§4 СПИСКИ ТА КОРТЕЖІ

Списки

Список у Python – це впорядкована колекція (тобто індексована послідовність) об'єктів довільних типів.

Елементи списку у літералах перераховуються через кому і записуються у квадратних дужках

```
[0, 1, 2, 3, 4, 5]
['first', 'second', 100, 1.234]
```

Списки у Python належать до **змінюваних** (**mutable**) типів.

Створення списків

Існує кілька способів створення списків. Розглянемо їх

1. Створення списку за допомогою літерала. Власне потрібно просто перерахувати елементи списку у **квадратних дужках через кому**

```
empty_list = []              # Порожній список

l = ['1', '2', 3.0, 4]       # Список з 4-х
                             # різнотипових елементів

M = [[1, 2, 3], [4, 5, 6]] # Список, що складається
                             # з двох списків
```

2. За допомогою перетворення у список іншої колекції використовуючи ключове слово `list`.

```
lst = list(collection)
```

тут `lst` – новий список, що будується на базі колекції `collection`.

```
l = list(range(5)) # Список [0, 1, 2, 3, 4]
s = list('list')   # Список ['l', 'i', 's', 't']
```

3. За допомогою оператора створення списку (також будемо використовувати термін спискоутворення). **Оператор створення списку** – це спосіб побудови нового списку на базі іншої колекції, до всіх елементів якої застосовується деякий вираз. Його синтаксис такий

```
new_lst = [expr(i) for i in collection if condition]
```

тут `new_lst` – новий список, що створюється, `collection` – деяка колекція.

Оператор створення списку працює таким чином:
змінна `i` послідовно пробігає всі елементи колекції `collection`. Якщо для поточного значення `i` виконується умова `condition`, то в список додається елемент `expr(i)`, що залежить від поточного значення ітератора.

Наприклад, для того, щоб створити список, що складається з квадратів натуральних чисел, що не перевищують 30 і є кратними 5: [25, 100, ...] потрібно виконати таку інструкцію

```
>>> l = [i ** 2 for i in range(5, 31) if i % 5 == 0]
>>> print(l)
[25, 100, 225, 400, 625, 900]
```

Якщо в операторі створення списку умова відсутня, то блок `if` опускають

Для прикладу створимо список, що складається з квадратів натуральних чисел, що не перевищують 4

```
>>> l = [i ** 2 for i in range(5)]
>>> print(l)
[0, 1, 4, 9, 16]
```

Індексація

Список, це послідовність, у якій всі елементи занумеровані, **починаючи з 0**. Номер елементу списку називається **індексом.** Наприклад, у такому списку

```
l = [1, 2, 3, 4]
```

`1` має індекс 0, `2` – індекс 1, `3` – індекс 2, `4` – індекс 3. Звернемо увагу на те, що останній елемент має номер на одиницю менший ніж кількість елементів у списку.

У Python існує механізм по-елементної роботи зі списком.

Для того, щоб звернутися до відповідного елемента списку, необхідно вказати номер цього елементу у квадратних дужках.

```
>>> l[0]
```

```
1
>>> l[3]
4
>>> l[4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Звернення до елементу списку за індексом надає прямий доступ до цього елементу у пам'яті. Таким чином, цей елемент можна замінити на інший елемент. При цьому список зміниться, оскільки списки належать до змінюваних типів даних

```
>>> l = [1, 2, 3, 4]
>>> l[0] = 111
>>> l
[111, 2, 3, 4]
```

Елемент списку може брати участь у різних операціях згідно з тими правилами які дозволені для об'єктів його типу

```
>>> l = [1, 2, 3, 4]
>>> suma = l[0] + l[1] + l[2] + l[3]
>>> print(suma)
10
```

Звернення до елементу списку може відбуватися з використанням **від'ємних індексів**. При цьому нумерація рахується з кінця списку з урахування, що –1-й елемент це останній елемент списку, –2-й – передостанній і т.д.

