті, проте провалитись в іншому при однакових викликах у працюючій, stateful програмі.
  - Будь-яка програма повинна мати певний вивід, щоб бути корисною, тому повністю побічні ефекти не усунути.
  - Їх можна тільки до певної міри ізолювати, думаючи у термінах функціонального програмування.

# Іменовані функції та лямбди

- Найбільш очевидний спосіб створення callables у Python.
  - Єдина відмінність між ними – наявність атрибуту .__qualname__, оскільки обидві можуть дуже добре прив'язуватись до однієї чи кількох назв (names).
  - У більшості випадків лямбда-вирази використовуються в Python лише для callbacks та інших використань, коли проста дія вбудовується у виклик функції.
  - Загалом управляючий потік програми за бажанням можна включити в single-expression лямбда.

```
>>> def hello1(name):
.....     print("Hello", name)
.....
>>> hello2 = lambda name: print("Hello", name)
>>> hello1('David')
Hello David
```

```
>>> hello2('David')
Hello David
>>> hello1.__qualname__
'hello1'
>>> hello2.__qualname__
'<lambda>'
>>> hello3 = hello2          # can bind func to other names
>>> hello3.__qualname__
'<lambda>'
>>> hello3.__qualname__ = 'hello3'
>>> hello3.__qualname__
'hello3'
```

# Замикання (Closures) та екземпляри, які викликаються (Callable Instances)

- Одна з причин корисності функцій – можливість ізолювати стан лексично та уникнути забруднення оточуючих просторів імен.
  - Це обмежена форма незмінюваності: за умовчанням ніякі дії всередині функції не прив'яжуть змінні стану до стану ззовні функції.
  - Така гарантія дуже обмежена, оскільки як global, так і nonlocal інструкції явно дозволяють стану «протікати» в функцію.
  - Крім того, багато типів даних самі по собі змінювані (mutable), тому якщо вони передаються в функцію, то ця функція може змінити їх вміст.
  - Також виконання вводу-виводу може змінювати "стан світу", а таким чином – і результати роботи функцій.
- Якщо про класи говорять, що це "дані з прикріпленими операціями", то ***замикання*** – це "операції з прикріпленими даними".
  - До певної міри вони виконують одну задачу – збереження логіки і даних в одному об'єкті.
  - Проте класи відображають змінюваний або rebindable стан, а замикання – незмінюваність (immutability) та чисті функції.

# Іграшковий приклад

```
# A class that creates callable adder instances
class Adder(object):
    def __init__(self, n):
        self.n = n
    def __call__(self, m):
        return self.n + m
add5_i = Adder(5)       # "instance" or "imperative"
```

- Сконструйовано callable-об'єкт, який додає 5 до переданого аргументу.
- Let us also try it as a closure:

```
def make_adder(n):
    def adder(m):
        return m + n
    return adder
add5_f = make_adder(5)
```

```
>>> add5_i(10)
15
>>> add5_f(10)          # only argument affects result
15
>>> add5_i.n = 10       # state is readily changeable
>>> add5_i(10)          # result is dependent on prior flow
20
```

# Існує маленька особливість того, як Python прив'язує (binds) змінні в замикання

- Це виконується за назвою, а не значенням.
  - Наприклад, бажаємо створити кілька пов'язаних замикань, які інкапсулюють різні дані:

```
# almost surely not the behavior we intended!
>>> adders = []
>>> for n in range(5):
        adders.append(lambda m: m+n)
>>> [adder(10) for adder in adders]
[14, 14, 14, 14, 14]
>>> n = 10
>>> [adder(10) for adder in adders]
[20, 20, 20, 20, 20]
```

- Невелика зміна поведінки покращить відповідність нашій меті:

```
>>> adders = []
>>> for n in range(5):
....     adders.append(lambda m, n=n: m+n)
....
>>> [adder(10) for adder in adders]
[10, 11, 12, 13, 14]
>>> n = 10
>>> [adder(10) for adder in adders]
[10, 11, 12, 13, 14]
>>> add4 = adders[4]
>>> add4(10, 100)        # Can override the bound value
110
```

# Методи класів

```
class Car(object):
    def __init__(self):
        self._speed = 100

    @property
    def speed(self):
        print("Speed is", self._speed)
        return self._speed

    @speed.setter
    def speed(self, value):
        print("Setting to", value)
        self._speed = value
```

Усі методи класів є callables.

