**Лабораторна робота №4**

**Тема. Розширені можливості функцій. Рекурсивні функції.**

**Мета роботи**. Освоїти розширені можливості функцій. Рекурсивні функції.

**Зміст**.

1. Вивчення відомостей про розширені можливості функцій. Рекурсивні функції.

2. Виконання роботи.

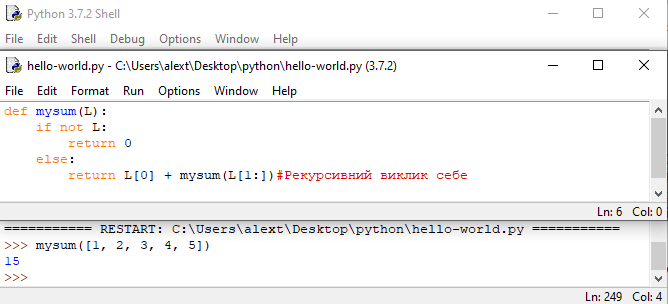
3. Отримання результату.

**Ключові положення.**

Рекурсія дозволяє програмами обходити структури, які мають довільні і непередбачувані форми та глибини, наприклад, при плануванні маршрутів в подорож, аналізі мови і проходження по посиланнях в веб-мережі. Рекурсія навіть є альтернативою нескладним циклам і ітераціям, хоча не обов'язково більш простою або ефективною.

**Підсумовування за допомогою рекурсії**

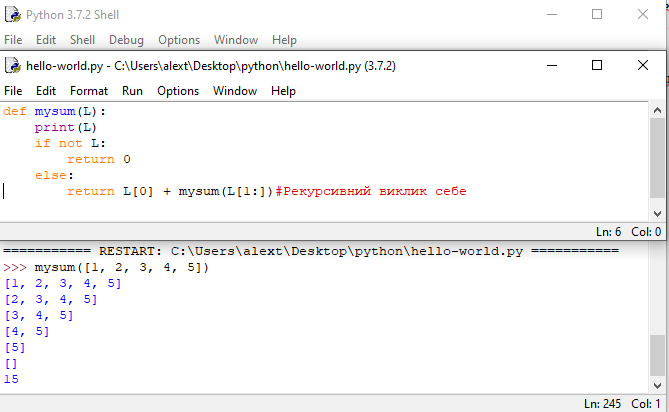
Щоб отримати суму списку (або іншій послідовності) чисел, ми можемо або використовувати вбудовану функцію sum, або написати власну більш спеціалізовану версію. Ось як може виглядати спеціальна функція підсумовування, в якій застосовується рекурсія:



На кожному рівні функція mysum рекурсивно викликає саму себе, щоб обчислити суму залишку списку, яка пізніше додається до елементу в голові списку.

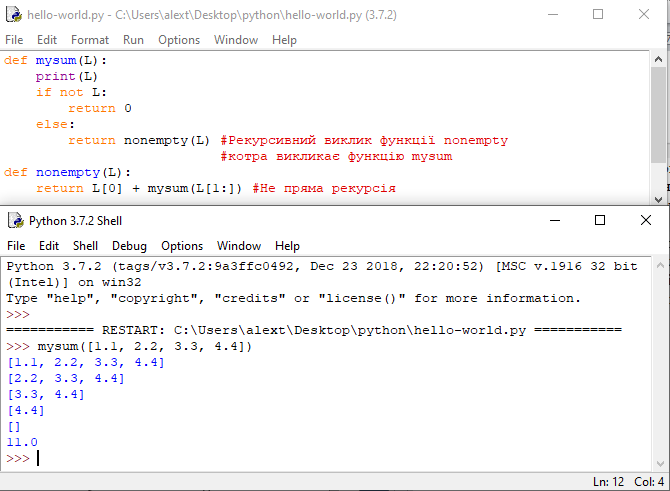
Коли список стає порожнім, рекурсивний цикл закінчується і повертається нуль. У разі використання рекурсії такого роду кожен відкритий рівень виклику функції має власну копію локальної області видимості функції в стек викликів часу виконання - тут це означає, що змінна L на кожному рівні різна.