```
>>> l = [1, 2, 3, 4]
>>> l[-3]
2
>>> l[-1] = 555
>>> l
[1, 2, 3, 555]
```

Зрізи

Крім індексів, для прямого доступу до елементів списків у Python існують **зрізи**. Фактично зріз списку – це фрагмент вихідного списку.

Нехай `lst` деякий список. Тоді зріз

```
lst[start : end : step]
```

це підпослідовність елементів списку `lst`, що починається з елементу з індексом `start`, закінчується елементом з індексом `end-1` та будується з кроком `step`.

```
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[2:7:2]
[3, 5, 7]
```

Для аргументів зрізу можуть використовуватися типові значення (eng. by default). Так для параметра `start` типовим значенням є 0, для `end` – кількість елементів у списку, для `step` – значення 1. Отже, якщо необхідно побудувати зріз, що має для одного (або кількох) з аргументів типове значення, його можна опускати в коді. При чому параметр `step` можна опускати разом з другою двокрапкою. Наприклад

```
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[:3]
[1, 2, 3]
>>> l[::2]
[1, 3, 5, 7, 9, 11, 13]
```

Якщо при побудові зрізу, значення `start >= end` або діапазон значень індексів виявиться за межами списку, то зріз буде порожнім

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[5:3]
[]
>>> l[10:20]
[]
>>> l[1:1]
[]
```

Параметри зрізу також можуть бути від'ємними. У цьому випадку нумерація для `start` та `end` рахується з кінця починаючи від `-1`. Якщо ж `step` є від'ємним, то значення `start` та `end` фактично міняються місцями

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[::-1]
[6, 5, 4, 3, 2, 1]
>>> l[-1:1:-1]
[6, 5, 4, 3]
```

Використовуючи зрізи можна не лише видобувати підсписок, але і модифікувати список, додаючи або видаляючи елементи

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[0:0] = [0]    # вставляється список [0]
                    # перед нульовою позицією
>>> l
[0, 1, 2, 3, 4, 5, 6]
>>> l[3:] = []    # видаляються всі елементи починаючи з 3-го
>>> l
[0, 1, 2]
>>> l[3:] = [88, 99] # додаються в кінець список два елементи
>>> l
[0, 1, 2, 88, 99]
```

Операції над списками

Таблиця 4.1. Конкатенація і зчеплення

Операція	Опис
`l1 + l2`	конкатенація `l1` та `l2`, тобто створення нового списку, що складається з першого списку до кінця якого дописано другий список.
`l * n` `n * l`	`n` зчеплених копій списка `l`

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> s = [10, 20, 30, 40, 50, 60]
>>> l + s
[1, 2, 3, 4, 5, 6, 10, 20, 30, 40, 50, 60]
>>> l * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Таблиця 4.2. Перевірка входження елементу у список

Операція	Опис
`x in l`	Повертає значення `True`, якщо елемент `x` входить до списку `l`
`x not in l`	Повертає значення `True`, якщо елемент `x` **не входить** до

	списку `l`

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> 2 in l
True
>>> 222 in l
False
>>> 222 not in l
True
```

Таблиця 4.3. Аналіз списку

Операція	Опис
`len(l)`	Довжина `l`, тобто кількість елементів у списку.
`min(l)`	Найменший елемент списку `l`
`max(l)`	Найбільший елемент списку `l`
`l.index(x,i,j)`	Індекс першого входження `x` до `l`. (починаючи з індекса `i` та перед індексом `j`) Параметри `i` та `j` мають типові значення 0 та кількість елементів у списку відповідно.
`l.count(x)`	Кількість входжень `x` до `s`

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> len(l)
6
>>> min(l)
1
>>> l.index(3)
2
>>> l = [1, 2, 2, 2, 2, 3]
>>> l.count(2)
4
```