- Проте у більшості випадків виклик методу екземпляру йде проти природи функціонального стилю програмування.
- Зазвичай методи використовуються, оскільки ми хочемо послатись на змінювані дані, запаковані в атрибути екземпляру.
- Таким чином кожен виклик методу може повернути різний результат незалежно від аргументів, що передані в метод.

Навіть аксесори, зокрема створені з декоратором @property, технічно є callables, хоч і з обмеженим використанням з точки зору функціонального програмування: вони не приймають аргументів as getters та не повертають значень as setters:

```
# >> car = Car()
# >>> car.speed = 80
# Setting to 80
# >>> x = car.speed
# Speed is 80
```

- В аксесорі синтаксис присвоєння в Python со-opt замість передачі аргументу.
  - Для більшості синтаксису це досить просто, наприклад:

```
>>> class TalkativeInt(int):
        def __lshift__(self, other):
            print("Shift", self, "by", other)
            return int.__lshift__(self, other)
....
>>> t = TalkativeInt(8)
>>> t << 3
Shift 8 by 3
64
```

  - Кожен оператор Python «під капотом» є викликом методу.

# Статичні методи of Instances

- Використання класів та їх методів у якості просторів імен, які містять пов'язані функції близько пов'язане з функціональним стилем програмування

```
import math
class RightTriangle(object):
    "Class used solely as namespace for related functions"
    @staticmethod
    def hypotenuse(a, b):
        return math.sqrt(a**2 + b**2)

    @staticmethod
    def sin(a, b):
        return a / RightTriangle.hypotenuse(a, b)

    @staticmethod
    def cos(a, b):
        return b / RightTriangle.hypotenuse(a, b)
```

- Утримування функціональності всередині класу:
  - уникає «забруднення» глобального простору імен (модуля тощо)
  - дозволяє назвати клас або його екземпляр, коли виконуються виклики чистих функцій.
  - For example:

```
>>> RightTriangle.hypotenuse(3,4)
5.0
>>> rt = RightTriangle()
>>> rt.sin(3,4)
0.6
>>> rt.cos(3,4)
0.8
```

# Найбільш прямий підхід до створення статичних методів – декорування

- Проте в Python 3.x також можна «викидати» (pull out) функції, які не були декоровані
  - Концепція «неприв'язаного методу» в сучасних версія непотрібна:

```
>>> import functools, operator
>>> class Math(object):
...     def product(*nums):
...         return functools.reduce(operator.mul, nums)
...     def power_chain(*nums):
...         return functools.reduce(operator.pow, nums)
...
>>> Math.product(3,4,5)
60
>>> Math.power_chain(3,4,5)
3486784401
```

Проте без @staticmethod це не працюватиме з екземплярами, оскільки вони передбачають передачу self

```
>>> m = Math()
>>> m.product(3,4,5)
---------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-5-e1de62cf88af> in <module>()
----> 1 m.product(3,4,5)

<ipython-input-2-535194f57a64> in product(*nums)
      2 class Math(object):
      3     def product(*nums):
----> 4         return functools.reduce(operator.mul, nums)
      5     def power_chain(*nums):
      6         return functools.reduce(operator.pow, nums)

TypeError: unsupported operand type(s) for *: 'Math' and 'int'
```

# Генераторні функції

```
>>> def get_primes():
...     "Simple lazy Sieve of Eratosthenes"
...     candidate = 2
...     found = []
...     while True:
...         if all(candidate % prime != 0 for prime in found):
...             yield candidate
...             found.append(candidate)
...         candidate += 1
...
>>> primes = get_primes()
>>> next(primes), next(primes), next(primes)
(2, 3, 5)
>>> for _, prime in zip(range(10), primes):
...     print(prime, end=" ")
....
7 11 13 17 19 23 29 31 37 41
```

- Спеціальний вид функцій у Python, який містить оператор yield, який перетворює функцію в генератор.
  - При виклику такої функції повертається не звичайне значення, а ітератор, який продукує послідовність значень після викликів функції next() для нього або проходу в циклі.
- У принципі генератор може бути “чистим”.
  - Це чиста функція, яка утворює (потенційно нескінченну) послідовність значень, яка все ще базується тільки на переданих в неї аргументах.
  - Проте генераторні функції зазвичай у значній мірі опираються на внутрішній стан.
- Кожного разу, коли створюється новий об’єкт за допомогою get_primes(), ітератор є тією ж нескінченною «лінивою послідовністю».
  - Проте сам об’єкт є stateful, оскільки він is consumed інкрементно.

