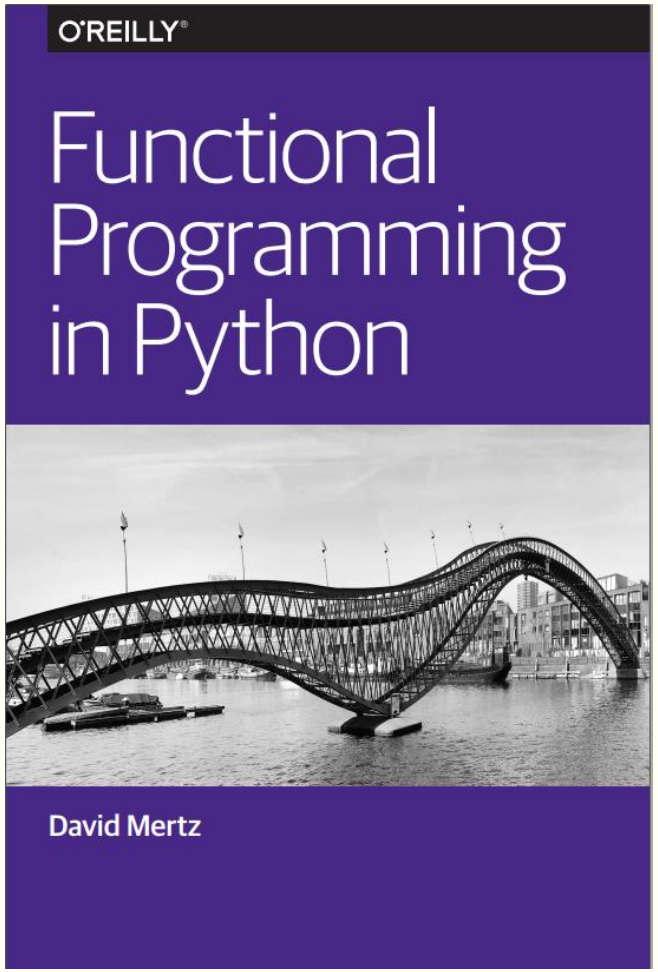
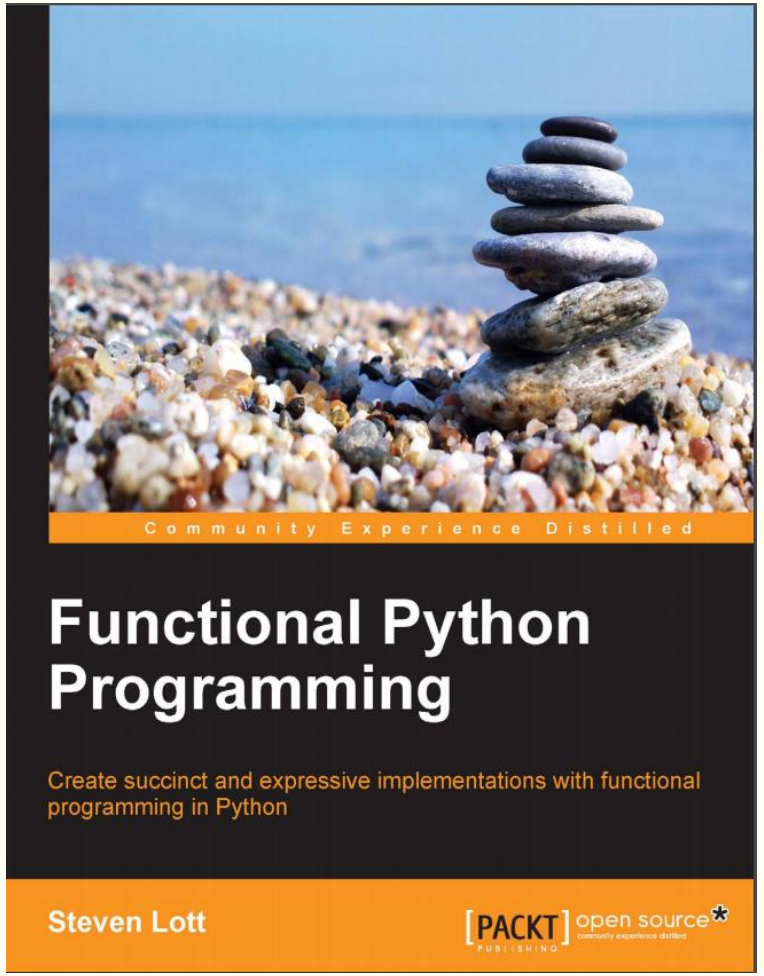
ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ В PYTHON

Тема 11

План лекції

- Виконання програми в функціональному стилі.
- Функції, ітератори та генератори.
- Робота з колекціями даних.
- Функції вищого порядку.