Таблиця 4.4. Модифікація списку

Операція	Опис
`s.append(x)`	Додає `x` у кінець списку `l`.
`l.clear()`	Вилучає всі елементи зі списку `l`
`l.copy()`	Повертає копію списку `l`
`l.extend(t)`	Розширює список `l` списком `t` (додає в кінець списку `l` список `t`)
`l.insert(i,x)`	Вставляє `x` у `l` у позицію, задану `i`
`l.pop(i)`	Повертає елемент з позиції `i` та вилучає його зі

	списку `l`
`l.pop()`	Повертає останній елемент списку `l`, та вилучає його зі списку `l`
`l.remove(x)`	Видаляє перший такий елемент `l`, для якого `l[i] == x`
`l.reverse()`	Переставляє елементи `l` у зворотному порядку
`l.sort()`	Впорядковує список `l` за неспаданням

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l.extend([20, 29])
>>> l
[1, 2, 3, 4, 5, 6, 20, 29]
>>> l.pop()
29
>>> l
[1, 2, 3, 4, 5, 6, 20]
>>> l.clear()
>>> l
[]
```

Наведені у таблиці 4.4 операції для списків можна реалізувати використовуючи зрізи та/або інші операції. Наприклад, інструкцію додавання елементу до списку

```
l.append(x)
```

можна записати як

```
l[len(l):len(l)] = [x]
```

Інструкцію

```
l.extend(t)
```

можна записати як

```
l[len(l):len(l)] = t
```

Інструкцію видалення всіх елементів зі списку

```
l.clear()
```

можна записати як

```
l[::] = []
```

Аналогічно можна записати й інші інструкції, що рекомендується виконати читачу самостійно.

Особливості роботи зі списками

Як було сказано раніше, списки відносяться до змінюваних (**mutable**) типів даних. Отже, через змінну, що є ім'ям списку надається безпосередній доступ, до пам'яті де список зберігає свої дані.

Інколи, особливо у новачків, не розуміння цього призводить до неочікуваних наслідків.

Розглянемо для прикладу такий код

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> s = l
>>> s.pop()
6
>>> l
[1, 2, 3, 4, 5]
```

Як бачимо список `l` змінився – зник елемент 6. Виникає цілком логічне запитання – Чому? Адже ми нічого не робили зі списком `l`.

Пояснення цього «феномену» полягає розумінні того як влаштовані змінні у Python. А саме: як було зазначено у першому параграфі – змінні це адреси пам'яті (посилання). Інструкція `s = l` провела копіювання адрес, а не списків і тому фактично `s` і `l` це два різних імені одного списку. Змінивши список `s` ми змінили і список `l`.

Якщо наведена вище поведінка не є бажаною, то список треба копіювати або інструкцією `copy` або з допомогою зрізів

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> s = l.copy()     # s = l[:]
>>> s.pop()
6
>>> l
[1, 2, 3, 4, 5, 6]
>>> s
[1, 2, 3, 4, 5]
```

Як бачимо у цьому випадку список лишився незмінним.

Введення списків з клавіатури

Список можна виводити на екран за допомогою інструкції `print`. Проте в Python не передбачено готової інструкції для введення списків з клавіатури.

Для цього використовується по-елементне введення, наприклад використовуючи цикл по проміжку значень.

```
# Блок введення списку з клавіатури
N = int(input("Кількість елементів списку "))
l = []              # Створюємо порожній список
for i in range(N):
    x = float(input("Задайте %d-й елемент списку " % (i)))
    l.append(x)    # Додаємо елемент x у список
# ==================================
# Виведення списку на екран
print(l)
```

Обхід списків

1. Ітерування списку

Оскільки список є колекцією, то всі його елементи можна послідовно перебрати використовуючи цикл **for**. Нехай маємо список `lst`. Тоді цикл

```
for elem in lst:
    process_iteration
```

послідовно виконує `process_iteration` для всіх елементів `elem` списку. Наприклад, знайдемо суму елементів списку, що містить натуральні числа від 1 до 5.