# Multiple Dispatch, або мультиметоди

- Ідея: оголосити кілька сигнатур для однієї функції та викликати відповідні обчислення залежно від типів/властивостей аргументів, що передаються.
  - Це дозволяє знизити використання умовних операторів.
  - Для демонстрації реалізуємо гру «камінь-ножиці-папір» трьома стилями.
  - Створимо класи для гри для всіх версій.

```
class Thing(object): pass
class Rock(Thing): pass
class Paper(Thing): pass
class Scissors(Thing): pass
```

- Спочатку повністю імперативна версія:

```
# >>> beats(paper, rock)
# <__main__.Paper at 0x103b96b00>
# >>> beats(paper, 3)
# TypeError: Unknown second thing
```

```
def beats(x, y):
    if isinstance(x, Rock):
        if isinstance(y, Rock):
            return None         # No winner
        elif isinstance(y, Paper):
            return y
        elif isinstance(y, Scissors):
            return x
        else:
            raise TypeError("Unknown second thing")
    elif isinstance(x, Paper):
        if isinstance(y, Rock):
            return x
        elif isinstance(y, Paper):
            return None         # No winner
        elif isinstance(y, Scissors):
            return y
        else:
            raise TypeError("Unknown second thing")
    elif isinstance(x, Scissors):
        if isinstance(y, Rock):
            return y
        elif isinstance(y, Paper):
            return x
        elif isinstance(y, Scissors):
            return None         # No winner
        else:
            raise TypeError("Unknown second thing")
    else:
        raise TypeError("Unknown first thing")

rock, paper, scissors = Rock(), Paper(), Scissors()
```

# Делегування об’єкта

- Друга спроба: усунемо деякі дублі коду за допомогою «качиної типізації»

```
class DuckRock(Rock):
    def beats(self, other):
        if isinstance(other, Rock):
            return None          # No winner
        elif isinstance(other, Paper):
            return other
        elif isinstance(other, Scissors):
            return self
        else:
            raise TypeError("Unknown second thing")
```

```
class DuckPaper(Paper):
    def beats(self, other):
        if isinstance(other, Rock):
            return self
        elif isinstance(other, Paper):
            return None          # No winner
        elif isinstance(other, Scissors):
            return other
        else:
            raise TypeError("Unknown second thing")

class DuckScissors(Scissors):
    def beats(self, other):
        if isinstance(other, Rock):
            return other
        elif isinstance(other, Paper):
            return self
        elif isinstance(other, Scissors):
            return None          # No winner
        else:
            raise TypeError("Unknown second thing")
```

# Запуск коду

```
def beats2(x, y):
    if hasattr(x, 'beats'):
        return x.beats(y)
    else:
        raise TypeError("Unknown first thing")

rock, paper, scissors = DuckRock(), DuckPaper(), DuckScissors()
# >>> beats2(rock, paper)
# <__main__.DuckPaper at 0x103b894a8>
# >>> beats2(3, rock)
# TypeError: Unknown first thing
```

- Код насправді не скоротився, проте став простішим для розуміння.
  - Більшість логіки зосереджено в окремих класах.
  - В ООП можна “делегувати dispatch об’єктові” (проте лише одному контролюючому об’єкту).

```
from multipledispatch import dispatch

@dispatch(Rock, Rock)
def beats3(x, y): return None

@dispatch(Rock, Paper)
def beats3(x, y): return y

@dispatch(Rock, Scissors)
def beats3(x, y): return x

@dispatch(Paper, Rock)
def beats3(x, y): return x

@dispatch(Paper, Paper)
def beats3(x, y): return None

@dispatch(Paper, Scissors)
def beats3(x, y): return x

@dispatch(Scissors, Rock)
def beats3(x, y): return y

@dispatch(Scissors, Paper)
def beats3(x, y): return x

@dispatch(Scissors, Scissors)
def beats3(x, y): return None
```