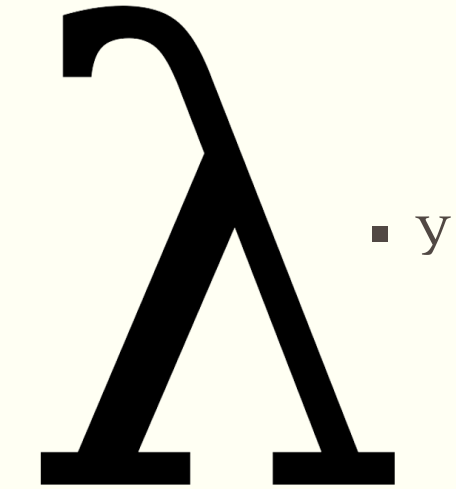

O'REILLY
Functional Programming in Python
David Mertz


Community Experience Distilled
Functional Python Programming
Create succinct and expressive implementations with functional programming in Python
Steven Lott
[PACKT] open source

ВИКОНАННЯ ПРОГРАМИ В ФУНКЦІОНАЛЬНОМУ СТИЛІ

Питання 11.1

Функціональне програмування описує обчислення за допомогою виразів та знаходження їх значень (evaluation, часто інкапсульовано в означенні функцій)



Functional Programming

- Парадигму вирізняє концепція ***стану (state)***.
  - В імперативній мові стан обчислень відображається значеннями змінних (у різних просторах імен).
    - Кожна інструкція виконує чітко визначену зміну стану через додавання чи зміну (або навіть видалення) змінної.
    - Мова імперативна, оскільки кожна інструкція (statement) є командою, яка змінює стан певним чином.
- У функціональній мові заміняємо поняття стану на знаходження значення функцій.
  - Кожне знаходження значення функції створює новий об'єкт чи об'єкти з уже існуючих.
  - Це композиція функцій, тому можна проектувати:
    - Низькорівневі функції – прості для розуміння
    - Високорівневі функції (композиції) – простіші для візуалізації, ніж складна послідовність інструкцій.
  - Спрощується модульне тестування.
  - Функціональні програми часто ефективніші та виразніші в порівнянні з імперативними.

Код у різних парадигмах

Імперативний варіант (без ООП)	Об'єктно-орієнтований варіант

```
s = 0
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        s += n
print(s)
```

```
class SummableList(list):
    def sum( self ):
        s= 0
        for v in self.__iter__():
            s += v
        return s
```

Функціональний варіант: сума багатьох трійок та п'ятірок представлена як:

- сума послідовності чисел
- послідовність значень, що відповідає простій тестовій умові. Наприклад, бути кратним 3 або 5.

```
def sum(seq):
    if len(seq) == 0: return 0
    return seq[0] + sum(seq[1:])
```

```
def until(n, filter_func, v):
    if v == n: return []
    if filter_func(v): return [v] + until( n, filter_func, v+1 )
    else: return until(n, filter_func, v+1)
```

Функція until()

- Функція filter_func() також додає два випадки:
  - Якщо значення v передається через filter_func(), створимо дуже малий список з одного елементу та допишемо (append) решту значень функції until() до цього списку.
- За допомогою функції until можна генерувати багато трійок або п'ятірок.
  - Спочатку визначимо об'єкт lambda, який відбиратиме значення: `mult_3_5= lambda x: x%3==0 or x%5==0`
  - Функція until():

```
1 def until(n, filter_func, v):
2     if v == n: return []
3     if filter_func(v): return [v] + until( n, filter_func, v+1 )
4     else: return until(n, filter_func, v+1)
```