```
l = [1, 2, 3, 4, 5]
suma = 0                # Змінна у якій підраховується сума
for elem in l:
    suma += elem        # Додаємо кожний елемент списку
print(suma)
```

Приклад 4.1. Знайти найбільший елемент списку.

Розв'язок. Хоча для розв'язання цієї задачі у Python є вбудована функція `max`, розв'яжемо цю задачу без її використання, для того, щоб краще зрозуміти принципи роботи зі списками. Схожу задачу, проте без використання списків ми вже раніше розв'язували, тому наведемо програму без додаткових пояснень самого алгоритму

```
# Блок введення списку з клавіатури
N = int(input("Кількість елементів списку "))
l = []              # Створюємо порожній список
```

```
for i in range(N):
    x = float(input("Задайте %d-й елемент списку " % (i)))
    l.append(x)    # Додаємо елемент x у список
# ==================================
# Знаходження максимального елементу
max = l[0]  # 0-й елемент списку покладемо найбільшим
for i in range(1, len(l)):  # Проходимо всі елементи списку
    if max < l[i]:  # Якщо поточний максимум
                    # менший за поточний елемент
        max = l[i]  # змінюємо поточний максимум
print("Найбільних елемент списку - ", max)
```

2. Обхід списку за індексами

Крім по-елементного проходу списку, елементи списку можна перебрати за їхніми індексами. Нехай `lst` деякий список. Тоді цикл

```
for i in range(len(lst)):
    process_iteration      # Тут використовують lst[i]
```

послідовно виконує `process_iteration` для всіх елементів списку, звертання до яких потрібно за їхніми індексами, тобто `lst[i]`.

Приклад 4.2. Знайти індекс найбільшого елементу списку.

Розв'язок. Оскільки алгоритм введення списку з клавіатури дослівно повторює такий з попередньої задачі, то просто зазначимо його схематично – при необхідності замість крапок потрібно вставити фрагмент коду з попереднього прикладу. У циклі будемо проходити всі елементи по індексах. Будемо запам'ятовувати не лише найбільший поточний елемент, але і його індекс

```
# Блок введення списку з клавіатури
.....
# Знаходження максимального елементу разом з його індексом
max_elem = l[0] # 0-й елемент списку покладемо найбільшим
max_index = 0   # Індекс найбільшого елементу списку
for i in range(1, len(l)): # Проходимо всі елементи списку
    if max_elem < l[i]: # Якщо поточний максимум
                        # менший за поточний елемент
        max_elem = l[i] # змінюємо поточний максимум
        max_index = i   # змінюємо індекс максимального ел-ту
print("Найбільних елемент %d має індекс %d" % (max_elem, max_index))
```

3. Обхід списку за допомогою функції `enumerate`.

Функція `enumerate`, дозволяє пробігати циклом `for` список як колекцію і при цьому надає доступ до індексів елементів списку

```
for i, elem in enumerate(lst):
    process_iteration # Тут використовують i та elem
```

Тут `i` – індекс елементу списку, `elem` – поточний елемент колекції.

Отже, цикл пошуку найбільшого елемента у попередньому прикладі можемо переписати таким чином

```
for i, elem in enumerate(l):
    if max_elem < elem:
        max_elem = elem
        max_index = i
```

Приклад 4.3. Знайти суму елементів дійсного вектора.

Розв'язок. Як відомо вектор – це впорядкований набір дійсних чисел. Отже цю задачу зручно розв'язати, використовуючи список. Як і у попередньому прикладі, розіб'ємо задачу на два кроки – введення вектора з клавіатури і визначення суми елементів заданого вектора. Аналогічно попередньому прикладу код введення списку з клавіатури зазначимо схематично. Суму елементів вектора знайдемо пробігаючи його елементи циклом по колекції.

Таким чином, програма буде мати вигляд