# Pattern Matching

- Остання спроба: виразимо логіку більш прямо за допомогою multiple dispatch.

```
@dispatch(object, object)
def beats3(x, y):
    if not isinstance(x, (Rock, Paper, Scissors)):
        raise TypeError("Unknown first thing")
    else:
        raise TypeError("Unknown second thing")

# >>> beats3(rock, paper)
# <__main__.DuckPaper at 0x103b894a8>
# >>> beats3(rock, 3)
# TypeError: Unknown second thing
```

# Протокол ітератора

- Має слабкий зв'язок з функціональним програмуванням, оскільки Python не підтримує «ліниві» структури даних на такому рівні, як наприклад, Haskell.
  - Проте використання протоколу ітератора створює подібні ефекти до роботи з lazy data structure.
- У мовах на зразок Haskell можна оголосити список простих чисел  схожим чином:

```haskell
-- Define a list of ALL the prime numbers
primes = sieve [2 ..]
  where sieve (p:xs) = p : sieve [x | x <- xs, (x `rem` p)/=0]
```

  - Тут є включення, подібне до включень з Python.
  - Також задіяна глибока рекурсія, яка в Python не спрацює коректно.
  - Основна відмінність: Haskell-версія утворює справжню (нескінченну) послідовність, а не об'єкт, здатний до послідовного створення елементів.

# Маючи генераторну функцію get_primes(), запишемо власний контейнер для такого ж моделювання

- Наприклад,

```
from collections.abc import Sequence
class ExpandingSequence(Sequence):
    def __init__(self, it):
        self.it = it
        self._cache = []
    def __getitem__(self, index):
        while len(self._cache) <= index:
            self._cache.append(next(self.it))
        return self._cache[index]
    def __len__(self):
        return len(self._cache)
```

- Звісно, що можна додати інші можливості, оскільки ліниві (lazy) структури даних are not inherently intertwined into Python.
  - Maybe we'd like to be able to slice this special sequence.
  - Maybe we'd like a prettier representation of the object when printed.
  - Maybe we should report the length as inf if we somehow signaled it was meant to be infinite.

Новий контейнер може бути як «лінивим», так і індексованим

```
>>> primes = ExpandingSequence(get_primes())
>>> for _, p in zip(range(10), primes):
....     print(p, end=" ")
....
2 3 5 7 11 13 17 19 23 29
>>> primes[10]
31
>>> primes[5]
13
>>> len(primes)
11
>>> primes[100]
547
>>> len(primes)
101
```

# Протокол ітератора

- Найпростіший спосіб створити ітератор (ліниву послідовність) у Python – визначити генераторну функцію.
  - Застосовується інструкція yield у тілі функції, щоб визначити місця (зазвичай у циклі), де продукуються значення.
  - Або технічно найпростіший спосіб – використовувати один з багатьох ітерованих (iterable) об'єктів, які вже вбудовані або постачаються стандартною бібліотекою.
  - Генераторні функції є синтаксичним цукром для визначення функції, яка повертає ітератор.
- Багато об'єктів мають метод .__iter__(), який при виклику повертає ітератор, загалом за допомогою вбудованої функції iter() або (навіть частіше) просто циклічним проходом по об'єкту (для елементів колекції).
  - Коли ітератор є об'єктом, який сам має метод .__iter__() та повертається після виклику iter(...), тоді просто повертається сам об'єкт та ынший метод – .__next__().
  - Причина такої потреби – в ідемпотентності iter():

```
iter_seq = iter(sequence)
iter(iter_seq) == iter_seq
```

# Конкретні приклади

```
>>> lazy = open('06-laziness.md')  # iterate over lines of file
>>> '__iter__' in dir(lazy) and '__next__' in dir(lazy)
True
>>> plus1 = map(lambda x: x+1, range(10))
>>> plus1                    # iterate over deferred computations
<map at 0x103b002b0>
>>> '__iter__' in dir(plus1) and '__next__' in dir(plus1)
True
>>> def to10():
...     for i in range(10):
...         yield i
...
>>> '__iter__' in dir(to10)
False
>>> '__iter__' in dir(to10()) and '__next__' in dir(to10())
True
```

```
>>> l = [1,2,3]
>>> '__iter__' in dir(l)
True
>>> '__next__' in dir(l)
False
>>> li = iter(l)          # iterate over concrete collection
>>> li
<list_iterator at 0x103b11278>
>>> li == iter(li)
True
```