  - Виклик та результати виводу:

```
In [9]: until(10, lambda x: x%3==0 or x%5==0, 0)
Out[9]: [0, 3, 5, 6, 9]
```

```
In [3]: mult_3_5(3)
Out[3]: True

In [4]: mult_3_5(4)
Out[4]: False

In [5]: mult_3_5(5)
Out[5]: True
```

Перехід до лямбда-функцій

- Анонімна функція, виражена одним виразом.
  - Може використовуватись замість маленької функції.
- Визначимо функцію edit_story() з аргументами:
  - words — список слів;
  - func — функція, яка повинна застосовуватись до кожного слова в списку words:

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

  - У якості слів візьмемо список гіпотетичних звуків, які може видавати кіт, який не помітив сходи:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

  - Функція запише з великої літери кожне слово та додасть до нього знак оклику:

```
>>> def enliven(word):   # больше эмоций!
...     return word.capitalize() + '!'
```

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Лямбда-функції

- Функція enliven() була короткою, тому можемо замінити її лямбдою:

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
```

  - Лямбда приймає один аргумент, у цьому прикладі – word.
  - Визначення даної функції знаходиться між двокрапкою та закриваючою дужкою.

- Часто використання справжніх функції на зразок enliven() набагато прозоріше, ніж використання лямбд.
  - Лямбди корисні в деяких випадках, коли потрібно визначити багато невеликих функцій та запам'ятати всі їх імена.
  - Зокрема, можна використовувати лямбди в графічних інтерфейсах користувача з метою визначення функції зовнішнього виклику.

Прийоми функціонального програмування

- Більшість особливостей функціонального програмування вже є частиною Python.
  - Потрібно змістити увагу від імперативних (процедурних чи об'єктно-орієнтованих) підходів.
- Далі розглянемо наступні прийоми ФП:
  - Першокласні (First-class) функції та функції вищого порядку (higher-order functions, pure functions).
  - Незмінні (Immutable) дані.
  - Строге (Strict, eager) та нестроге (non-strict, lazy) знаходження значень (evaluation).
  - Рекурсія замість явного стану в циклі.
- Функції повинні бути ***першокласними об'єктами*** у середовищі виконання.
  - У мовах на зразок C це не так.
  - У Python функції є об'єктами, якими можуть маніпулювати інші Python-функції.
  - Можна також створити функцію як об'єкт, що викликається (callable object) або присвоївши їй lambda-функції.

```
1 def example(a, b, **kw):
2     return a*b
3
4 print(type(example))
5 print(example.__code__.co_varnames)
6 print(example.__code__.co_argcount)
```

```
<class 'function'>
('a', 'b', 'kw')
2
```

Чисті функції

- Функції у ФП вільні від впливу побічних ефектів (немає змінних).
  - Чисті функції значно спрощують для тестування.
- Для запису чистої функції в Python потрібно використовувати local-only код.
  - уникати оператору global.
  - уважно стежити за використанням nonlocal;
  - Чисті функції поширені в Python-програмах.
- Гарантувати чистоту функцій – задача нетривіальна - її легко порушити.
- У Python lambda – це чиста функція.
  - Це не дуже рекомендований стиль, проте така можливість існує.

```
>>> mersenne = lambda x: 2**x-1
>>> mersenne(17)
131071
```

Функції вищого порядку (Higher-order functions)

- Це функції, які приймають функцію в якості аргументу чи повертають функцію в якості значення.
  - Функції вищого порядку можна використовувати для створення композитних функцій з більш простих.
  - Розглянемо Python-функцію max().

```
>>> year_cheese = [(2000, 29.87), (2001, 30.12), (2002, 30.6), (2003,
30.66),(2004, 31.33), (2005, 32.62), (2006, 32.73), (2007, 33.5),
(2008, 32.84), (2009, 33.02), (2010, 32.92)]
>>> max(year_cheese)

(2010, 32.92)
```

  - Поведінка за замовчуванням: порівняти кожен кортеж у послідовності.
  - Поверне кортеж з найбільшим значенням на позиції 0.
  - Оскільки max() – функція вищого порядку, можна передати іншу функцію в якості аргументу.