```
# Блок введення списку з клавіатури
.....
# Обчислення суми елементів заданого вектора
suma = 0          # змінна у якій підраховується сума
for elem in l:    # пробігаємо список по-елементно
    suma += elem # додаємо до суми значення компоненти
print("Сума елементів вектора =", suma)
```

Моделювання роботи з матрицями за допомогою списків

Як відомо, будь-яку матрицю можна трактувати як вектор, що складається з векторів.

$$\begin{pmatrix} a_{00} & \dots & a_{0,N-1} \\ \dots & \dots & \dots \\ a_{N-1,0} & \dots & a_{N-1,N-1} \end{pmatrix} = \begin{pmatrix} (a_{00} & \dots & a_{0,N-1}) \\ \dots & \dots & \dots \\ (a_{N-1,0} & \dots & a_{N-1,N-1}) \end{pmatrix} =$$

Тоді, очевидно, що з матрицями можна працювати у Python як зі списками, елементами яких є списки дійсних чисел. Наприклад, матрицю

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

можна зобразити списком з двох елементів, кожен з яких містить відповідний рядок матриці

```
M = [ [1, 2, 3], [4, 5, 6] ]
```

Таке зображення списків наближає роботу з ними до роботи зі звичайними матрицями. Фактично єдиною відмінністю буде те, що в математиці, як правило номери рядків і стовпчиків починаються з одиниці, а елементи списків нумеруються з нуля. Отже, для того, щоб отримати елемент матриці розташований на перетині i-го рядка і j-го стовпчика необхідно брати відповідні елементи списків з номерами рядка і стовпчика $i-1$ і $j-1$ відповідно.

Наприклад, якщо для вищенаведеної матриці треба взяти елемент, що знаходиться у другому рядку і першому стовпчику (що має значення 4) необхідно взяти елемент `M[1][0]`.

```
>>> M = [ [1, 2, 3], [4, 5, 6] ]
>>> M[1][0]
4
```

Зображення матриць як списків, що містять вкладені списки підказує алгоритм обробки матриць, а саме за допомогою вкладених циклів по колекції. При цьому, ітератор зовнішнього циклу біжить "по рядках матриці", а ітератор внутрішнього циклу – по елементах відповідного рядка:

```
for row in M:           # row біжить по рядках матриці M
    for elem in row:    # elem пробігає елементи рядка row
        # тут можна проводити операції з
        # поточним елементом матриці M
```

Тут деяка `M` задана матриця.

Аналогічно, потрібно використовувати вкладені цикли для зчитування матриць з клавіатури

```
# Введення матриці з клавіатури
N = int(input("Кількість рядків матриці "))
L = int(input("Кількість стовпчиків матриці "))
M = []              # Створюємо порожню матрицю
for i in range(N):
```

```
    row = []       # Створюємо порожній список - рядок матриці
    # Заповнюємо рядок матриці з клавіатури
    for j in range(L):
        elem = float(input("M[%d][%d] = " % (i, j)))
        row.append(elem)
    M.append(row) # Додаємо заповнений рядок до матриці
```

Оскільки вводити дані у матрицю за допомогою вищенаведеного коду потрібно по-елементно, то таке введення матриці не зручне з точки зору практичного використання. Користувачу зручніше вводити зразу весь рядок матриці. Наведемо без пояснень (оскільки це виходить за межі розглянутих тем) код, який дозволяє вводити матрицю по рядках, розділяючи елементи рядка, символом (або кількома символами) пропуску

```
# Блок введення списку з клавіатури
N = int(input("Кількість рядків матриці "))
M = []             # Створюємо порожню матрицю
for i in range(N):
    # Вводимо рядок матриці row з клавіатури
    row = map(float, input("%d: " % i).split())
    M.append(row)  # Додаємо рядок до матриці
```

Введену матрицю можна вивести на екран за допомогою інструкції `print`. Проте у цьому випадку виведення буде здійснено як для списку