Додамо в функцію вивід L на екран і запустимо програму знову, щоб відстежити поточний список на кожному рівні виклику.



Як легко помітити, підсумовуючий список на кожному рівні рекурсії стає все менше, поки остаточно не спорожніє - кінець рекурсивного циклу. Сума обчислюється при розкручуванні рекурсивних викликів по поверненню.

Рекурсія може бути прямою, як було показано в прикладах чи непрямою, як в наступному прикладі (функція, що викликає іншу функцію, яка знову викликає першу функцію). Сукупний ефект виявляється таким же, хоча на кожному рівні є два виклики функцій замість одного:

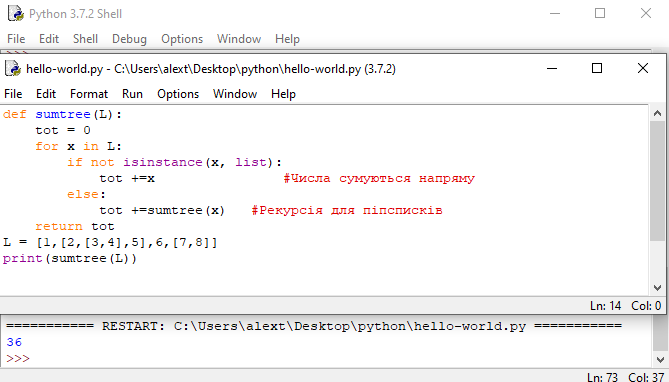


**Обробка довільних структур**

Рекурсія (або еквівалентні явні алгоритми, засновані на стеці) може вимагатися для обходу структур довільної форми. Як простий приклад рекурсії в такому контексті візьмемо задачу обчислення суми всіх чисел в структурі з вкладеними підсписками такого вигляду:

[1, [2, [3, 4], 5], 6, [7, 8]]

Прості оператори циклів тут не підходять, оскільки це не лінійна ітерація. Вкладених операторів циклу теж не буде достатньо, тому що підсписки можуть бути вкладеними на довільну глибину і в довільній формі. Нема ніякого способу дізнатися, скільки вкладених циклів необхідно написати для обробки всіх випадків. Натомість наступний код пристосовується до такого універсального вкладенню за рахунок застосування рекурсії для відвідування підсписків:



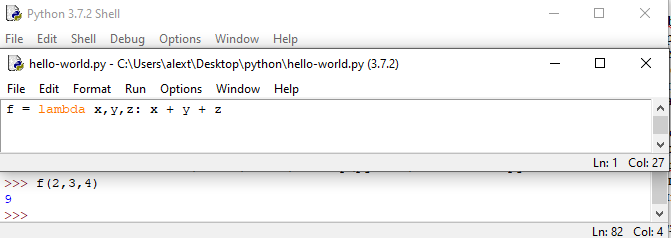
**Анонімні функції: вираження lambda**

Крім оператора def в Python також пропонується форма вираження, яка генерує об'єкти функцій. Через схожість з інструментом в мові Lisp вона називається lambda. Подібно def такий вислів створює функцію, яка буде викликатися пізніше, але повертає сам об'єкт функції, не привласнюючи його імені. З цієї причини вираження lambda іноді називають анонімними (тобто безіменними) функціями.

На практиці вони часто застосовуються для того, щоб вбудувати визначення функції в рядок або відкласти виконання порції коду.