```
>>> max(year_cheese, key=lambda yc: yc[1])

(2007, 33.5)
```

Незмінювані (immutable) дані

- Оскільки змінні не використовуються для відстеження стану (state) обчислень, зупинимось на незмінюваних об'єктах.
  - Можна використовувати кортежі та іменовані кортежі, щоб розробляти складніші структури даних, які є незмінюваними.
- Будемо уникати (майже) повністю визначення класів.
  - ФП просто не потребує об'єктів зі станом (stateful objects).
- Поширений шаблон проектування, що працює з незмінюваними об'єктами – функція wrapper().
  - Список кортежів – досить поширена структура даних.
- Утворюють список кортежів зазвичай одним з двох способів:
  - **Використовуючи функції вищого порядку**: застосовуємо lambda як аргумент функції max() function: max(year_cheese, key=lambda yc: yc[1])
  - **Використовуючи шаблон Wrap-Process-Unwrap**: це виклик у вигляді unwrap(process(wrap(structure)))

Wrap-Process-Unwrap

```
>>> max(map(lambda yc: (yc[1],yc), year_cheese))
(33.5, (2007, 33.5))
>>> _[1]
(2007, 33.5)
```

- 1) wrap за допомогою map(lambda yc: (yc[1],yc), year_cheese). Перетворить кожен елемент у кортеж з ключем, що відповідає початковому значенню. Тут comparison key – просто yc[1].
- 2) виконати обробку за допомогою функції max(). Тут поведінка функції max() за замовчуванням нам підходить – кожен елемент є кортежем із значенням на нульовій позиції.
- 3) unwrap за допомогою звернення по індексу [1]. Бере другий елемент кортежу, вибраного функцією max().

- Деякі мови програмування для такого шаблону передбачають спеціальні функції, на зразок fst() і snd().
- Скоригуємо приклад:
  - snd= lambda x: x[1]
- snd(max(map(lambda yc: (yc[1],yc), year_cheese)))
  - Функція snd() бере другий елемент кортежу, що спрощує код для читання.
  - map(lambda… , year_cheese) бралось для обгортання raw data items.
  - max() для обробки
  - snd() для обирання другого елементу з кортежу.

Строгі (Strict) та нестрогі (non-strict) обчислення

- Ефективність ФП витікає, зокрема, з можливості відкладання (defer) обчислень до моменту, коли вони знадобляться.
  - У Python логічні оператори (and, or, if-then-else) - нестрогі.
  - Інколи їх називають short-circuit operators, оскільки їм не потрібно знаходити значення всіх аргументів, щоб визначити результат.

```
>>> 0 and print("right")
0
>>> True and print("right")
right
```

- Без логічних операторів вираз обчислюється негайно.
  - Послідовність інструкцій також виконується у строгому порядку.
  - Списки та кортежі літералів вимагають негайних обчислень.
- Після створення класу методи визначаються у строгому порядку.
  - З точки зору класу його методи за замовчуванням збираються у словник, а порядок не підтримується після їх створення.
  - При постачанні 2 одноіменних методів останній з них зберігається у зв'язку зі строгим порядком evaluation.

Рекурсія замість явного loop state

- Функціональні програми майже не застосовують цикли та відстеження стану.
  - Вони покладаються на рекурсивні функції.
  - У деяких мовах програми, що написані рекурсивно, заміняються компілятором на цикли за допомогою Tail-Call Optimization (TCO).
- Приклад: пошук взаємопростих чисел.

$$prime(n) = \forall x \left[\left(2 \le x < 1 + \sqrt{n} \right) and \left(n \left(\text{mod } x \right) \neq 0 \right) \right]$$

  - У Python: `not any(n%p==0 for p in range(2,int(math.sqrt(n))+1))`
  - Більш прямий математичний запис у Python використовуватиме all(n%p != 0…), проте це вимагає строгих обчислень усіх значень p.
  - Версія «not any» може завершитись раніше, якщо буде знайдено значення True.
- Приклад має цикл всередині, тому не є прикладом функціонального програмування без станів.

Перепишемо формулу у функцію, яка працює з колекцією значень

- Чи є число n взаємопростим з будь-яким числом з діапазону $[2, 1+\sqrt{n})$?
  - Обмежимось натуральними числами: неявно обріжемо дробову частину у значення кореня.
- Переформулюємо визначення простого числа:

$$\text{prime}(n) = \neg\text{coprime}\left(n, \left[2, 1+\sqrt{n}\right)\right), \text{ given } n > 1.$$

$$\text{coprime}\left(n, [a,b)\right) = \begin{cases} \text{True} & \textbf{if } a = b \\ n \pmod a \neq 0 \wedge \text{coprime}\left(n, [a+1, b)\right) & \textbf{if } a < b \end{cases}$$