```
>>> print(M)
[[1, 2, 3], [4, 5, 6]]
```

Наведено приклад виведення матриці по рядках

```
for row in M:
    for elem in row:
        print(elem, end=" ")
    print()
```

Результатом виконання цього коду для заданої вище матриці M буде

```
1 2 3
4 5 6
```

Приклад 4.4. Визначити найменший з елементів квадратної дійсної матриці порядку N.

Розв'язок. Розіб'ємо задачу на дві частини. Перша – введення матриці з клавіатури. Друга – це пошук мінімуму у матриці. Алгоритм введення матриці з клавіатури розглянутий вище. Тому у програмі введення матриці позначимо схематично.

Для знаходження найменшого елементу матриці будемо пробігати всі елементи матриці за допомогою двох вкладених циклів, як показано вище. Використаємо додаткову змінну `min`, яка буде містити поточний мінімум серед елементів, які вже було розглянуто.

```
# Введення матриці M з клавіатури
……
# Пошук найменшого
min = M[0][0]
for row in M:
    for elem in row:
        if elem <= min:
            min = elem

print(min)
```

Приклад 4.5. Перевірити чи є задана матриця симетричною.

Розв'язок. Умова симетричності квадратної матриці має вигляд: $a_{ij} = a_{ji}$ для всіх i, j. Тому задачу можна розглядати як задачу пошуку таких i, j, що $a_{ij} \neq a_{ji}$. Якщо таких індексів немає, то матриця симетрична. У програмі використаємо змінну `res`, яка буде містити **`True`**, якщо не існує таких i, j, що $a_{ij} \neq a_{ji}$ і **`False`**, якщо такі i, j буде знайдено.

```
# Введення матриці M з клавіатури
……
# Визначення чи є матриця симетрична
n = len(M)
res = True
for i in range(n):
    for j in range(n):
        if M[i][j] != M[j][i]:
            res = False

print(res)
```

Кортежі

Кортеж у Python – це впорядкована незмінювана (immutable) колекція об'єктів довільних типів.

Елементи кортежу перераховуються через кому і записуються у круглих дужках

```
(0, 1, 2, 3, 4, 5)
('first', 'second', 100, 1.234)
```

Фактично кортеж це незмінюваний список.

Виникає запитання, навіщо потрібні кортежі, якщо є списки? Відповідь на це запитання така:

1. Кортежі надають захист колекції від несанкціонованої зміни.
2. Кортежі мають менший розмір у порівнянні зі списками.
3. Кортежі, на відміну від списків можна використовувати у якості ключа словника (певний тип даних Python, що буде розглянуто пізніше).

Робота з кортежами подібна до роботи зі списками. Власне для кортежів можна використовувати всі операції що і для списків, які не змінюють сам кортеж.

```
>>> l = (1, 2, 3, 4, 5)
>>> print(l[4])
5
```

Якщо ж потрібно змінити кортеж, то створюють новий кортеж на базі вихідного і здійснюють переприсвоєння подібно до простих типів. Спробуємо змінити останній елемент кортежу `l`.

```
>>> l[4] = 333
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Як бачимо система виводить повідомлення про помилку. Проте, зміна кортежу, через створення нового проходить без помилок

```
>>> l = l + (555, 666) # Додаємо два елементи до кортежу
>>> print(l)
(1, 2, 3, 4, 5, 555, 666)
```

Створення кортежів

Існує кілька способів створення кортежів.

