§3 ЦИКЛІЧНІ ПРОГРАМИ

Цикли, поруч з лінійними програмами та розгалуженнями, є третьою з базових конструкцій структурного програмування.

Цикл – структура керування мови програмування, що здійснює повторення певної послідовності команд.

У Python цикли поділяються на два типи:

- Цикли з умовою продовження (**`while`**).
- Цикли-ітератори по колекції (**`for`**).

Цикл з умовою продовження

Цикл з умовою продовження, це структура керування, що здійснює повторення деякої послідовності інструкцій, поки виконується задана умова.

Надалі цикл з умовою продовження будемо називати просто циклом з умовою. Синтаксис циклу з умовою такий

```
while condition:
    process_iteration
```

де `condition` називається умовою продовження циклу, `process_iteration` – тілом (або ітерацією) циклу.

Правило роботи виконання циклу з умовою:

1. Python обчислює значення умови `condition`:
2. якщо `condition == True`, то інтерпретатор виконує інструкцію `process_iteration`, після чого все починається спочатку з пункту 1.
3. якщо `condition == False`, то інтерпретатор **не** виконує інструкцію `process_iteration` і завершує виконання циклу.

Іншими словами **`while`** виконує тіло циклу, поки умова циклу є істиною. Очевидно, що якщо умова продовження циклу завжди буде залишатися істинною, то цикл один раз почавшись, ніколи не закінчить свою роботу. Така ситуація називається «зациклюванням» або «зависанням» програми. Така ситуація виникає, наприклад, якщо тіло циклу не змінює умову продовження циклу. Відповідно, програмуючи цикли, необхідно слідкувати за тим, щоб цикли були скінченними.

Приклад 3.1. Визначити чи є введене з клавіатури число N простим.

Розв'язок. Згідно з означенням, простими називаються натуральні числа більші за 1, які діляться лише на 1 і на себе. Отже, для того, щоб вияснити, чи є число простим, необхідно поділити його на всі числа від 2 до $N-1$. Якщо так станеться, що число N поділиться без остачі хоча б на одне з них, то це і буде означати, що воно не просте.

Розіб'ємо задачу на кілька кроків. Цикл, що буде перебирати всі числа від 2 до $N-1$ буде виглядати таким чином

```
i = 2              # змінна i - лічильник, починається з 2
while i <= N-1:    # поки лічильник і не перевищить N-1
    # тут буде запрограмовано алгоритм
    i = i + 1      # збільшуємо лічильник i на 1
```

Як бачимо, тут використовується змінна `i`, яка називається лічильником. Цикл буде виконуватися доки змінна-лічильник буде меншою за значення виразу `N-1`. Цикли подібні до вищенаведеного будуть досить часто зустрічатися у наших програмах.

Задача, яку ми розв'язуємо зараз, це задача пошуку, оскільки нам фактично потрібно знайти таке число яке поділить без остачі число `N`. Існує безліч алгоритмів пошуку. Розглянемо найзагальніший з них. А саме. Використаємо у нашій програмі змінну-індикатор `prime`, яка просигналізує нам, що знайдено число, на яке ділиться вихідне число `N`, тобто що воно не є простим.

```
N = int(input("N = "))
prime = True   # Вважаємо спочатку, що число є простим
i = 2          # Починаючи з 2
while i <= N-1:        # до N-1 включно
    if N % i == 0:     # якщо N ділиться без остачі на i
        prime = False  # То воно не є простим
    i = i + 1          # Беремо наступне i

if prime:
    print("Число", N, "є простим")
else:
    print("Число", N, "не є простим")
```

Наведений код програми не буде оптимальним. Дійсно, якщо на певній `i`-й ітерації при буде виявлено, що число `N` не є простим, то продовжувати цикл перевірки вже не потрібно. Для оптимізації замінимо умову цикла на таку

```
while (i <= N-1) and prime: # до N-1 включно та
                            # всі попередні числа прості
```

Приклад 3.2. Знайти суму цифр заданого натурального числа.

Розв'язок. Для розв'язання задачі треба розкласти число на цифри.

Для цього, досить згадати, що ми використовуємо позиційну десяткову систему числення. Таким чином, беручи остачу від ділення числа на 10 будемо отримувати останню цифру числа, а діленням націло на 10 прибирати цю цифру з запису числа. Наприклад,

$$256 \,\%\, 10 \;=\; 6,$$
$$256 \,//\, 10 \;=\; 25.$$

Очевидно, що повторюючи ці дві операції поки число лишається більшим за нуль і підсумовуючи у деякій змінній отримані остачі отримаємо бажаний результат. Програма буде мати вигляд

```
N = int(input("N = "))
suma = 0              # Змінна, у якій буде міститися сума
while N > 0:          # Поки число не нульове (має цифри)
    last = N % 10     # Визначаємо останню цифру числа
    suma += last      # Додаємо її до змінної suma
    N //= 10          # Прибираємо останню цифру числа
print("Сума цифр заданого числа є", suma)
```

Приклад 3.3. Дано натуральне число K. Скласти програму знаходження всіх натуральних чисел менших або рівних K, які при піднесенні до квадрата дають паліндром.

Розв'язок. Натуральне число називається паліндромом, якщо його запис читається однаково зліва направо і справа наліво. Наприклад, числа

$$12321, 626, 10001$$

є паліндромами.

Для чисел $k = 1, \ldots, K$ будемо порівнювати значення $N = k^2$ із числом P, що є записом числа N у зворотному порядку.