```
def isprimer(n):
    def isprime(k, coprime):
        """Is k relatively prime to the value coprime?"""
        if k < coprime*coprime: return True
        if k % coprime == 0: return False
        return isprime(k, coprime+2)
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False
    return isprime(n, 3)
```

- Це приклад хвостової рекурсії.
  - Рекурсивний виклик знаходиться наприкінці функції.

Виникають 2 проблеми

- 1) Python накладає межу на рекурсивне вкладення – 1000.
  - Змінити межу можна за допомогою функції sys.setrecursionlimit().
  - Збільшувати межу суттєво не рекомендується: можуть виникнути обмеження пам'яті при роботі ОС та збій інтерпретатора Python.
- 2) Компілятор Python виконує TCO-оптимізацію.
- При запуску рекурсивної версії isprimer() для чисел понад 1,000,000 рекурсивна межа буде повністю вичерпана.
  - При перевірці на взаємопостоту тільки з простими числами зупинка на 1000му простому числі (7919), найбільше число для перевірки - 62,710,561.
- Деякі функціональні МП оптимізують прості рекурсивні функції.
  - Компілятор може перетворити рекурсивні обчислення в методі isprimer(n, coprime+1) у цикл.
  - Оптимізація створює hash of call stacks; налагодження оптимізованих програм ускладнюється.
  - Python не виконує таку оптимізацію.
  - У Python при використанні генераторного виразу замість рекурсивної функції виконується ручна tail-call оптимізація.

ТСО здійснено як генераторний вираз

```
def isprime(p):
    if p < 2: return False
    if p == 2: return True
    if p % 2 == 0: return False
    return not any(p==0 for p in range(3,int(math.sqrt(n))+1,2))
```

- Функція range() включає багато принципів ФП, проте використовує генераторний вираз замість чистої рекурсії.
  - Алгоритм повільний для великих простих чисел.
  - Для складених чисел функція повертає значення досить швидко.
- Функціональне програмування піднімає багато інших тем:
  - ***Посилальна прозорість (Referential transparency):*** при розгляді лінивих (lazy) обчислень та різних оптимізацій, можливих у компільованих МП, важлива ідея - multiple routes до одного об'єкта. У Python це неважливо, оскільки відсутні relevant compile-time optimizations.
  - ***Каррінг (Currying):*** системи типів застосовують каррінг, щоб звести багатоаргументні функції на одноаргументні.
  - ***Монади (Monads):*** повністю функціональні конструкти, які дозволяють гнучко структурувати sequential pipeline of processing.

Знову про недоліки імперативного коду

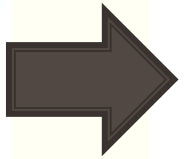
- Блок типового імперативного коду загалом складається з циклів, присвоювання змінним стану в цих циклах, операцій зі структурами даних (як стандартних, так і зі сторонніх бібліотек) та операторів галуження.
  - Проблеми часто виникають зі станом змінних та змінюваними структурами даних.
  - Вони часто досить добре моделюють поняття фізичного світу, проте визначити, в якому стані знаходиться змінна в деякий момент часу може бути складно.
- Одне з вирішень – сконцентруватись не на конструюванні колекцій, а на описі того, «з чого» структура даних складається.

Інкапсуляція

- Очевидний спосіб – рефакторинг коду, що помістить код для конструювання даних в ізольоване місце – функцію чи метод.