1. Створення за допомогою літерала. Власне потрібно перерахувати елементи кортежу у **круглих дужках через кому**

```
empty_tuple = ()      # Порожній кортеж

one_element = (333,) # Кортеж з одного елементу.

l = ('1', '2', 3.0, 4) # Кортеж з 4-х елементів

M = ((1, 2, 3), (4, 5, 6)) # Кортеж, що складається
                           # з двох кортежів
                           # - незмінювана матриця
```

Зверніть увагу на створення кортежу з одного елементу.

```
one_element = (333,) # Кортеж з одного елементу.
```

Кома в дужках після числа **333** знаходиться не випадково. Справа у тому, що без неї Python буде розглядати дужки виключно як інструмент зміни пріоритету арифметичної операції, а не створення кортежу. І таким чином, без коми після першого елементу змінна `one_element` буде цілим числом, а не кортежом.

2. За допомогою перетворення у кортеж іншої колекції використовуючи інструкцію `tuple`.

```
tpl = tuple(collection)
```

Тут `tpl` – новий кортеж, що створюється на базі колекції `collection`.

```
t = tuple(range(5))         # Кортеж (0, 1, 2, 3, 4)
s = tuple([1, 2, 3, 4, 5]) # Кортеж (1, 2, 3, 4, 5)
```

Пакування колекцій

Пакування кортежу – це присвоєння виду

```
t = (x1, …, xN)
```

де `t` – ім'я запакованого кортежу, `x1, …, xN` – послідовність змінних (виразів, об'єктів тощо). Отже, пакування кортежу, це створення кортежу з декількох змінних або виразів.

Розпакування кортежу – це присвоєння виду

```
x1, …, xN = t
```

де `x1, …, xN` – послідовність змінних. `t` – деякий кортеж. Для розпакування кортежу, важливо, щоб кількість змінних, що стоїть зліва від знаку рівності дорівнювала кількості елементів у кортежі.

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> tpl = (a, b, c)  # Пакування кортежу
>>> print(tpl)
(1, 2, 3)
>>> x, y, z = tpl    # Розпакування кортежу у змінні x,y,z
>>> print(x, y, z)
1 2 3
```

Для списків операції пакування та розпакування також допустимі. Їхній синтаксис відрізняється лише тим, що потрібно використовувати замість круглих дужок – квадратні.

Пакування та розпакування кортежу може здійснюватися одночасно в одному виразі. Таким чином допустимі такі присвоєння:

```
x, y = u, v
```

Така інструкція присвоєння рівносильна такому ланцюгу присвоєнь

```
t = (u, v)
x, y = t
```

Приклад 4.6. Обміняти значення двох змінних.

Розв'язок. Класичне програмування для розв'язання поставленої задачі вимагає використання допоміжної змінної. Проте операції пакування/розпакування дозволяють розв'язати цю задачу (не рахуючи введення даних та виведення результату) одним рядком вихідного коду

```
a = int(input("a = "))
b = int(input("b = "))
a, b = b, a
print("a =", a, "b = ", b)
```

Операції пакування/розпакування допомагають зробити програми для обчислення елементів послідовностей заданих рекурентними співвідношеннями старших порядків значно простішими.

Приклад 4.7. Обчислити задане число Фібоначчі.

Розв'язок. Для спрощення коду, скористаємося операцією пакування/розпакування.

```
N = int(input('введіть N: '))
F2 = 1       #F2 - нульове число Фібоначчі
F1 = 1       #F1 - перше число Фібоначчі

for n in range(2, N+1):
    F2, F1 = F1, F1 + F2  # третя змінна стає непотрібною

print ('Результат', F1)
```

Приклад 4.8. Написати програму для обчислення найбільшого спільного дільника двох натуральних чисел.

Розв'язок. Ця задача була розв'язана у прикладі 3.4. Для спрощення та скорочення коду, скористаємося операціями пакування/розпакування.