Розіб'ємо задачу на два етапи. Для початку напишемо програму, яка будує число P, що є записом числа N у зворотному порядку. Ця програма, майже повністю повторює програму з попереднього прикладу. Єдина відмінність полягає у тому, що будемо шукати не суму числа, а будувати число P з огляду на позиційну десяткову систему числення:

$$625 = (62) * 10 + 5 = \big((6) * 10 + \ 2\big) * 10 + 5$$

Отже вихідний код, для обернення числа буде виглядати таким чином

```
# Вважаємо, що змінна N задана
P = 0      # P буде містити обернений запис числа N
while N > 0:
    last = N % 10
    P = P * 10 + last
    N //= 10
```

Після виконання цього коду, у змінній `P` буде міститися число `N` записане у зворотному порядку.
Нарешті напишемо програму, що розв'язує поставлену задачу.

```
K = int(input("K = "))
k = 1
while k <= K:
    N = k * k

    # Будуємо обернений запис числа N
    P = 0  # P буде містити обернений запис числа N
    while N > 0:
        last = N % 10
        P = P * 10 + last
        N //= 10

    # Якщо обернений запис числа N збігається з N=k**2
    if P == k * k:
        print(k) # Виводимо k на екран
    k += 1
```

Приклад 3.4. Написати програму для обчислення найбільшого спільного дільника двох цілих чисел за допомогою алгоритма Евкліда.

Розв'язок. Для розв'язання задачі скористаємося модифікованим алгоритмом Евкліда, у якому послідовне віднімання замінюється знаходженням остачі від ділення.

```
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 в
# U було більше число, а у V - менше
```

```
if U < V:
    P = U
    U = V
    V = P
# Запускаємо алгорим Евкліда
while V > 0:
    P = V
    V = U % V
    U = P
# Виводимо результат на екран
print("НСД(%d, %d) = %d" % (N, M, U))
```

Цикл по колекції.

Колекція – це структура, що містить скінченний набір елементів, до яких реалізовано доступ.

Колекції розрізняють в залежності від типу елементів у структурі, способів зображення її елементів у пам'яті, а також доступу до елементів.

Прикладом колекції може бути послідовність

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Ітератор – це інтерфейс, що надає послідовний доступ до елементів колекції та навігацію по ним.

Для перебору всіх елементів колекції у Python використовується цикл по колекції елементів.

Цикл по колекції – це структура керування, що здійснює повторення деякої послідовності інструкцій, для всіх елементів колекції.

Через призначення, яке виконують цикли по колекції їх часто називають «для всіх» (eng. "for each").

Синтаксис циклу по колекції такий

```
for iterator in collection:
    process_iteration
```

змінна `iterator` це ітератор, який послідовно проходить всі елементи колекції `collection`. `process_iteration` – тіло циклу, що виконується для кожного поточного значення змінної `iterator` з колекції.

Розглянемо приклад використання циклу по колекції, котра є послідовністю натуральних чисел від 1 до 9. Цикл по колекції у цьому прикладі використовується для того, щоб підрахувати суму чисел колекції.

```
suma = 0
collection = {1, 2, 3, 4, 5, 6, 7, 8, 9} # колекція
for i in  collection :                   # i – ітератор
    suma += i
print(suma)
```

Однією з найпростіших, проте дуже важливих колекцій у Python є арифметична прогресія натуральних чисел

$$a_0 = b,\ a_1 = a_0 + d, \dots, a_n = a_{n-1} + d$$

Наприклад, якщо $b = 1, d = 2$, то колекція, найбільший член якої менший за число $c = 10$, буде

$$\{1,3,5,7,9\}$$

У Python для побудови такої колекції існує спеціальна функція

```
range(b,c,d)
```

вона повертає впорядковану колекцію, що містить арифметичну прогресію, перший член якої `b`, останній член менший за `c`, а різниця прогресії дорівнює `d`. Наприклад, для створення колекції, що містить непарні числа, від 3 до 11 треба викликати функцію

```
range(3,12,2)
```

Якщо різниця прогресії $d = 1$, то `d` можна опускати:

```
range(b,c)
```

Крім того, якщо перший член прогресії $b = 0$, то `b` можна також опускати:

```
range(c)
```

Отже, попередній приклад, підрахунку суми натуральних чисел від 1 до 9 можемо переписати у такий спосіб:

```
suma = 0
for i in range(1, 10):
    suma += i

print(suma)
```

Приклад 3.5. Знайти суму парних чисел з діапазону від 1 до N.

Розв'язок. Для розв'язання задачі треба згенерувати колекцію парних чисел, після чого просумувати її члени

Інструкція `range(2, N + 1, 2)` створює колекцію парних чисел з діапазону [2, N]. Таким чином, для розв'язання задачі досить підсумувати члени створеної колекції.

```
N = int(input("N = "))
suma = 0
for i in range(2, N + 1, 2):
    suma += i
print("Сума парних чисел від 1 до %d є %d" % (N, suma))
```

Цикл **for** у комплексі з інструкцією `range` часто використовується для того, щоб виконати певну інструкцію наперед задану кількість разів. Наприклад, інструкція

```
for i in range(10):
    process_iteration
```

виконає тіло циклу `process_iteration` рівно 10 разів, оскільки ітератор циклу (який у цьому випадку буде лічильником) `i` пробігає значення з діапазону від 0 до 9.

Приклад 3.6. Скласти програму для обчислення добутку двох натуральних чисел m і n, використовуючи тільки операцію додавання.

Розв'язок. Легко помітити, що

$$m \cdot n = \underbrace{n + \ldots + n}_{m \text{ разів}}$$

Таким чином, програма має вигляд

```
m = int(input("m = "))
n = int(input("n = "))
s = 0
for i in range(m):
```

```
    s += n
print(s)
```

Приклад 3.7. Дано натуральне число n. Написати програму для обчислення значень многочлена

$$y(x) = x^n + x^{n-1} + \ldots + x^2 + x + 1$$

при заданому значенні x.

Розв'язок. Розглянемо два способи розв`язання.