Загальна форма lambda виглядає як ключове слово lambda, за яким слідують один або більше аргументів (дуже схоже на список аргументів, укладений в круглі дужки в заголовку def) і далі вираз після двокрапки:

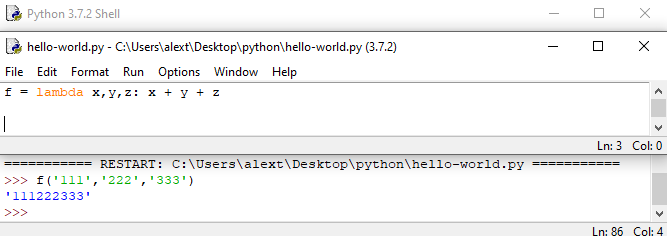
*lambda аргумент 1, аргумент2. . . аргумент№: вираз, що використовує аргументи*



Тут імені f присвоюється об'єкт функції, створений виразом lambda; так

працює оператор def, але він виробляє присвоювання автоматично.

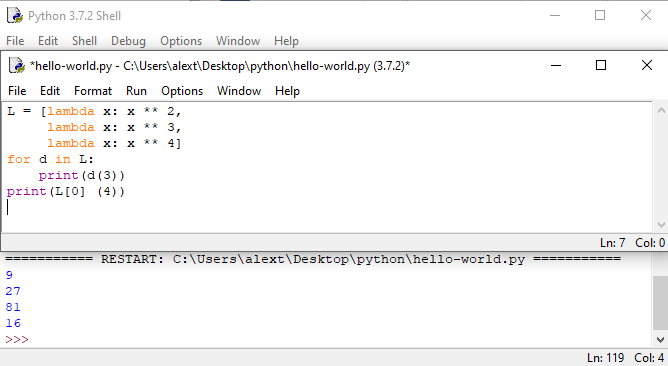
Як і в разі def, для аргументів lambda можна вказувати стандартні значення:



Функції lambda, на відміну від звичайної, не потрібна інструкція return, а в іншому, поводиться точно так же.

Вираз lambda корисний як свого роду коротке умовне позначення функції, яке дозволяє вбудовувати визначення функції всередину коду, де воно застосовується. Вираз lambda зовсім необов'язково використовувати завжди. Замість нього використовується оператор def, якщо функція вимагає мощі повних операторів, яку lambda не здатна легко надати. Але в сценаріях, де потрібно всього лише вбудовувати невеликі порції виконуваного коду в місцях їх застосування, вираження lambda виявляться більш простими кодовими конструкціями.

Вирази lambda також широко використовуються при реалізації таблиць переходів, які представляють собою списки або словники дій, що підлягають виконанню за запитом.

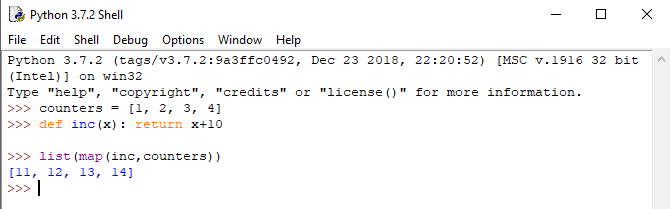


**Інструменти функціонального програмування**

Вбудована функція **map**

Одним з найбільш частих дій, які виконуються в програмах зі списками і іншими послідовностями, є застосування якоїсь операції до кожного елементу і накопичення результатів - вибір стовпців в таблицях бази даних, збільшення значень в полях із заробітною платою співробітників компанії, розбір вкладень в повідомленнях електронної пошти і т.д.

Вбудована функція **map** застосовує передану функцію до кожного елементу в ітеріруемому об'єкті і повертає список, що містить всі результати викликів функції.

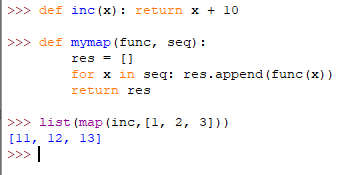


def inc(x): return (x + 10) – Функція яка буде виконуватись.

list(map(inc,counters))- Накопичення результатів.

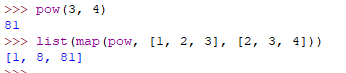
У списку **map** - викликає inc на кожному елементі списку і збирає все повернені значення в новий список. Для виведення всіх результатів використовується виклик list.