```
N = int(input("Введіть перше число "))
M = int(input("Введіть друге число "))

# Запам'ятовуємо вхідні змінні
U = N
V = M
# Модифікуємо змінні U та V так, щоб U >= V
if U < V:
    U, V = V, U

# Алгоритм Евкліда з використанням пакування/розпакування
while V > 0:
    V, U = U % V, V

# Виводимо результат на екран
print("НСД(%d, %d) = %d" % (N, M, U))
```

Генератор-вирази для послідовностей

Генератор-вирази у Python дозволяють побудувати та опрацювати деяку послідовність на базі вже створеної колекції або згідно з деяким законом. Синтаксис генератор-виразу такий

```
generator = (expr(i) for i in collection if condition)
```

де `generator` – новий генератор-вираз, `collection` – деяка колекція, змінна `i` послідовно пробігає всі елементи колекції, `condition` – деяка умова, `expr(i)` – вираз, що залежить від `i`. Як і для оператора створення списку, блок з умовою не є обов'язковим.

Наступний блок коду, у якому визначено генератор-вираз `generator` виведе всі непарні числа з діапазону від 1 до 10.

```
generator = (k for k in range(1, 10, 2))
for a in generator:
    print(a)
```

Як можна помітити, генератор-вираз подібні до оператора створення списку – і той і інший будують послідовність на базі деякої колекції. Проте, на відміну від оператора створення списку, генератор-вирази не будують всю послідовність одразу – вони повертають послідовність по-елементно при кожному зверненні до об'єкту генератора. Це робить генератор-вирази незамінними у випадку коли треба обробляти великі обсяги даних якщо одночасно з цього масиву даних необхідно лише один або декілька елементів.

Приклад 4.9. Напишемо програму для наближеного обчислення числа π за формулою Грегорі

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

з використанням генератор-виразів.

Розв'язок. Знаходити число π за формулою Грегорі не найоптимальніший варіант, оскільки послідовність часткових сум для вищенаведеного ряду дуже повільно збігається. Тому для прийнятного результату треба брати значну кількість доданків. Відповідно, скористаємося генератор-виразом для побудови послідовності

$$\left\{1, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, \ldots\right\}$$

```
N = 99999  # Найбільший знаменник послід {1/k}

# Геренератор-вираз послідовності {1/k}
generator = (1.0 / k for k in range(1, N + 1, 2))

# Обчислення pi з використанням генератор-виразу
pi_4 = 0.0 # Змінна для pi/4
sign = 1.0 # Змінна для чергування знаку
```

```
for a in generator:
    pi_4 += sign * a
    sign = -sign

print(4.0 * pi_4)
```

Результатом виконання програми буде виведення такого значення

```
3.1415726535897814
```

Задачі для самостійної роботи

4.1. Дано список натуральних чисел. Знайти кількість:
a) парних компонент;
b) компонент, що діляться на 3 або 5;
c) нульових компонент;
d) компонент, що є простими числами.
e) компонент, що є числами Фібоначчі.

4.2. Дано список натуральних чисел. Використовуючи оператор створення списку, побудувати список, що містить:
a) всі парні елементи заданого списку;
b) всі елементи заданого списку, що є повними квадратами;
c) квадрати непарних компонент заданого списку.

4.3. Дано список чисел. Знайти кількість елементів списку,
a) більших b) менших
за задане число.

4.4. Обчислити суму елементів числового списку, які розташовані між позиціями максимального та мінімального елементів. Вважати, що всі елементи списку різні.

4.5. Знайти:
a) середнє арифметичне дійсного вектора;
b) довжину дійсного вектора;
c) відстань між двома точками у n-мірному евклідовому просторі;
d) скалярний добуток двох дійсних векторів.

4.6. Написати програми для:
a) множення дійсного вектора на число;
b) нормування дійсного вектора;
c) знаходження суми двох дійсних векторів;
d) обміну значень двох дійсних векторів;
e) пошуку однакових компонент.
f) перестановки компонент вектора у зворотному порядку.

4.7. Заданий список дійсних чисел $a_1, a_2, \ldots, a_n$. Написати програми для знаходження:
a) $\min(a_1, a_2, \ldots, a_n)$; b) $\max(|a_1|, \ldots, |a_n|)$;