Спосіб 1. Позначимо $z_k = x^k, k \geq 0$. Тоді фрагмент вихідного коду

```
z = 1
for i in range(1, n + 1):
    z *= x
```

забезпечить послідовне обчислення у змінній `z` значень $z_0, z_1, \ldots, z_n$. Таким чином, отримаємо програму

```
x = int(input("x = "))
n = int(input("n = "))

z = 1
y = 1
for i in range(1, n + 1):
    z *= x
    y += z

print("y(%f) = %f" %(x, y))
```

Спосіб 2. Скористаємось схемою Горнера. Розставивши дужки таким чином:

$$y = x^n + x^{n-1} + \cdots + x^2 + x + 1 = = (\ldots(x + 1)x + 1)x + \ldots + 1)x + 1$$

отримаємо програму

```
x = int(input("x = "))
n = int(input("n = "))

y = 1
for i in range(1, n + 1):
    y = y * x + 1
```

```
print("y(%f) = %f" %(x, y))
```

Приклад 3.8. Нехай задано натуральне число $n \geq 1$ і послідовність чисел $a_1, \dots, a_n$. Знайти $\max(a_1, \dots, a_n)$.

Розв'язок. Для розв'язання задачі досить помітити, що

$$max(a_1, a_2, \dots, a_n) = max(\dots(max(max(a_1, a_2), a_3), \dots), a_n).$$

Нехай у змінній `a` міститься чергове число послідовності a_n, що вводиться з клавіатури. Оскільки кількість членів послідовності a_n відома, то для їхнього зчитування використаємо цикл по проміжку значень. Для визначення максимального числа використаємо змінну `max`. Таким чином, програма буде мати вигляд:

```
N = int(input("N = "))
# Зчитуємо 1-й член послідовності
a = float(input("Задайте член послідовності = "))

max = a                 # Вважаємо, що він є найбільшим
for i in range(1, N): # Цикл виконається рівно N-1 раз
    a = float(input("Уведіть член послідовності"))
    # У змінну max записуємо більше з числом max і
    # поточного члена зчитаного з клавіатури
    max = a if a > max else max

print("Найбільшим є число", max)
```

Приклад 3.9. Написати програму для обчислення подвійного факторіала натурального числа n : $y = n!!$

Розв'язок. За означенням

$$n!! = \begin{cases} 1 \cdot 3 \cdot 5 \cdot \ldots \cdot n, & n - \text{непарне}, \\ 2 \cdot 4 \cdot 6 \cdot \ldots \cdot n, & n - \text{парне}. \end{cases}$$

В обох випадках маємо як співмножники всі члени спадної арифметичної прогресії з різницею -2, які містяться між n та 1. Звідси програма

```
n = int(input("n = "))

p = 1
for k in range(n, 0, -2):
    p = p * k

print("%d!! = %d" % (n, p))
```

розв`язує задачу. Зауважимо, що при непарному n останнім значенням k буде 1, а при парному буде 2.

Переривання та продовження циклів

Для переривання виконання циклу (як **while** так і **for**) або ігнорування частини тіла циклу, у Python існують дві спеціальні команди **break** та **continue**.

Оператор **break** достроково перериває виконання циклу. Оператор **continue** починає наступний прохід циклу, минаючи невиконаний залишок тіла циклу.

Приклад 3.10. Задано послідовність чисел, що завершується числом 0, серед елементів якої є принаймні одне додатне число. Потрібно знайти найбільше серед додатних чисел.

Розв'язок. Зчитувати члени послідовності будемо у циклі. При цьому, якщо зчитане число дорівнює нулю, то будемо використовувати **break** для завершення виконання циклу. Якщо число менше нуля, то будемо використовувати **continue**, щоб пропустити це число і не враховувати його у обчисленні найбільшого.

```
max = 0     # Поточний найбільший член послідовності
while True:         # Завершення циклу буде через break
    a = float(input("Задайте член послідовності"))
    if a == 0:      # якщо введене число 0,
        break       # завершуємо цикл
    elif a < 0:     # якщо введене число < 0,
        continue    # переходимо на початок циклу
    max = a if a > max else max

print("Найбільшим є число", max)
```

До операторів циклів **while** та **for** може застосовуватися оператор **else**. Синтаксис його використання для циклу **while** такий

```
while condition:
    process_iteration
else:
    state_else
```

а для циклу **for** такий

```
for iterator in collection
```

```
    process_iteration
else:
    state_else
```

Оператор **else** перевіряє чи відбувся вихід з циклу за допомогою оператора **break**. Блок інструкцій state_else виконується лише у тому випадку, якщо вихід з циклу відбувся без допомоги оператора **else**.

Приклад 3.11. Задано послідовність чисел. Перевірити чи міститься серед них число a.

Розв'язок. Для зчитування членів послідовності скористаємося циклом **for**. При цьому, якщо зчитане число дорівнює заданому числу a, то виведемо на екран відповідне повідомлення та припинимо виконання циклу інструкцією **break**. Якщо числа a серед членів послідовності не знайдено, то цикл завершиться природнім чином і у блоці **else** можемо вивести повідомлення, що числа a серед членів послідовності не знайдено. Отже, програма буде мати вигляд

```
N = int(input("Задайте кількість членів послідовності "))
a = int(input("Задайте шукане число "))

for i in range(N):
    current = float(input("Задайте поточний елемент "))
    if current == a:
        print("Послідовність містить число ", a)
        break
else:
    print("Послідовність не містить число ", a)
```

Рекурентні співвідношення

Рекурентне співвідношення першого порядку

Нехай $\{a_n : n \geq 0\}$ деяка послідовність дійсних чисел.

Послідовність $\{a_n : n \geq 0\}$ називається заданою **рекурентним співвідношенням першого порядку**, якщо явно задано її перший член a_0, а кожен наступний член a_n цієї послідовності визначається деякою залежністю через її попередній член a_{n-1}, тобто

$$\begin{cases} a_0 = u \\ a_n = f(n, p, a_{n-1}),\ n \geq 1 \end{cases}$$

де u задане (початкове) числове значення, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування (зокрема у нашому випадку Python).