- У функціональному стилі програмування запис власних ітераторів як генераторних функцій є найбільш природнім.
  - Проте можна створювати власні класи, які відповідають протоколу; часто вони також матимуть додаткову поведінку (методи), проте вона обов'язково покладатиметься на statefulness та побічні ефекти, щоб бути змістовною.

```python
from collections.abc import Iterable
class Fibonacci(Iterable):
    def __init__(self):
        self.a, self.b = 0, 1
        self.total = 0
    def __iter__(self):
        return self
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        self.total += self.a
        return self.a
    def running_sum(self):
        return self.total
```

```python
# >>> fib = Fibonacci()
# >>> fib.running_sum()
# 0
# >>> for _, i in zip(range(10), fib):
# ...     print(i, end=" ")
# ...
# 1 1 2 3 5 8 13 21 34 55
# >>> fib.running_sum()
# 143
# >>> next(fib)
# 89
```

# Модуль itertools

- Колекція потужних та ретельно спроектованих функцій для застосування алгебри ітераторів, що дозволяє потужно комбінувати ітератори.
  - Також документація модуля містить багато коротких прикладів додаткових функцій, кожна з яких використовує комбінацію з 2-3 базових функцій.
  - Сторонній модуль more_itertools постачає додаткові функції, які спроектовані для уникання поширених підводних каменів у роботі з itertools.
- Основна мета використання будівельних блоків усередині itertools – уникнути виконання обчислень до моменту їх потреби, зберігання величезних колекцій даних у пам’яті, потенційно повільного вводу-виводу тощо до строгої потреби.
  - Ітератори – ліниві послідовності (lazy sequences), які при роботі з функціями чи можливостями itertools зберігають таку властивість.

# Швидкий приклад комбінації речей

```
>>> def fibonacci():
...     a, b = 1, 1
...     while True:
...         yield a
...         a, b = b, a+b
...
>>> from itertools import tee, accumulate
>>> s, t = tee(fibonacci())
>>> pairs = zip(t, accumulate(s))
>>> for _, (fib, total) in zip(range(7), pairs):
...     print(fib, total)
...
1 1
1 2
2 4
3 7
5 12
8 20
13 33
```

- Замість stateful класу Fibonaccі можна створити один лінивий ітератор, щоб згенерувати поточне число та суму.
- Для практичних цілей zip(), map(), filter() та range() могли б знаходитись в itertools, якби не були вбудованими.
  - Ці функції ліниво генерують послідовні елементи (часто на основі існуючих iterables) без створення конкретної послідовності.
  - Вбудовані функції на зразок all(), any(), sum(), min(), max() та functools.reduce() також діють на iterables, проте загалом потребують to exhaust ітератор, ніж залишатись лінивими.
  - Функція itertools.product() може бути не на своєму місці, оскільки також створює конкретні кешовані послідовності та не може оперувати нескінченними ітераторами.

# Ланцюговий виклик (Chaining Iterables)

- Функції itertools.chain() та itertools.chain.from_iterable() комбінують кілька iterables.
  - Вбудовані методи zip() та itertools.zip_longest() також це роблять, проте дозволяють інкрементне розширення through the iterables.
  - Наслідок: хоч побудова нескінченного ланцюга iterables синтаксично і семантично можлива, реальні програми will exhaust the earlier iterable.
  - Наприклад:

```
from itertools import chain, count
thrice_to_inf = chain(count(), count(), count())
```

  - thrice_to_inf має тричі рахувати до нескінченності, проте на практиці одного разу завжди досить.
  - Просте ланцюгування великих скінченних ітерованих об'єктів може бути корисним та економним:

```
def from_logs(fnames):
    yield from (open(file) for file in fnames)
lines = chain.from_iterable(from_logs(
                ['huge.log', 'gigantic.log']))
```

  - У прикладі навіть не потрібно передавати конкретний список файлів—послідовність filenames сама може бути лінивим iterable-об'єктом.

# ДЯКУЮ ЗА УВАГУ!

Наступне питання:  Робота з колекціями даних. Функції вищого порядку.