```
# configure the data to start with
collection = get_initial_state()
state_var = None
for datum in data_set:
    if condition(state_var):
        state_var = calculate_from(datum)
        new = modify(datum, state_var)
        collection.add_to(new)
    else:
        new = modify_differently(datum)
        collection.add_to(new)

# Now actually work with the data
for thing in collection:
    process(thing)
```



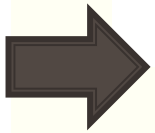
```
# tuck away construction of data
def make_collection(data_set):
    collection = get_initial_state()
    state_var = None
    for datum in data_set:
        if condition(state_var):
            state_var = calculate_from(datum, state_var)
            new = modify(datum, state_var)
            collection.add_to(new)
        else:
            new = modify_differently(datum)
            collection.add_to(new)
    return collection

# Now actually work with the data
for thing in make_collection(data_set):
    process(thing)
```

Включення (Comprehensions)

- Включення – вираз, який використовує ті ж ключові слова, що і блоки з циклами чи галуженнями, проте інвертує їх порядок, щоб сфокусуватись на даних, а не процедурі.

```
collection = list()
for datum in data_set:
    if condition(datum):
        collection.append(datum)
    else:
        new = modify(datum)
        collection.append(new)
```



```
collection = [d if condition(d) else modify(d)
              for d in data_set]
```

- Набагато важливіший ментальний зсув – уникнення потреби слідкувати за тим, у якому стані знаходиться колекція на даному етапі циклу.

- Словники та множини створюються відразу та цілковито, без повторних викликів .update() чи .add() у циклі.

```
>>> {i:chr(65+i) for i in range(6)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
>>> {chr(65+i) for i in range(6)}
{'A', 'B', 'C', 'D', 'E', 'F'}
```

Генератори

- Генераторні включення (Generator comprehensions) мають той же синтаксис, що і спискові включення, за винятком відсутності квадратних дужок—проте вони теж є лінивими виразами.
  - Вони просто описують, як отримуються дані (викликаючи .next() для об'єкту або проходячи по ньому в циклі).
  - Часто це зберігає пам'ять для великих послідовностей та відкладає обчислення до того моменту, коли вони дійсно потрібні.
  - Наприклад:

```
log_lines = (line for line in read_line(huge_log_file)
                    if complex_condition(line))
```

  - Імперативний аналог генератора
  - Навіть використання yield є в деякій мірі абстракцією над "iterator protocol", який лежить в основі.
  - Це можна зробити для класу, який має методи .__next__() та .__iter__().

```
def get_log_lines(log_file):
    line = read_line(log_file)
    while True:
        try:
            if complex_condition(line):
                yield line
            line = read_line(log_file)
        except StopIteration:
            raise

log_lines = get_log_lines(huge_log_file)
```

Об'єктно-орієнтований аналог генератора

```
class GetLogLines(object):
    def __init__(self, log_file):
        self.log_file = log_file
        self.line = None
    def __iter__(self):
        return self
    def __next__(self):
        if self.line is None:
            self.line = read_line(log_file)
        while not complex_condition(self.line):
            self.line = read_line(self.log_file)
        return self.line

log_lines = GetLogLines(huge_log_file)
```

- Включення значно більше уваги концентрує на "what", а імперативна версія зберігає увагу на питанні «як».

Найшвидша реалізація обчислення факторіалу – у функціональному стилі

- Виражає “what” для алгоритму:

```
from functools import reduce
from operator import mul
def factorialHOF(n):
    return reduce(mul, range(1, n+1), 1)
```

  - Загальна порада: it is good practice to look for possibilities of recursive expression—and especially for versions that avoid the need for state variables or mutable data collections—whenever a problem looks partitionable into smaller problems.
  - У Python в більшості випадків заміна рекурсією ітерацій це не є хорошою ідеєю.

Усунення циклів

- Дуже часто це погана ідея, як для читабельності, так і продуктивності коду, проте варто глянути, як просто системно це робити.
  - Якщо просто викликати функцію в циклі for, на допомогу може прийти функція вищого рівня map():

```
for e in it:
    func(e)
```

```
map(func, it)
```

  - Немає повторного прив'язування (rebinding) до змінної e, тобто не видно стану.
  - Схожа техніка доступна для функціонального підходу при послідовному виконанні програм.
- Імперативний підхід складається з інструкцій типу "спочатку зроби це, потім зроби це, а після цього – інші дії".
  - Якщо обгорнути окремі дії у функції, map() дозволить зробити таке:

```
do_it = lambda f, *args: f(*args)
# map()-based action sequence
map(do_it, [f1, f2, f3])
```

Усунення циклів

- Ми можемо комбінувати послідовність викликів функції з передачею аргументів від ітерованих об'єктів (iterables):

```
>>> hello = lambda first, last: print("Hello", first, last)
>>> bye = lambda first, last: print("Bye", first, last)
>>> _ = list(map(do_it, [hello, bye],
>>>                     ['David','Jane'], ['Mertz','Doe']))
Hello David Mertz
Bye Jane Doe
```

- Більш загальний випадок: всі аргументи у всі функції.