Для прикладу розглянемо послідовність $\{a_n = n!: n \geq 0\}$. Її можна задати рекурентним співвідношенням першого порядку. Дійсно, враховуючи означення факторіалу отримаємо

$$\begin{cases} a_0 = 1 \\ a_n = na_{n-1},\ n \geq 1 \end{cases}$$

Маючи рекурентне співвідношення можна знайти який завгодно член послідовності. Наприклад, якщо потрібно знайти a_5, то використовуючи рекурентні формули, послідовно від першого члена отримаємо

$$\begin{gathered} a_0 = 1 \\ a_1 = 1 \cdot a_0 = 1 \cdot 1 = 1 \\ a_2 = 2 \cdot a_1 = 2 \cdot 1 = 2 \\ a_3 = 3 \cdot a_2 = 3 \cdot 2 = 6 \\ a_4 = 4 \cdot a_3 = 4 \cdot 6 = 24 \\ a_5 = 5 \cdot a_4 = 5 \cdot 24 = 120 \end{gathered}$$

З точки зору програмування, послідовність задана рекурентним співвідношенням значно зручніша, ніж задана у явному вигляді. Для обчислення членів послідовностей, заданих рекурентними співвідношеннями, використовують цикли.

Нехай послідовність a_n задана рекурентним співвідношенням

$$\begin{cases} a_0 = u \\ a_n = f(n, p,\ a_{n-1}),\ n \geq 1 \end{cases}$$

Тоді, після виконання коду

```
a = u
for n in range(1, N +1):
    a = f(n, p, a)
```

у змінній `a` буде міститися значення елемента a_N послідовності

Приклад 3.12. Для введеного з клавіатури значення N обчислити $N!$

Розв'язок. Як було зазначено раніше послідовність $a_n = n!$ може бути задана рекурентним співвідношенням

$$\begin{cases} a_0 = 1 \\ a_n = na_{n-1},\ n \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо

```
N = int(input("N = "))
a = 1                          # a = u
for n in range(1, N+1):
    a = n * a                  # a = f(n, p, a)
print ("%d! = %d" % (N, a))# виводимо на екран результат
```

Приклад 3.13. Скласти програму для обчислення елементів послідовності

$$a_n = \frac{x^n}{n!},\ n \geq 0.$$

Розв'язок. Складемо рекурентне співвідношення для заданої послідовності. Легко бачити, що кожен член послідовності a_n є добутком чисел. Враховуючи це, обчислимо частку двох сусідніх членів послідовності. Для $n \geq 1$ отримаємо

$$\frac{a_n}{a_{n-1}} = \frac{x^n}{n!} \cdot \frac{(n-1)!}{x^{n-1}} = \frac{x}{n}$$

Звідки випливає, що для $n \geq 1$

$$a_n = \frac{x}{n} a_{n-1}$$

Отже ми отримали для послідовності a_n рекурентну формулу, у якій кожен член послідовності для всіх $n \geq 1$ визначається через попередній член a_{n-1}. Щоб задати рекурентне співвідношення, залишилося задати перший член a_0. Для цього просто підставимо 0 у вихідну формулу

$$a_0 = \frac{x^0}{0!} = 1.$$

Отже остаточно отримаємо рекурентне співвідношення першого порядку

$$\begin{cases} a_0 = 1 \\ a_n = \dfrac{x}{n} a_{n-1},\ n \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо програму

```
N = int(input("N = "))
x = float(input("x = "))
a = 1
for n in range(1, N+1):
    a = x / n * a          # можна так: a *= x / n
print ("a = ", a)          # виводимо на екран результат
```

Зауважимо, що нумерація членів послідовності інколи починається не з 0, а з деякого натурального числа m, тобто $\{a_n: n \geq \mathrm{m}\}$. Припустимо, що рекурентне співвідношення для цієї послідовності має вигляд

$$\begin{cases} a_m = u, \\ a_n = f(n, p, a_{k-1}), n \geq m + 1. \end{cases}$$

Тоді для того, щоб отримати a_N, необхідно замінити наведений вище алгоритм на такий

```
a = u
for n in range(m + 1, N +1):
    a = f(n, p, a)
```

який, отриманий заміною стартового значення у інструкції `range` на значення `m + 1`.

Приклад 3.14. Скласти програму обчислення суми:

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Розв'язок. Зазначимо, що задане співвідношення має сенс тільки для $n \geq 1$. Складемо рекурентне співвідношення. Помічаємо, що на відміну від попереднього прикладу, кожен член послідовності S_n є сумою елементів вигляду $1/k$, де k змінюється від 1 до n. Отже, для побудови рекурентного співвідношення знайдемо різницю двох сусідніх членів послідовності S_n. Для $n \geq 2$

$$S_n - S_{n-1} = 1/n$$

Підставляючи у вихідне співвідношення $n = 1$, отримаємо $S_1 = 1$. Отже, рекурентне співвідношення для послідовності S_n матиме вигляд:

$$\begin{cases} S_1 = 1 \\ S_n = S_{n-1} + \dfrac{1}{n}, \ n \geq 2 \end{cases}$$

Аналогічно до попереднього прикладу, враховуючи, що нумерація членів послідовності починається з 1, а не з нуля, отримаємо програму.

```
N = int(input("N = "))
S = 1
for n in range(2, N + 1):
    S += 1 / n
print("S = ", S)
```

Приклад 3.15. Скласти програму обчислення суми

$$S_n = \sum_{i=1}^{n} 2^{n-i} i^2,\ n \geq 1.$$

Розв'язок. Складемо рекурентне співвідношення для заданої послідовності. Підставляючи $n = 1$, отримаємо $S_1 = 1$. Щоб отримати вираз для загального члена, розкриємо суму для $n \geq 2$

$$S_n = 2^n \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(n-1)^2}{2^{n-1}} + \frac{n^2}{2^n} \right) =$$
$$= 2 \cdot 2^{n-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(n-1)^2}{2^{n-1}} + \frac{n^2}{2^n} \right) =$$
$$2 \cdot 2^{n-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(n-1)^2}{2^{n-1}} \right) + 2 \cdot 2^{n-1} \frac{n^2}{2^n} = 2 \cdot S_{n-1} + n^2$$

Отже, рекурентне співвідношення для буде мати вигляд

$$\begin{cases} S_1 = 1, \\ S_n = 2 \cdot S_{n-1} + n^2,\ n \geq 2. \end{cases}$$

і відповідно програма

```
N = int(input("N = "))
S = 1
for n in range(2, N + 1):
    S = 2 * S + n ** 2
print("S = ", S)
```

Рекурентні співвідношення старших порядків

Нехай $\{a_n : n \geq 0\}$ деяка послідовність дійсних чисел. m – деяке натуральне число більше за одиницю.