Через те, що таке використання куля еквівалентно циклам for, додавши трохи коду, ви можете отримати універсальну утиліту відображення:



Функцію **map** можна використовувати більш розвиненими способами,

ніж показано тут. Скажімо, отримавши в якості аргументів кілька послідовностей, **map** передає витягнуті з послідовностей елементи як індивідуальні аргументи функції **pow**:

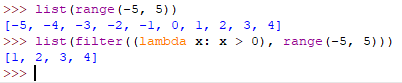


Вибір елементів з ітеріруемих об'єктів: **filter**

Функція map - початковий і щодо прямолінійний представник

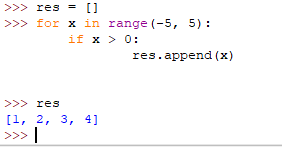
інструментального набору для функціонального програмування в Python. Її близькі родичі, **filter** та **reduce**, вибирають елементи з ітеріруемих об'єктів на основі перевірочної функції і застосовують функції до пар елементів відповідно.

Через повернення ітеріруемого об'єкта функція **filter** (подібно range) вимагає виклику **list** при відображенні всіх її результатів . Наприклад filter вибирає з послідовності елементи більше нуля:



Як і **map**, функція **filter** приблизно еквівалентна циклу for, але вона є

вбудованої, лаконічною і часто більш швидкою:

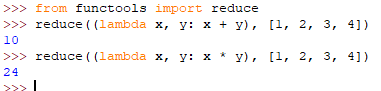


Також подібно **map** функцію **filter** можна емалювати за допомогою синтаксису спискового включення з часто більш простими результатами і за допомогою схожого генераторного виразу, коли бажано відкладати випуск результатів.



Комбінування елементів з ітеріруемих об'єктів: **reduce**

Виклик функції reduce, яка знаходиться в модулі functools, виглядає складніше. Вона приймає ітеріруемий об'єкт, що підлягає обробці, але сама не є ітеріруемим об'єктом, а повертає одиночний результат. Нижче показані два виклики reduce, які обчислюють суму і добуток елементів у списку:



На кожному кроці reduce передає поточну суму або добуток разом з черговим елементом зі списку зазначеної функції lambda. За замовчуванням початкове значення відповідає першому елементу послідовності.

**Лабораторне завдання**

1. За допомогою вбудованої функції **map** збільшити у 5 разів усі елементи списку А=[22, 33, 44, 55, 66]. Вивести новий список на екран.
2. За допомогою вбудованої функції **map** ввести масив[5, 6, 7, 8, 9] у ступінь [3, 4, 5, 8, №]. Де № - остання цифра номеру студента у списку групи. Перший елемент першого масиву возводиться у ступень першого елемента другого масиву, другий елемент першого масиву возводиться у ступень другого елемента другого масиву….
3. За допомогою lambda створити список з чотирьох функцій: перша збільшує число на №, друга зводить у ступінь 3, третя зменшує на одиницю, наступна зводить у ступінь 5. Де №- остання цифра номеру студента у списку групи. Використати число №+2 для перевірки функції.
4. За допомогою filter вибрати з генерованої послідовності на інтервалі [-15, 15] елементи менше №. Де №- остання цифра номеру студента у списку групи.
5. За допомогою функції reduce обчислити суму і добуток елементів у списку [12, 34, 45, 66, №]. Де №- остання цифра номеру студента у списку групи.
6. Написати програму розрахунку №+10 числа послідовності Фібоначчі. Де №- остання цифра номеру студента у списку групи.
7. Написати програму розрахунку суми №+10 перших чисел послідовності Фібоначчі. Де №- остання цифра номеру студента у списку групи.
8. Написати програму розрахунку факторіалу числа №+10. Де №- остання цифра номеру студента у списку групи.
9. За допомогою lambda створити список з чотирьох функцій: перша збільшує число у №, друга зводить у ступінь 6, третя зменшує на два, наступна зводить у ступінь 2. Де №- остання цифра номеру студента у списку групи. Використати число №+3 для перевірки функції.