```
>>> do_all_funcs = lambda fns, *args: [
                            list(map(fn, *args)) for fn in fns]
>>> _ = do_all_funcs([hello, bye],
                     ['David','Jane'], ['Mertz','Doe'])
Hello David Mertz
Hello Jane Doe
Bye David Mertz
Bye Jane Doe
```

Заміна циклу while

```
# statement-based while loop
while <cond>:
    <pre-suite>
    if <break_condition>:
        break
    else:
        <suite>
```

```
# FP-style recursive while loop
def while_block():
    <pre-suite>
    if <break_condition>:
        return 1
    else:
        <suite>
    return 0

while_FP = lambda: (<cond> and while_block()) or while_FP()
while_FP()
```

- Один із способів додати кориснішу умову – дозволити while_block() повертати щось цікавіше та перевіряти це значення в умові переривання.

```
# imperative version of "echo()"
def echo_IMP():
    while 1:
        x = input("IMP -- ")
        if x == 'quit':
            break
        else:
            print(x)
echo_IMP()
```

```
# FP version of "echo()"
def identity_print(x):      # "identity with side-effect"
    print(x)
    return x
echo_FP = lambda: identity_print(input("FP -- "))=='quit' or
echo_FP()
echo_FP()
```

Усунення рекурсії

- Аналогічно до прикладу з факторіалом, інколи слід виконувати «рекурсію без рекурсії», використовуючи функцію tools.reduce() або інші folding-операції (інші “folds” не в стандартній бібліотеці Python, проте можуть конструюватись/з’являтись у сторонніх бібліотеках).
  - Часто рекурсія є просто способом комбінування чогось простішого з накопиченим intermediate result, саме це робить reduce() at heart.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Функції, ітератори та генератори

Рекурсія

- Функціональні програмісти часто виражають управляючий потік програми за допомогою рекурсії, а не циклів.
  - Так можна уникнути зміни стану будь-якої змінної чи структури даних в алгоритмі, а ще важливіше – змістити акцент обчислень на «що» замість «як».
  - З одного боку, рекурсія дозволяє ефективно проходити по послідовності елементів, хоч і виглядає не дуже “Pythonic.”
  - З іншого боку, Python відносно повільний при виконанні рекурсії і має обмежену глибину стеку.
  - Хоч останнє можна змінити за допомогою виклику limit() з sys.setrecursion, висока ймовірність, що це помилка.

- Мові Python не вистачає ***оптимізації хвостової рекурсії*** (***tail call elimination***), яка робить глибоку рекурсію більш ефективною в деяких мовах (Гвідо ван Россум заборонив і ще раз відмовив).

- Розглянемо тривіальний приклад, коли рекурсія просто вид ітерації:

```
def running_sum(numbers, start=0):
    if len(numbers) == 0:
        print()
        return
    total = numbers[0] + start
    print(total, end=" ")
    running_sum(numbers[1:], total)
```

  - Ітерація, яка просто виконує повторювані зміни стану змінної total, буде більш читабельна and moreover this function is perfectly reasonable to want to call against sequences of much larger length than 1000.

- Проте в інших випадках рекурсивний стиль часто виражає алгоритми більш стисло та інтуїтивно.

```
def factorialR(N):
    "Recursive factorial function"
    assert isinstance(N, int) and N >= 1
    return 1 if N <= 1 else N * factorialR(N-1)
```

```
def factorialI(N):
    "Iterative factorial function"
    assert isinstance(N, int) and N >= 1
    product = 1
    while N >= 1:
        product *= N
        N -= 1
    return product
```

Рекурсивний вираз ближче до “what” than the “how” of the algorithm.