Тоді

Послідовність $\{a_n : n \geq 0\}$ називається заданою **рекурентним співвідношенням m-го порядку**, якщо

$$\begin{cases} a_0 = u,\ a_1 = v, \ldots, a_{m-1} = w, \\ a_n = f(n, p,\ a_{n-1}, \ldots, a_{n-m},),\ n \geq m \end{cases}$$

де $u, v, \ldots, w$ – задані числові сталі, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування.

Найпоширенішим прикладом послідовності заданої рекурентним співвідношенням 2-го порядку є послідовність чисел Фібоначчі. Перші два члени цієї послідовності дорівнюють одиниці, а кожен наступний член є сумою двох попередніх

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2},\ n \geq 2 \end{cases}$$

Як і у випадку рекурентного співвідношення першого порядку, маючи рекурентне співвідношення можна знайти який завгодно член послідовності.

$$\begin{gathered} F_0 = 1,\ F_1 = 1 \\ F_2 = F_1 + F_0 = 1 + 1 = 2 \\ F_3 = F_2 + F_1 = 2 + 1 = 3 \\ F_4 = F_3 + F_2 = 3 + 2 = 5 \\ F_5 = F_4 + F_3 = 5 + 3 = 8 \\ F_6 = F_5 + F_4 = 5 + 3 = 13 \end{gathered}$$

Для обчислення елементів послідовності, заданої рекурентним співвідношенням вищого порядку, застосовується інший підхід ніж для співвідношень першого порядку.

Алгоритм наведемо на прикладі співвідношення 3-го порядку. Нехай послідовність a_n задана рекурентним співвідношенням

$$\begin{cases} a_0 = u,\ a_1 = v, a_2 = w, \\ a_n = f(n, p,\ a_{n-1},, a_{n-2}, a_{n-3},),\ n \geq 3 \end{cases}$$

Тоді, після виконання коду

```
a3 = u  # a3 - змінна для (n-3)-го члену послідовності
a2 = v  # a2 - змінна для (n-2)-го члену послідовності
a1 = w  # a1 - змінна для (n-1)-го члену послідовності
for n in range(3, N + 1):
    # Обчислення наступного члену
    a = f(n, p, a1, a2, a3)
    # Зміщення змінних для наступних ітерацій
    a3 = a2
    a2 = a3
    a1 = a
```

у змінних `a` і `a1` буде міститися a_N, у змінній `a2` – a_{N-1} , а в змінній `a3` – a_{N-2}.

Звернемо увагу на той факт, що для обчислення членів послідовності заданої рекурентним співвідношенням першого порядку не потрібно жодних додаткових змінних – лише змінна у якій обчислюється поточний член послідовності. Для рекурентних співвідношень старших порядків, крім змінної, у якій обчислюється поточних член послідовності, необхідні ще додаткові змінні, кількість яких дорівнює порядку рекурентного співвідношення.

Приклад 3.16. Знайти N-й член послідовності Фібоначі

Розв'язок. Як було зазначено раніше послідовність чисел Фібоначчі F_n може бути задана рекурентним співвідношенням

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2},\ n \geq 2 \end{cases}$$

Оскільки послідовність Фібоначчі задана рекурентним співвідношенням другого порядку, то для того, щоб запрограмувати обчислення її членів, необхідно три змінних. Модифікувавши наведений вище алгоритм для обчислення відповідного члена послідовності заданої рекурентним співвідношенням третього порядку на випадок рекурентного співвідношення другого порядку, отримаємо програму

```
N = int(input("N = "))
F2 = 1
F1 = 1
for n in range(2, N + 1):
    F = F1 + F2
    F2 = F1
    F1 = F
print("F = ", F)
```

Приклад 3.17. Скласти програму для обчислення визначника порядку n:

$$D_n = \begin{vmatrix} 5 & 3 & 0 & 0 & \dots & 0 & 0 \\ 2 & 5 & 3 & 0 & \dots & 0 & 0 \\ 0 & 2 & 5 & 3 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 2 & 5 \end{vmatrix}.$$

Розв'язок. Легко обчислити, що

$$D_1 = 5;$$
$$D_2 = \begin{vmatrix} 5 & 3 \\ 2 & 5 \end{vmatrix} = 19.$$

Розкладаючи для всіх $n \geq 3$ визначник D_n по першому рядку отримаємо рекурентне співвідношення

$$D_n = 5D_{n-1} - 6D_{n-2}, n \geq 3.$$

Тоді, згідно з вищенаведеним алгоритмом, програма для знаходження N-го члена послідовності D_n буде мати вигляд

```
N = int(input("N = "))
D2 = 5    # 1-й член послідовності
```

```
D1 = 19  # 2-й член послідовності
for n in range(3, N + 1):
    D = 5 * D1 - 6 * D2
    D2 = D1
    D1 = D
print("D_%d = %d" % (N, D1))
```

Системи рекурентних співвідношень

Вищенаведена теорія рекурентних співвідношень легко узагальнюється на системи рекурентних співвідношень, якщо вважати, що послідовності у означеннях вище є векторними.

Розглянемо системи рекурентних співвідношень на прикладах

Приклад 3.18. Скласти програму для обчислення N-го члена послідовності, що визначається сумою

$$S_n = \sum_{i=0}^{n} \frac{x^i}{i!}, \ , \ n \geq 0.$$

Розв'язок. Розкриваючи суму побачимо, що для всіх $n \geq 1$

$$S_n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^{n-1}}{(n-1)!} + \frac{x^n}{n!} = S_{n-1} + \frac{x^n}{n!}$$

Отже послідовність S_n визначається рекурентним співвідношенням

$$\begin{cases} S_0 = 1, \\ S_n = S_{n-1} + \dfrac{x^n}{n!}, \ n \geq 1 \end{cases}$$

Позначимо

$$a_n := \frac{x^n}{n!}, \ n \geq 0.$$

У прикладі 3.13 для цієї послідовності було отримано рекурентне співвідношення. Тоді для вихідної послідовності S_n система рекурентних співвідношень матиме вигляд

$$\begin{cases} S_0 = 1, \ a_0 = 1 \\ a_n = \dfrac{x}{n} a_{n-1}, \ n \geq 1, \\ S_n = S_{n-1} + a_n, \ n \geq 1. \end{cases}$$

Отже, програма для знаходження N-го члена послідовності S_n буде мати вигляд

```
N = int(input("N = "))
x = float(input("x = "))
a = 1
S = 1
for n in range(1, N+1):
    a = x / n * a
    S = S + a
print ("S = ", S)
```

Приклад 3.19. Скласти програму для обчислення суми

$$S_n = \sum_{k=0}^{n} a^k b^{n-k}$$

Розв'язок. Рекурентне співвідношення можемо побудувати двома способами.

Спосіб 1. Очевидно, що $S_0 = 1$. Розкриваючи суму і групуючи доданки аналогічно до прикладу 3.15, отримаємо

$$\begin{cases} S_0 = 1, \\ S_n = b \cdot S_{n-1} + a^n, \qquad n \geq 1. \end{cases}$$

Введемо позначення $x_n = a^n$. Запишемо для послідовності $\{x_n : n \geq 0\}$ рекурентне співвідношення:

$$\begin{cases} x_0 = 1, \\ x_n = a \cdot x_{n-1}, \qquad n \geq 1. \end{cases}$$

Таким чином, отримаємо систему рекурентних співвідношень

$$\begin{cases} S_1 = x_1 = 1, \\ x_n = a \cdot x_{n-1}, \qquad n \geq 1. \\ S_n = b \cdot S_{n-1} + x_n, \end{cases}$$

Спосіб 2. Легко бачити, що послідовність S_n можна зобразити у вигляді

$$S_n = \frac{a^{n-1} - b^{n-1}}{a - b}, \qquad n \geq 1$$

Тоді система рекурентних співвідношень буде мати вигляд

$$\begin{cases} x_1 = a, y_1 = b, \\ x_n = a \cdot x_{n-1}, \\ y_n = b \cdot y_{n-1}, \qquad n \geq 1. \\ S_n = \dfrac{x_n - y_n}{a - b}, \end{cases}$$

Програма для знаходження N-го члена послідовності S_n, заданого рекурентним співвідношенням, котре отримано першим способом, має вигляд:

```
N = int(input("N = "))
a = float(input("a = "))
b = float(input("b = "))

S = x = 1
for n in range(1, N + 1):
    x = a * x
    S = b * S + x

print(S)
```

Приклад 3.20. Обчислити суму, задану рекурентним співвідношенням

$$S_n = \sum_{k=1}^{n} \frac{a_k}{2^k},$$

де $a_1 = a_2 = a_3 = 1$, $a_k = a_{k-1} + a_{k-3}$, $k \geq 4$.

Розв'язок. Звернемо увагу на те, що послідовність a_k задана рекурентним співвідношенням третього порядку. Введемо допоміжну послідовність $b_k = 2^k, k \geq 0$, для якої рекурентне співвідношення буде мати вигляд $b_1 = 1, b_k = 2b_{k-1}, k \geq 1$.

Тоді, поєднуючи алгоритми для визначення відповідних членів послідовностей, заданих рекурентними співвідношеннями першого і третього порядків, отримаємо програму

```
N = int(input("N = "))
a1 = a2 = a3 = 1  # Одночасна ініціалізація кількох
                  # змінних одним значенням
b = 1
S = 1 / 2
# обчислення перших трьох членів послідовності S
for k in range(1, min(4, N + 1)):
    b = 2 * b
    S = S + 1 / b
for n in range(4, N + 1):
    b = 2 * b
    a = a1 + a3
    S = S + a / b
    a3 = a2
    a2 = a1
    a1 = a
```

```
print(S)
```

Приклад 3.21. Обчислити суму, задану рекурентним співвідношенням

$$S_n = \sum_{k=0}^{n} \frac{a_k}{1 + b_k},$$

де

$$\begin{cases} a_0 = 1, \\ a_k = a_{k-1} b_{k-1}, \end{cases} \quad \begin{cases} b_0 = 1, \\ b_k = a_{k-1} + b_{k-1}, \end{cases} \quad k \geq 1.$$

Розв'язок. Послідовності $\{a_k\}$ і $\{b_k\}$ задані рекурентним співвідношеннями першого порядку, проте залежність перехресна. Використаємо по одній допоміжній змінній для кожної з послідовностей.

Тоді, програма для знаходження N-го члена послідовності S_n:

```
N = int(input("N = "))

S = 0.5
a = 1
b = 1
for n in range (1, N+1):
    a_k = a * b   # допоміжна змінна для a_k
    b_k = a + b   # допоміжна змінна для b_k
    a = a_k
    b = b_k
    S = S + a / (1 + b)

print(S)
```

Приклад 3.22. Обчислити добуток, заданий рекурентним співвідношенням

$$P_n = \prod_{k=0}^{n} \frac{a_k}{3^k},$$

де $a_0 = a_1 = 1, a_2 = 3$, $a_k = a_{k-3} + \frac{a_{k-2}}{2^{k-1}}$, $k \geq 3$.

Розв'язок. Послідовність $\{a_k\}$ задана рекурентним співвідношенням третього порядку. Тоді добуток P_n обчислюється за допомогою рекурентного співвідношення

$$\begin{cases} P_2 = 1/9, \\ P_k = P_{k-1} \cdot a_k / z_k, \end{cases} \qquad k \geq 3,$$

де z_k – k-й степінь числа 3, визначений рекурентним співвідношенням

$$\begin{cases} z_2 = 9, \\ z_k = 3z_{k-1}, \qquad k \geq 3. \end{cases}$$

Передбачивши змінну `t` для обчислення членів послідовності $\{t_k = 2^{k-1} : k \geq 3\}$, отримаємо програму

```
N = int(input("N = "))

P = 1.0 / 9.0
z = 9
t = 2
a2 = a3 = 1
a1 = 3
for n in range(3, N + 1):
    z = z * 3
    t = t * 2
    a = a3 + a2 / t
    a3 = a2
    a2 = a1
    a1 = a
    P = P * a / z

print(P)
```

Відшукання членів послідовності, що задовольняють умову

Досі ми будували програми, що знаходять значення члену послідовності за його номером. Проте, часто постає задача, коли потрібно знайти найперший член послідовності, що задовольняє певну умову. У такому разі цикл **`for`** по діапазону значень заміняється циклом з умовою **`while`**. Умова у цьому циклі є запереченням до умови, яка визначає коли потрібно припинити обчислення членів послідовності.

Розглянемо приклади

Приклад 3.23. Для довільного натурального $N \geq 2$ знайти найменше число вигляду 3^k, де k – натуральне, таке, що $3^k \geq N$.

Розв'язок. Розглянемо послідовність $a_k = 3^k, k \geq 0$. Легко бачити, що її можна задати рекурентним співвідношенням першого порядку

$$\begin{cases} a_0 = 1, \\ a_k = 3a_{k-1}, \qquad k \geq 1. \end{cases}$$

Отже, враховуючи, що послідовність a_k строго зростаюча, щоб виконати завдання задачі необхідно обчислювати члени послідовності a_k в циклі використовуючи вищенаведене рекурентне співвідношення, доки не знайдемо перший такий, що $a_k \geq N$. Відповідно умова у циклі, буде запереченням до $a_k \geq N$, тобто $a_k < N$. Далі очевидним чином маємо програму

```
N = int(input("N = "))

a = 1
while a < N:
    a = a * 3

print(a)
```

Приклад 3.24. Послідовність задана рекурентним співвідношенням

$$\begin{cases} x_0 = 1,\ x_1 = 0,\ x_2 = 1, \\ x_n = x_{n-1} + 2x_{n-2} + x_{n-3},\ n \geq 3. \end{cases}$$

Створити програму для знаходження найбільшого члена цієї послідовності разом з його номером, який не перевищує число a.

Розв'язок. Запишемо кілька перших членів заданої послідовності

$$1, 0, 1, 2, 4, 9, 19, 41, 88$$

Звернемо увагу на те, що ця послідовність є зростаючою. Тоді, для того, щоб знайти найбільший член цієї послідовності, що не перевищує задане число a, необхідно обчислювати члени цієї послідовності, доки не знайдемо перший такий член, що буде більшим за задане число a. Тоді, член послідовності, що вимагається умовою задачі – елемент послідовності, що передує знайденому. Наприклад, якщо $a = 30$, то перший член послідовності, що більший за число 30 є 41, а відповідно шуканий згідно з умовою задачі член послідовності – той який йому передує, тобто 19.

Нехай `x, x1, x2, x3` – змінні, що використовуються згідно до алгоритму для обчислення членів послідовності заданої вищенаведеним рекурентним співвідношенням третього порядку. Нагадаємо, що тоді у змінних `x, x1` будуть знаходитися поточні члени послідовності x_n. Тоді, умова циклу буде `x1 <= a`.

Таким чином, маємо програму

```
a = int(input("a = "))
x3 = 1
x2 = 0
x1 = 1
n = 2              # Номер поточного члену послідовності
```

```
while x1 <= a:    # Поки поточний член послідовності
    n += 1
    x = x1 + 2 * x2 + x3
    x3 = x2
    x2 = x1
    x1 = x

print ("x(%d) = %d <= %d = a" % (n - 1 , x2, a))
```

Наведемо результат виконання програми для числа $a = 30$.

```
a = 30
x(6) = 19 <= 30 = a
```

Наближені обчислення границь послідовностей

Одне з важливих призначень рекурентних співвідношень це апарат для наближених обчислень границь послідовностей та значень аналітичних функцій.

Нехай задано послідовність $\{y_n : n \geq 0\}$, така, що $y_n \to y, n \to \infty$.

Під наближеним з точністю ε значенням границі послідовності y_n будемо розуміти такий член y_N послідовності, що виконується співвідношення

$$|y_N - y_{N-1}| < \varepsilon$$

Вищенаведене означення не є строгим з точки зору чисельних методів. У загальному випадку математичний апарат наближеного обчислення границь послідовностей та значень алгебраїчних функцій перебуває поза межами цього посібника і вимагає від читача додаткових знань з теорії чисельних методів [17].

Отже, згідно з наведеним вище означенням, для того щоб знайти значення границі послідовності потрібно обчислювати елементи послідовності доки виконується умова

$$|y_N - y_{N-1}| \geq \varepsilon$$

Розглянемо застосування зазначеного підходу на прикладах.

Приклад 3.25. Скласти програму наближеного обчислення золотого перетину c, використовуючи:

а) границю

$$\Phi = \lim_{n\to\infty} \frac{F_n}{F_{n-1}}$$

де F_n – послідовність чисел Фібоначчі;

б) ланцюговий дріб

$$\Phi = 1 + \frac{1}{1 + \frac{1}{1+\ddots}}.$$

Розв'язок. Золотий перетин це число

$$\Phi \approx 1{,}6180339887..$$

яке має багато унікальних властивостей, причому не лише математичних. Це число можна зустріти як у різноманітних сферах діяльності людини (наприклад, у мистецтві, архітектурі, культурі) так і у оточуючому нас світі, зокрема фізиці, біології тощо. Для того, щоб переконатися у тому, що наш алгоритм працює правильно, скористаємося однією з властивостей золотого перетину, а саме:

$$\Phi - 1 = \frac{1}{\Phi}.$$

а) Розглянемо послідовність

$$\Phi_n = \frac{F_n}{F_{n-1}}, n \geq 1.$$

Знайдемо рекурентне співвідношення для c_n. Очевидно, що $c_1 = 1$, далі для $n \geq 2$ отримаємо

$$\Phi_n = \frac{F_n}{F_{n-1}} = \frac{F_{n-1} + F_{n-2}}{F_{n-1}} = 1 + \frac{1}{\frac{F_{n-1}}{F_{n-2}}} = 1 + \frac{1}{\Phi_{n-1}}.$$

б) Розглянемо послідовність

$$\Phi_n = 1 + \frac{1}{1 + \frac{1}{1+\ddots + \frac{1}{1}}},$$

що містить $n-1$ риску дробу. Очевидно, що для цієї послідовності рекурентне співвідношення буде таким же, як у пункті а).

Напишемо програму, що знаходить наближене з точністю ε значення границі послідовності Φ_n. Використаємо змінну `current` для обчислення поточного члену послідовності Φ_n і змінну `prev`, у якій будемо запам'ятовувати попередній член Φ_{n-1} цієї послідовності.

Тоді програма має вигляд

```
eps = 0.0000000001  # точність
prev = 0    # попередній член послідовності
current = 1 # поточний член послідовності

while abs(current - prev) >= eps:
    prev = current
    current = 1 + 1 / current

print("Φ =", current)  # Виводимо значення золотого перетину

# Перевірка результату згідно з властивостями
print("Φ - 1 =", current - 1)
print("1 / Φ =", 1 / current)
```

Наведемо результат виконання програми.

```
Φ = 1.618033988738303
Φ - 1 = 0.6180339887383031
1 / Φ = 0.6180339887543225
```

Приклад 3.26. За допомогою розкладу функції e^x в ряд Тейлора

$$y(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

обчислити з точністю $\varepsilon > 0$ її значення для заданого значення x.

Розв'язок. Позначимо загальний член вищенаведеного ряду через a_n, а його часткову суму

$$S_n = \sum_{i=0}^{n} \frac{x^i}{i!},$$

Очевидно, що $S_n \to e^x$, $n \to \infty$.

У прикладі 3.18 було отримано, що послідовність S_n визначається системою рекурентних співвідношень

$$\begin{cases} S_0 = 1,\ a_0 = 1 \\ a_n = \dfrac{x}{n} a_{n-1},\ n \geq 1, \\ S_n = S_{n-1} + a_n,\ n \geq 1. \end{cases}$$

Відповідно до означення, наведеного вище і з огляду на рекурентне співвідношення, під наближеним значенням границі послідовності S_n будемо розуміти такий член S_N , що виконується співвідношення

$$|S_N - S_{N-1}| = |a_N| < \varepsilon$$

Отже, програма буде мати вигляд

```
eps = 0.0000000001  # точність

x = float(input("x = "))
a = 1
S = 1
n = 0
while abs(a) >= eps:
    n += 1
    a = x / n * a
    S = S + a

print("exp(%f) = %f" % (x, S))
```

Звернемо увагу на те, що на відміну від цикла **for**, цикл **while** не має вбудованого лічильника. Тому, оскільки нам необхідно враховувати у формулі номер члена послідовності, то ми задали змінну n, яка відіграє роль лічильника.

Наведемо результат виконання програми.

```
x = 1.0
exp(1.000000) = 2.718282
```

Найпростіші методи розв'язання алгебраїчних рівнянь

Розглянемо алгебраїчне рівняння вигляду

$$f(x) = 0,$$

де функція $f(x)$ неперервна на відрізку $[a, b]$.

Розглянемо найпростіші наближені методи відшукання розв’язків рівнянь вищенаведеного вигляду на заданому відрізку $[a, b]$. До таких методів належать **метод бісекції** та **метод хорд**.

Метод бісекції або метод ділення відрізка навпіл, полягає у такому. Припустимо, що відомо, що на відрізку $[a, b]$ існує єдиний корінь цього рівняння. Тоді очевидно, що на кінцях цього відрізка функція набуває різних знаків. Поділивши відрізок навпіл, будемо брати ту з половин, для якої на кінцях функція буде набувати значення різних знаків. Якщо середина відрізка

виявиться нулем функції, то процес завершується. Якщо задана точність обчислення ε, то вищеописану процедуру поділу слід виконувати доти, доки довжина відрізка не стане меншою за ε, а наближеним значенням тоді вважають середину цього відрізка.

Метод хорд полягає в обчислені елементів послідовності $\{u_n : n \geq 0\}$, визначеної рекурентним співвідношенням

$$u_0 = a;$$
$$u_n = u_{n-1} - f(u_{n-1}) \cdot \frac{b - u_{n-1}}{f(b) - f(u_{n-1})}, \qquad n \geq 1$$

до виконання умови $|u_n - u_{n-1}| < \varepsilon$.

Приклад 3.27. Використовуючи метод бісекції знайти корінь рівняння $\tan x = 2x$ на відрізку $[0.5, 1.5]$ із заданою точністю ε.

Розв'язок. Згідно з вищенаведеним алгоритмом, для розв'язання рівняння розглянемо допоміжну функцію вигляду

$$f(x) = tan\, x - 2x.$$

Тоді програма буде мати вигляд

```
import math                 # підключення математичної бібліотеки
eps = 0.0000000000001       # точність
l = 0.5                     # лівий кінець відрізка
r = 1.5                     # правий кінець відрізка
# tan(l) - 2 * l < 0, tan(r) - 2 * r > 0

while r - l >= eps:
    x = (l + r) / 2.0  # середина відрізка [l,r]
    f = math.tan(x) - 2.0 * x
    if f == 0.0:
        break
    elif f < 0.0:
        l = x  # [l,r] = [x,r]
    else:
        r = x  # [l,r] = [l,x]

# корінь – середина останнього відрізка [l,r]
x = (l + r) / 2.0
print(x)
```

Задачі для самостійної роботи

3.1. Вивести на екран такий рядок:

```
n! = 1*2*3*4*5*...*n
```